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

Л.В. Городняя

LISP И ЕГО ДИАЛЕКТЫ

Препринт

Новосибирск 2025

Препринт посвящен результатам анализа и сравнения особенностей языка Lisp и наиболее успешных его диалектов (Scheme, Common Lisp, Racket, Clojure). Приведён краткий очерк истории языка Lisp, описаны его особенности, влияющие на практику программирования, и характеристики диалектов, направленных на решение конкретных проблем. Представлена таблица особенностей языка Lisp для представления результатов сравнения языка Lisp с его диалектами в подобных таблицах. Показаны их сходства и различия с оценкой эволюции и энтропии знания в истории языков программирования на примере рассмотренных языков.

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

*****

L.V. Gorodnyaya

LISP AND ITS DIALECTS

Preprint

Novosibirsk 2025

The preprint is devoted to the results of the analysis and comparison of the features of the Lisp language and its most successful dialects (Scheme, Common Lisp, Racket, Clojure). A brief outline of the history of the Lisp language is given, its features that influence programming practice and the characteristics of dialects aimed at solving specific problems are described. A table of Lisp language features is presented to present the results of comparison of the Lisp language with its dialects in similar tables. Their similarities and differences are shown with an assessment of the evolution and entropy of knowledge in the history of programming languages using the example of the languages considered.

Lisp и его диалекты

Lisp — гениальное творение Джона Маккарти (*John McCarthy*), значение этого языка для истории программирования и вычислительной техники трудно переоценить. Язык Lisp был создан в 1958 году в процессе развёртывания работ по искусственному интеллекту[1] и исследования новых задач программирования [1,2]. Математической основой языка Lisp является лямбда-исчисление Алонсо Чёрча (Alonzo_Church's lambda calculus), достаточное для полноты языка по Тьюрингу, что позволяет моделировать любые вычислимые средства и методы, а также парадигмы программирования. Это дало основания Дж. Маккарти называть Lisp универсальным языком программирования. Ключевая концепция языка Lisp — любую информацию для компьютерной обработки можно представлять с помощью символов и процесс обработки данных рассматривать как применение или определение функций над символьными формами, приспособленными к представлению основных понятий программирования. В отличие от большинства языков программирования, ориентированных на распределение памяти и изменение хранящихся в памяти данных, Lisp ориентирован на символьное представление данных, из которых строится программа, и вычисление выражений независимо от их расположения в памяти. В результате структура программы в языке Lisp представляется в виде структуры данных, что теперь называют гомоиконностью, обеспечивающей представление исходного текста программы непосредственно в виде абстрактного синтаксического дерева [3-13].

Идеи языка Lisp вызвали ревнивую критику как со стороны программистов, так и со стороны математиков[2]. Программисты не могли смириться с отсутствием в языке понятий «адрес», «машинное слово», «передача управления», «изменение состояний памяти», а также медленной обработкой списков в сравнении с быстрой обработкой векторов. Математиков смущала противоречивость некоторых построений с точки зрения классической математики, например, различие контекстов определения, вызова и вычисления функций[3]. В середине 1970-ых годов Дана Скотт (Dana S. Scott) опубликовал конструктивную теорию, смягчившую критицизм математиков[4].

1. Немного истории

Язык Lisp – один из старейших языков высокого уровня (ЯВУ), первый, использующий автоматическое управление распределением памяти, встроенную базу данных и операционную систему, систему разделения времени, компиляцию на лету, сборку мусора[5]. Таким образом программист был освобождён от преждевременной заботы о повторном использовании памяти в пользу продумывания методов решения своих основных задач. Приоритет интерпретации программ даёт возможность быстрой отладки в условиях полного динамического контроля хода вычислений, что освобождает от обязанности заранее объявлять типы данных, их время придёт после экспериментальной разработки алгоритма при переходе к компиляции и оптимизации отлаженных функций и к применению программы. Компилятор функций преобразует список, представляющий определение функции в виде абстрактного синтаксического дерева, в машинный код или байт-код для выполнения, сохраняя взаимодействие с интерпретатором, что теперь рассматривается как прецедент компиляции на лету. Такой

---

1 IPL – предшественник, предназначенной для автоматического вывода теорем математической логики.

2 Джон Маккарти не счёл нужным уделять внимание критике.

3 Funarg-проблема.

4 turing100.acm.org/lambda_calculus_timeline.pdf — Дана Скотт об истории λ-исчисления.

5 *Филд А., Харрисон П.* Функциональное программирование = Functional Programming.— М.: Мир, 1993. — 637 с. — ISBN 5-03-001870-0.

код может работать столь же быстро, как код, скомпилированный на обычных языках, таких как C.

Lisp позволяет создавать программы, динамически порождающие код, выполнять любые реализационные трюки, строить виртуальные машины, специализированные системы, диалекты и расширяющие язык пакеты. Концепции языка Lisp со временем кристаллизовались как парадигма функционального программирования[6], хотя реализация языка изначально поддерживает основные императивные черты, введённые ради сохранения привычного стиля программирования, а также ради возможности повышать надёжность и эффективность программирования использованием разных стилей определения одних и тех же функций. Это позволило языку Lisp стать одним из мощнейших мультипарадигмальных ЯВУ, он до сих пор применяется в практическом программировании, опробованные в языке Lisp средства и методы сохранены в его диалектах и унаследованы многими современными ЯВУ. Существует манга, посвящённая языку Lisp[7].

Исторически первой реализацией языка Lisp, включающей все базовые элементы языка, был интерпретатор, работавший на IBM 704, появившийся в октябре 1958 года. В 1960 году была реализована интерактивная система «Lisp 1», включающая в себя интерпретатор, редактор исходного кода и отладчик, позволявшая выполнять полный цикл работ над программой [4]. Язык Lisp, включая интерпретатор и компилятор, был описан в виде формализма на самом языке Lisp, «Lisp on Lisp» стал одним из наиболее употребимых в литературе по теории программирования. Ещё одним технологическим новшеством, появившимся в связи с реализацией системы «Lisp 1» был изобретённый Дж. Маккарти механизм, позволявший запускать интерпретатор языка Lisp одновременно с выполнением вычислительных работ в пакетном режиме - система разделения времени [4].

2. Основные положения

Любой информационные процесс на компьютере можно понимать как преобразование данных[8], а именно — символьных форм (S-выражение). Символьные формы в языке Lisp могут быть атомарными или составными. Атомы не делятся на части базовыми средствами языка, выполняют роль имён функций и переменных и выглядят как идентификаторы. Кроме имён переменных и функций могут быть и другие роли атомов, сопровождаемых расширяемыми списками свойств, что позволяет им работать как встроенная база данных. В работах по представлению знаний такие конструкции называют «фреймы», а свойства - «слоты».

Кроме того, к атомам отнесены числа и строки — само-определимые данные, смысл которых задан их представлением. Целые числа и строки в языке Lisp имеют произвольную длину. Можно представлять дроби (3/8) и вещественные числа с высокой точностью. Строки представляются произвольным числом символов, заключённых в кавычки, например, «это строка». В языке Lisp формально нет работы с адресами или указателями, но при необходимости она в системе программирования обеспечивается неявно специальными функциями на уровне прагматики. Строго говоря, атом является неявным уникальным указателем на свой список свойств.

Составные данные языка Lisp на уровне синтаксиса выглядят как списки элементов любой природы, хотя неявно на уровне прагматики языка используются и другие структуры данных, такие как вектора, множества и хэш-таблицы, 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, а вектора

---

6 https://alexott.net/ru/fp/books/ - список источников по функциональному программированию

7 Сообщение И. А. Кирпотиной.

8 А не «значений», как принято говорить.

используются для сопряжения со встроенными машинными процедурами. Семантический базис языка включает в себя функции обработки списков, представленных с их помощью выражений и определения границ вычислимости, а его расширение содержит функции доступа к неявным средствам уровня прагматики языка, включая средства отладки программ и взаимодействия с внешним миром. Списки строятся из пар любых данных функцией CONS[9]. Функция CONS способна строить любые деревья и допускает общие подсписки — слияние ссылок. Пара содержит очередной элемент списка и продолжение списка. Список завершается парой из его последнего элемента и пустого списка «()», являющегося значением атома NIL. Первый элемент списка можно получить функцией CAR[10], остальной список без первого элемента — функцией CDR[11]. Пара из двух атомов «x» и «y» не является списком, имеет вид «(x . y)», её часто называют точечной записью. Имеется предикат сравнения указателей (EQ) и предикат, отличающий атомы от составных форм (ATOM). Составными считаются структуры, созданные функцией CONS, поэтому строки и числа отнесены к атомам. Пустой список также не может быть разделён на части, поэтому и он является атомом, по соглашению его представляют как атом NIL.

Определения и вызовы функций в языке Lisp представляются списками в префиксной форме — первый элемент такого списка является представлением функции, остальные элементы являются представлениями аргументов этой функции. И представление функции, и представления аргументов можно обрабатывать точно так же, как и любые данные. Это позволяет легко писать программы, выполняющие отображения, редукции, свёртки и манипулирующие другими программами (мета-программирование), включая синтаксически управляемые конструкторы обработчиков данных. Обычно выполнение функций начинается с вычисления аргументов, значения которых связываются с параметрами, к значениям аргументов затем применяется функция. В языке Lisp кроме обычных функций существуют специальные — конструкторы или макросы, они отличаются от обычных функций тем, что формы, представляющие аргументы, без вычисления связываются с параметрами функции, применяемой к представлением аргументов. В списке свойств имени обычной функции имеется свойство с индикатором EXPR или SUBR в зависимости от того представлено её определение как список или как исполнимый машинный код. Для специальных функций это соответственно свойства FEXPR или FSUBR [3].

3. Pure Lisp – чисто функциональное программирование

Для быстрого ознакомления с идеями языка Lisp Дж. Маккарти выделил семантический базис — лаконичное подмножество языка [3]. Это диалект Pure Lisp, включающий в себя пять функций обработки списков (CONS, CAR, CDR, EQ, ATOM), четыре специальных функции управления вычислениями — конструкторы (QUOTE, COND, LAMBDA, LABEL), объявляющие константы, ветвления, безымянные и именованные локальные функции, и универсальную функцию EVAL, способную вычислить любое правильно представленное списком выражение, что определяет границы вычислимости. Выполнение функции EVAL было реализовано как интерпретация. Наличия функций QUOTE и EVAL достаточно для поддержки разных схем вычислений, включая ленивые вычисления, оптимизацию программ и любую макротехнику, а также программирование новых схем управления, что образует основу для обучения мета-программированию, включая определение интерпретаторов и компиляторов. Специальная функция QUOTE позволяет заблокировать любое вычисление подобно технике применения комментариев при отладке программ. Конструктор COND представляет общую форму ветвления, работающую и как условный

9 consolidation
10 Content of Address part of Register
11 Content of Decrement part of Register

оператор, и как переключатель с произвольным числом ветвей. Специальная функция LAMBDA создает представление безымянной функции, выделяющее в её определении контекст — область видимости связанных переменных — параметров, которым предстоит получить значение при вызове функции, размещаемую в ассоциативном списке, устроенном подобно стеку. В определении могут встречаться и свободные переменные, наследующие значения с более внешнего уровня, в контексте вызова или на уровне программы, такой механизм наследования существует со времён λ-исчисления А. Чёрча. Динамическая область видимости расширяется при вызове функции и сужается при выходе из функции. Специальная функция LABEL позволяет любой функции дать имя, что удобно[12] для определения рекурсивных функций, имена которых размещаются в той же области видимости, что и переменные.

Обычно при расширении семантического базиса кроме динамической области видимости связанных переменных и функций, поддержана и статическая область видимости глобальных определений, формируемая дополнительной функцией DEFINE[13], позволяющей определять глобальные функции, давать им имена независимо от локальных определений. При совпадении имён определений предпочитается самое новое. Поддержка глобальных определений использует список свойств атома, хранящий системные и программируемые значения свойств с их индикаторами. Для целей отладки используются псевдо-функции Read и Print, обеспечивающие доступ к периферийным устройствам. Для оптимизации программ используется специальная псевдо-функция Compile, выполняющая компиляцию достаточно отлаженных функций и размещающая результат компиляции в списке свойств имена функции с индикатором SUBR, исходное списочное представление сохраняется с индикатором EXPR.

В отличие от большинства ЯВУ, в языке Pure Lisp при необходимости можно создавать свои схемы управления вычислениями. Первоначально основой рабочего цикла была функция EvalQuote, она читала представление функции, затем список константных значений[14] её аргументов, потом применяла функцию к полученным значениям параметров и выводила полученный результат. Позднее стали использовать рабочий цикл работы системы read–eval–print loop **(**REPL**),** просто вычисляющий формы одну за другой, его можно представить выражением вида: (loop (print (**eval** (print (read))))), в котором:

read — специальная псевдо-функция без аргументов, превращающая текст на устройстве ввода в данное языка Lisp, рассматриваемое как результат функции read.

print — тождественная псевдо-функция, выполнение которой сопровождается печатью её аргумента на устройство вывода. Результатом функции print является её аргумент, что удобно для размещения вывода в любой позиции программы при отладке.

loop — бесконечный цикл.

На вход подаётся форма в виде списка, первый элемент которого представляет функцию, а остальные — представления аргументов[15], которые предстоит вычислить до применения к ним обычной функции или можно не вычислять для специальных функций - конструкторов.

Такой системы понятий и средств достаточно, чтобы поддержать определяющие семантические и прагматические принципы функционального программирования:

- универсальность символьных представлений;
- равноправие и независимость параметров функции;
- само-применимость определений (рекурсия);
- гибкость распределения памяти;

---

12 Теоретически можно без именования функции обойтись именами переменных, но получаются громоздкие формулы.

13 В более поздних версиях стали использовать имя Defun для объявления глобальных функций.

14 Не требовалось перед значениями писать «Quote» или ставить апостроф «'».

15 А не «значений» аргументов, для констант нужен апостроф «'»

- неизменяемость хранимых данных;
- единственность результата функции.

Универсальность нацеливает на профилактику необоснованной частичности и повышение темпа отладки, **равноправие** параметров — точка роста для перехода к параллельным вычислениям[16], само-применимость или **рекурсия** даёт свободу лаконичным определениям, **гибкость** распределения памяти освобождает программирование от забот о конечности размеров, неизменяемость данных обеспечивает надёжность и обратимость вычислений, **единственность** результата функции полезна для чёткой комбинаторики взаимосвязанных определений [14-25].

Сборник из 250 упражнений по языку Pure Lisp содержал 15-ти минутные задачи на обработку формул, включая перевод выражений из инфиксной записи в префиксную и обратно, программирование отображений, символьное дифференцирование и интегрирование, что показывало приспособленность языка к формированию проблемно-ориентированных диалектов для применения в разных сферах компьютерной обработки данных.

4. Эволюция языка

Джон Маккарти ожидал, что оставшиеся проблемы организации вычислений будут решены в более поздней версии, условно названной Lisp 2, в которой планировал включить в язык обработку многомерных векторов, сопоставление с образцами и организацию параллельных вычислений [8]. Рассказывая о языке Lisp, Джон 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 [3]. В опубликованном определении интерпретатора было решение FunArg-проблемы, называемое теперь как «замыкание» функции. Описание языка стало основой для создания Lisp-систем на многих других компьютерах как в США, так и за её пределами, в нашей стране на БЭСМ-6, ЕС ЭВМ, СМ-4 и других машинах[17] [26-37]. В начале 1960-ых на MacLisp был реализован высокоэффективный компилятор. Компилировать рекомендовалось отдельные функции, достаточно отлаженные для компиляции.

В 1964 году П. Лэндин (Peter J. Landin) предложил машину SECD – это виртуальная и/или абстрактная машина, предназначенная для использования в качестве целевого языка (бэкэнд) при компиляции языков функционального программирования[18]. Эта машина выдерживала принцип неизменяемости данных за исключением одной позиции — для организации рекурсивных функций предложено заранее объявлять изменяемым регистр для указателя на очередное поколение значений переменных. Вскоре появился Lispkit[19] — реализация чисто функционального диалекта Pure Lisp с лексической областью видимости, разработанная в качестве испытательного и учебного стенда при изучении концепций функционального программирования, включая ранние эксперименты с ленивыми вычислениями. Полученные компилятор и виртуальная машина были легко переносимы.

В 1966 году на диалекте Portable Standard Lisp (PSL) для PDP-10 был реализован транслятор SYSLisp и написанный на том же SYSLisp кросс-компилятор, с помощью которого ядро PSL можно было перенести на любую аппаратуру, что является одним из первых примеров раскрутки компиляторов при переносе программ на новую архитектуру,

16 Близко по времени появляется APL – первый язык параллельного программирвоания.

17 https://www.computer-museum.ru/histsoft/lisp_sorucom_2011.htm

18 Хендерсон П. Функциональное программирование. Применение и реализация = Functional Programming. — М.: Мир, 1983. — 349 с.
Landin, P. J. (January 1964). "The Mechanical Evaluation of Expressions". Comput. J. **6** (4): 308—320. doi:10.1093/comjnl/6.4.308

19 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

когда для переноса системы ядро изначально пишется на машинно-независимом промежуточном языке, для которого, в свою очередь, создаются реализации на всех целевых платформах. На базе PSL в 1968 году была создана система компьютерной алгебры Reduce, используемая физиками до сих пор. На диалекте MacLisp разработаны система компьютерной алгебры Macsyma и семейство расширяемых текстовых редакторов Emacs. Джозеф Вейценбаум опубликовал программу-собеседник Элиза, выдержавшую тест Тьюринга, что показало зависимость теста от человеческого восприятия[20].

В 1974 году в Xerox началась разработка аппаратуры и системы машинных команд для аппаратной реализации языка Lisp. На основе диалекта Interlisp был впервые создан многооконный графический интерфейс пользователя, использована графика с высокой разрешающей способностью и применён манипулятор «мышь». Interlisp базировался на динамическом связывании, тогда как многие новые версии языка — статические. Язык Scheme был разработан в 1976 году в MIT в рамках проекта по созданию Lisp-машин [38-39]. В США такие разработки велись в 1970-х годах в исследовательском центре Palo Alto. Для своего времени Lisp-машины были одними из мощнейших ЭВМ в классе персональных рабочих станций (около 7 тыс. штук во всём мире на 1988 год). Это явилось доказательством, что так называемая «неэффективность языка Lisp» обусловлена не свойствами языка, а особенностями компьютеров и методов реализации языков программирования. При разработке диалекта Scheme[21] произошёл переход от программируемой компиляции отлаженных функций к принудительной компиляции программы при её чтении, ради эффективности произошло совмещение интерпретации с компиляцией, что потребовало унификации их семантики. Ранее функциональная эквивалентность результатов интерпретации и компиляции рассматривалась как подтверждение правильности программ.

При распространении микропроцессоров появился диалект muLISP, интерпретатор которого обладал предельной гибкостью и компактностью[22]. Это позволило на малых объёмах памяти реализовать серию средств компьютерной алгебры (muSIMP, muMATH и DERIVE). Версия этого диалекта распространялась как Microsoft LISP. Пришло признание, что на языке Lisp выполнена разработка наиболее сложных проектов в области системного программирования, исследования методов оптимизации и преобразования программ.

В 1984 году появился диалект Common Lisp — мультипарадигмальный язык общего назначения, дополняющий традиционные динамические решения языка Lisp механизмами статического связывания переменных и раздельных областей видимости, программирования макросов, функционалов, пакетов и лексических замыканий функций [40-47]. В 1995 году Common Lisp был стандартизован ANSI. Этот диалект резко расширил сферу производственного применения, используя средства работы с матрицами, хэш-таблицами, программируемыми структурами данных, подобными структурам в языке C, и мультизначениями, удобными для моделирования потоков. Всё это представляется в виде списков, но на уровне прагматики реализуется как эффективные структуры данных. Семантика Common Lisp поддерживает формирование проблемно ориентированных пакетов, версий и диалектов.

В 1986 году появился диалект AutoLISP, на котором была создана система инженерного проектирования AutoCAD, используемая до сих пор[23].[48-49] Вскоре

---

20 Вейценбаум Дж. Возможности вычислительных машин и человеческий разум. От суждений к вычислениям. Пер. с англ. М., 1982.

21 https://habr.com/ru/companies/tbank/articles/267015/

22 Программирование на языке ЛИСП в системе muLISP-90 .. Байдун В.В., Кружилов С.И., Сергиевский А.Е, Чернов П.Л. - М.: Моск. энерг. ин-т, 1993. ─ 40 с. https://studfile.net/preview/1542615/ https://al.cs.msu.ru/system/files/PosobieLisp.pdf, http://www.machinelearning.ru/wiki/images/b/bf/Lect_15_fp_mdv.pdf

23 http://www.caddsoftsolutions.com/AutoLISP.htm

появился диалект ISLISP — спецификация этого языка разработана в 1990-х годах, опубликована ISO в 1997 году и обновлена в 2007 году. Много внимания уделено взаимодействию с кодом на языках C/C++ и Java и сравнению с ними [50]. Система продолжает развиваться и поддерживаться, существуют версии для большинства доступных ОС и аппаратных платформ. У нас в ИСИ СО РАН была выполнена реализация аналога этой системы bCad[24], она применялась в НЭТИ и архитектурной академии.

Диалект XMLisp — расширение Common Lisp, которое использует протокол мета-объектов для интеграции символьных выражений с расширяемым языком разметки (XML). Ранее Lisp был применён для разработки предшественников HTML, некоторые специалисты считают, что таким предшественником являются S-выражения, точное списочное представление функций, послужившее примером для TEX, LATEX и других языков разметки текстов.

Сторонники чисто функционального программирования избегают деструктивных функций, подчёркивая опасность изменения состояний памяти. Тем не менее, для большинства версий языка Lisp деструктивные функции являются обычным явлением (rplaca, rplacd), только они на периферии внимания. Применение деструктивных функций может приводить к циклическим структурам, влекущим проблемы при отладке и применении программ. За редким исключением деструктивные функции в языке Lisp являются эффективными аналогами доступных обычных функций [41].

В середине 1980-ых семейство Lisp разделилось на две фракции (Lisp-1 и Lisp-2)[25] в зависимости от использования динамического (периода выполнения) или статического (лексического, периода компиляции) связывания переменных. **Lisp-2** — Lisp 1.5, Common Lisp, Emacs, Ruby, Perl. **Lisp-1** — Pure Lisp, Scheme, Clojure, JavaScript, Python[26]. Эта разница объясняется приоритетами в областях применения языков [51]. Различаются отношение к булеву типу данных, кодированию пустого списка, возможностям программируемых списков свойств атомов и выбору между динамической компиляцией функций или принудительной компиляцией программ.

На сайте Wikipedia можно ознакомиться с временной шкалой диалектов языка Lisp и сфер его применения (**Lisp 1.5,** Maclisp, Interlisp, ZetaLisp, **Scheme**, NIL, **Common Lisp**, LeLisp, T, Emacs Lisp, Emacs Lisp, OpenLisp, PicoLisp, EuLisp, ISLISP, **Racket**, Guile, **Clojure**, Arc, LFE, Hy, Chialisp)[27]. Со времени своего появления язык Lisp оказал влияние на большое число языков программирования, причём не только функциональных[28], включая JavaScript, CLU, COWSEL, C#, Dylan, Elixir, Excel, Forth, Haskell, Io, Ioke, Java, Julia, Logo, Lua, ML, Nim, Nu, OPS5, Perl, POP-2/11, Python, R, Rebol, Red, Ruby, Scala, Swift, Smalltalk, Tcl, Wolfram и другие.

Ряд операционных систем были написаны на диалектах Lisp Machine Lisp, Interlisp (Xerox), а позже частично и на Common Lisp. Многие операционные системы используют унаследованные от языка Lisp понятия функции, соглашения, методы, структуры и т. д. или написаны на языке Lisp (Open Genera, Medley на Interlisp, ChrysaLisp и др.). Lisp — один из немногих языков программирования, прошедших процесс стандартизации для

---

24 *Владимир Малюх.* Система bCAD для проектных и дизайнерских работ // PCWEEK/Russian Edition. — №14 (138)1998

25 Вопрос является ли отдельное пространство имен для функций преимуществом стало источником разногласий в сообществе Lisp. Обычно это называют дебатами о Lisp-1 и Lisp-2. Lisp-1 относится к модели Scheme, а Lisp-2 относится к модели Common Lisp. Эти названия были придуманы в статье 1988 года Ричарда П. Габриэля и Кента Питмана, в которой два подхода подробно сравниваются. http://www.nhplace.com/kent/Papers/Technical-Issues.html

26 Язык Python изначально был объявлен как функциональный, позднее стал мультипарадигмальным с доминированием ООП.

27 Отсутствуют muLISP и Logo.

28 https://habr.com/ru/articles/428229/ - Как Lisp стал языком программирования для Бога

использования в промышленности в виде диалекта Common Lisp, стандартизованного ANSI, реализации которого существуют для большинства платформ, имеет официально стандартизированные диалекты: схема R6RS, схема R7RS, схема IEEE, ANSI Common Lisp и ISO ISLISP.

В нашем столетии появился современный отечественный диалект HomeLisp, достаточный для показа и изучения концепций функционального программирования. На сайте http://homelisp.ru/ имеется подробная инструкция по работе с языком Lisp и системой, включая средства машинной графики, работы под Windows и в Интернете [52].

Во многих источниках утверждается, что язык Lisp не поддерживает механизм наследования[29], необходимый для парадигмы ООП. Следует отметить, что начиная с λ-исчисления А.Чёрча через свободные переменные выполняется наследование значений из внешнего контекста. Для ООП механизм наследования может быть смоделирован иерархией функций или списков с учётом того, что инкапсуляция и полиморфизм поддержаны общим механизмом работы с атомами. В диалекте Clisp реализована полная поддержка ООП пакетом CLOS. Пример создания собственной модели ООП в языке Lisp опубликован в книге Пола Грэма (*Paul Graham*) «ANSI Common Lisp» на базе иерархии хэш-таблиц[30]. В октябре 2019 года Пол Грэм опубликовал спецификацию языка Bel[31], «нового диалекта Lisp», ранее, с 2001 года Грэм работал над новым диалектом языка Lisp под названием Arc[32], выпущенном в 2008 году. Пол Грэм является автором книг «On Lisp» (1993, «ANSI Common Lisp» (1995) и «Hackers & Painters» (2004) [52-56].

Если изначально язык Lisp был связан с символьной обработкой данных и процессами принятия решений при исследовании задач искусственного интеллекта, то вскоре он нашёл применение в математике, в естественных и гуманитарных науках, в промышленности, образовании и медицине, от декодирования генома человека до систем инженерного проектирования и языков преобразования и верификации программ (Applicative Common Lisp[33], Jitawa Lisp[34]), включая использование в качестве скриптового языка для разработки прикладных программ (AutoLISP, Emacs Lisp, Interleaf Lisp, Nyquist, Rep, SKILL, TinyScheme, ZWCAD[35]) и применения операционных систем. Возникли специализированные диалекты языка Lisp (Game Oriented Assembly Lisp — GOAL для высоко-динамичных трёхмерных игр, таких как Jak и Daxter; ICAD — система «знаний на основе знаний»). На базе языка Lisp было выполнено много первых экспериментов по системному программированию, особенно по оптимизирующим преобразованиям и верификации программ. Язык Logo — его часто называют «Lisp без скобок», используемый для детского программирования, послужил основой проектов ЮНЕСКО по созданию Logo-миров, поспособствовавших подъёму экономики в странах так называемых «восточно-азиатских драконов» (Гонконг, Сингапур, Южная Корея и Тайвань) через освоение и применение ИТ [57]. Образовательное значение языка Scheme отмечали разработчики языка Java, вспоминавшие в первых публикациях комфорт программирования на языке Scheme при учёбе в MIT, что влияло на их решения по Java.

5. Обзор особенностей

Общие концепции языка Lisp можно представить следующими принципами:

---

29 Точнее, наследование поддерживается через свободные переменные, наследующие значения из более внешних фукнций.

30 http://www.paulgraham.com/avg.html, http://www.paulgraham.com/rootsoflisp.html

31 https://habr.com/ru/articles/484328/ - статья о появлении языка Bel.

32 http://www.arclanguage.org/tut.txt — про язык Arc

33 https://en.wikipedia.org/wiki/ACL2 — диалект для верификации

34 https://www.cl.cam.ac.uk/~mom22/jitawa/ - верифицирующий Lisp

35 https://sapr-soft.ru/stati/lisp-debugger - Отладчик Lisp, разработанный на базе Visual Studio Code от Microsoft

1) Любую информацию для компьютерной обработки можно представить в **символьной форме**. Числа — не более чем частный случай символьного представления информации.

2) Данные бывают неделимыми, в языке Lisp не предусмотрен их разбор на части, или составными структурами, допускающими сборку из частей и разделение на части. Неделимые выглядят как идентификаторы или числа/строки. Символьные формы для представления неделимых данных называют атомами. Составные структуры строятся из пар элементов произвольной природы, разделённых точкой и заключенных в скобки.

(Left **.** Right)

Круглые скобки в языке Lisp возникли по простой причине - на многих терминалах не было других скобок, зато и теперь они набираются на любой раскладке (рус/eng).

Любую сущность, включая ключевые слова операторов управления вычислениями, можно представить как атом, именовать в соответствии с лексиконом области приложения. Атомы, смысл которых может отличаться от их имени, сопровождаются списком свойств, часть свойств определены в системе программирования, другие могут программироваться. Разница между переменными и разными категориями функций представляется в списке свойств атомов. Атом, обладающий непустым списком свойств, называется символом. Список свойств атома можно рассматривать как реализацию размеченного множества.

Основная структура составных данных — список, элементы которого заключёны в скобки и при необходимости разделяются пробелами. Нет загромождения запятыми или другими разделителями. Элементы могут быть любой природы и разного типа. Список строится из пар и завершается парой, содержащей последний элемент и пустой список «( )».

(Left **.** ( )) = (Left)

(Elem1 Elem2 (Elem3-1 …) …)

Пустой список не делится на части средствами языка, поэтому он является одновременно списком и атомом. Его можно реализовать отдельным типом данных, он по соглашению изображается как атом «NIL» и обычно кодируется адресом 0, что удобно для его использования в качестве значения «ложь».

(Left **.** Nil) = (Left)

3) Любое понятие программирования можно рассматривать как функцию или применение функции. Применение функции выглядит как список из представления функции и представлений её аргументов.

(Function Argument1 …)

Кроме обычной схемы вычислений с предварительным вычислением аргументов, поддержана возможность производить вычисления значений параметров внутри запрограммированной функции — специальные функции или конструкторы. Это позволяет программировать разные схемы управления вычислениями и конструировать представления выражений, других типов данных и разных категорий функций, использовать конструкторы для разных форм вычислений, а также локальных и глобальных определений функций.

Язык Lisp использует **два пространства имён** — отдельно для локального (динамического) или глобального (статического) связывания имён со значениями переменных или определениями функций. Пространство имён для локального связывания устроено как ассоциативный список, содержащий пары из имён и их значений или определений. Он функционирует подобно стеку параметров во многих языках программирования — наращивается при вызове функции и восстанавливается при возврате из функции. Для глобального связывания используется хэш-таблица,

отображающая атомы в и их списки свойств, порядок определения которых не имеет значения, лишь бы определение существовало в момент вызова функции.

Безымянные локальные функции конструируются специальной функцией LAMBDA, первый аргумент которой представляет список имён параметров, а второй выражение, вычисляющее результат функции над этими параметрами. Имена параметров — это связанные переменные, определяющее выражение может содержать и другие имена — свободные переменные.

(Lambda (Par1 …) Expr)

Именование локальной функции выполняет специальная функция LABEL.

(Label Name Definition)

Глобальные функции именует специальная функция DEFUN[36], для которой вслед за именем определяемой функции представляется список её параметров и выражение, вычисляющее результат этой функции. Именование позволяет удобно определять рекурсивные функции[37].

(Defun Name (Par1 …) Expr)

При конструировании форм и определений функций различаются связанные и свободные переменные в зависимости от синтаксической позиции. Если одна функция вызывает другую, то свободные переменные внутренней функции могут использовать значения одноимённых переменных из вызвавшей, внешней функции. Функции LAMBDA и DEFUN позволяют совмещать два пространства имён — статический контекст определения и динамический контекст вызова функции.

((Lambda (x) (Car x)) '(s d f)) ; = S '(s d f) = (Quote (s d f))[38]
((Defun Fun (x) (Car x)) '(s d f)) ; = S
(Fun '(s d f)) ; = S

Для преодоления возможно разных областей видимости переменных (контекст) при вычислении функциональных параметров интерпретатор для работы с ними строит специальный рецепт из определения функции и контекста в точке её вызова, роль которого подобна замыканию, формируемому в точке определения функции в языках семейства Lisp 1.

Некоторые виды функций могут быть определены для работы с произвольным числом параметров — мультифункции.

Элементарные базовые функции строят структуры данных, разбирают их на части, проверяют совпадение атомов и атомарность форм (Табл.1).

Таблица 1

Базовые функции языка Lisp/

Функция	*Параметры*	*Результат*	*Примечание*
Cons	A B	(A . B)	Создание пары из двух атомов.
Cons	A Nil	(A)	Если правый параметр Nil, то получается список.
Cons	A (B)	(A B)	Если справа список, то он удлиняется.
Car	(A . B)	A	Левый элемент пары.
Cdr	(A . B)	B	Правый элемент пары.
Eq	A A	T[39]	Атомы совпадают
Eq	A B	Nil	Атомы различны
Atom	A	T	Атомарность подтверждена
Atom	(A . B)	Nil	Параметр не атом.

36 Define в ранних версиях.

37 Можно глобальные определения рассматривать как самые внешние, а кроме того, можно вообще обойтись без именования функций, но это на практике слишком громоздко.

38 Сокращённое обозначение.

39 Атом «T» обычно представляет значение «истина», но можно использовать любое значение, кроме Nil.

Базовые функции над конкретными типами значений группируются в комплекты, обладающие полнотой операций и/или отношений, имеются обратные функции и минимальные значения, выполняющие роль нулей или единиц. Например, при любых X и Y и составном Z выполняются следующие отношения:

(Car (Cons x y)) = x
(Cdr (Cons x y)) = y
(Cons (Car z) (Cdr z) = z
(Atom (Cons x y)) = Nil

Можно сказать, что при выполнении таких операций поддерживаются аксиомы[40], позволяющие в отдельных случаях получать результаты функций без выполнения вычислений.

4) Любую форму можно заблокировать от вычисления конструктором констант QUOTE.

(Quote A) *; = A[41] константа*
(Quote (Atom (Quote A))) *; = (Atom (Quote A)) константа*
'A *; = (Quote A) - A[42] константа*
'(Cons x y) *; = (Cons x y) константа*

Любую представленную или построенную в программе форму можно вычислять по мере необходимости, например, вычислять ранее заблокированные формы, **универсальной функцией EVAL**.

(Eval '(Atom 'A)) ; = *T - результат вычисления (Atom 'A)*
(Eval '(list 'car ' '(A B C))) ; = *(CAR (QUOTE (A B C))) - результат создания формы*
(Eval (Eval '(list 'car ' '(A B C)))) ; = *A - вычисление сконструированной формы*

5) Для каждого типа данных имеется предикат, позволяющий в любой момент выполнения программы выяснять принадлежность данного нужному типу.

Любая функция может выполнять роль предиката.

Процесс вычислений может обладать произвольным числом ветвей, выбор которых задан конкретными предикатами с помощью конструктора COND. Модель обычного «If Predicate Then **True** Else **False»** можно представить в виде:

(Cond (Predicate **True**)
(**T** **False**))

Хотя часто поддержан и конструктор IF, выглядящий проще:

(If Predicate **True False)**

Поиск в списках при неудаче часто завершается на пустом списке. Поскольку эффективность анализа данных зависит от скорости обнаружения неудачи, нет смысла тратить время на сравнение результата с пустым списком для получения логического значения. Эффективнее пустой список рассматривать как значение «ложь», подобно коду «0» во многих языках программирования. Роль пустого списка для составных значений аналогична роли нуля для чисел. Пример — функция, каждое число из списка увеличивающая на единицу.

```
(defun  next (xl)    ; Следующие числа
    (cond          ; достаточно одной ветви, иначе ().
          (xl (cons (1+  (car  xl))   ; пока список не пуст прибавляем 1 к его голове
                    (next (cdr  xl))
                   ; переход к остальным, собирая результаты в список
```

40 Подобно абстрактным типам данных

41 Символ «;» означает комментарий до конца строки.

42 Символ «'» - сокращенное обозначение конструктора Quote.

```
 ) )    ) )
(next '(1  2  5))    ; = (2  3  6)
```

6) Для сфер приложения, обладающих исторически сложившимися представлениями данных, язык расширен введением так называемых псевдо-атомов, смысл которых виден по форме их записи, - это числа и строки. Они не имеют списков свойств. Числа бывают целыми произвольной длины, дробными (2/5) и вещественными. Строки могут разделяться на символы специальными функциями, но по соглашению они считаются псевдо-атомами, т. к. они не созданы функцией CONS.

7) Ввод-вывод организован как псевдо-функции, отличающиеся тем, что кроме получения результата они выполняют некоторый побочный эффект. Вывод формально является тождественной функцией, но одновременно он печатает свой аргумент на внешнем носителе, фактически являющемся неявным объектом. Ввод принимает данное с внешнего устройства, строит представленную этим данным символьную форму, становящуюся результатом ввода, что удобно при переходе от обработки констант к обработке произвольных выражений.

8) В системе имеется функция ERROR для обработки ошибок, сообщающая диагноз при невозможности вычислить результат функции и позволяющая программировать продолжение работы. Имеются средства управления отладочным выводом аргументов и результатов функций. Слежение за ходом вычислений обеспечено функциями TRACE/UNTRACE. Атом OBLIST хранит список всех доступных атомов. Функция GENSYM конструирует атомы, обладающие уникальными именами.

Программа имеет доступ ко всем функциям системы программирования (интерпретатора, компилятора и отладчика) и может им давать свои определения. Всё, что понадобилось системе, может быть полезно и программисту. Изменять можно определения любых атомов, кроме Nil.

9) Предпочтение интересов малых программ и широкой аудитории потребовало автоматизации повторного использования памяти, чтобы освободить специалиста для решения своих, более существенных, задач. Автоматизация «сборки мусора» (*garbage collection*) допускает вызов (GC) из программы.

10) Поддержаны **альтернативные решения** организации вычислений. Реализация вычислений с помощью обычных функций, выполняемых после вычисления аргументов, дополнена специальными функциями, вместо значений аргументов, обрабатывающих представления аргументов. Кроме организации правильных вычислений поддерживается представление обработки ошибочных ситуаций и продолжения вычислений. Пространство атомов расширено конкретными типами данных, принятыми в основных областях приложения — псевдо-атомы. Сосуществуют интерпретация и компиляция функций. Результат компиляции размещается в списке свойств имени функции без удаления символьного определения. Некоторая разница в семантике интерпретации и компиляции полезна для проверки правильности программ. Интерпретатор и компилятор для языка Lisp первоначально были определены на языке Lisp, а затем были воспроизведены в машинном коде.

Для более детального сравнения языка Lisp с его диалектами в таблице 2 перечислены особенности языка Lisp, заодно показано отсутствие случайных решений. Важно обратить внимание, что с языком Lisp связан конкретный свод семантических и прагматических принципов. Определение языка структурировано на базис, консервативное расширение, функции диагностики и отладки программ, средства связи с внешним миром, оно опирается на ряд неявных конструкций уровня прагматики, необходимых для реализации языка.

Таблица 2
Сводка особенностей Lisp

Характеристика языка	*Обоснования и решения*
	Принципы и выбор решений
Универсальность	Любую информацию для компьютерной обработки можно представить в символьной форме.
( )	На терминале не было других скобок. Зато и теперь они набираются при любой раскладке (рус/eng), в отличие от фигурных и квадратных
Виды символьных форм	Формы бывают неделимыми, если в языке не предусмотрен разбор их на части, или составными, если язык допускает их сборку из частей и разделение на части операциями языка.
Атомы	Символьные формы для представления неделимых данных называются атомами. Любую сущность, включая ключевые слова, можно представить как атом, именовать в соответствии с особенностями области приложения. Атомы выполняют роль идентификаторов для значений любой природы.
Константы и само-определимые псевдо-атомы	Введены так называемые псевдо-атомы, смысл которых виден по форме их записи, - это числа и строки. Строки могут разделяться на символы специальными функциями, а не операциями языка, по соглашению они считаются псевдо-атомами.
Гибкость границ памяти и размеры данных	Имена атомов, числа, стоки, списки не имеют ограничений на длину, их размеры не зависят от формата машинных слов и границ блоков памяти..
Списки, иерархия форм	Составные формы строятся из пар элементов произвольной природы, разделённых точкой и заключенных в скобки. Основная структура составных данных — список, элементы которого заключёны в скобки и при необходимости разделяются пробелами. Нет загромождения запятыми или другими разделителями. Список строится из пар и завершается парой, содержащей последний элемент и пустой список. Элементы списка могут быть любой природы и разного вида.
Пустой список **Nil = ()**	Пустой список не может быть разделён на части операциями языка, поэтому он является одновременно списком и атомом, его можно считать отдельным типом значений.
Неизменяемость данных	Обработка списков подчинена принципу неизменяемости данных. Преобразованные списки строятся в новой памяти без искажения исходных данных.
Списки свойств атома	Атомы, смысл которых не ограничен их именем, сопровождаются списком свойств, часть которых определены в системе программирования, другие могут программироваться. Разница между переменными и разными категориями функций представляется в списке свойств атомов, что соответствует метаданным в современных ИТ. Обработка списков свойств подчинена принципу неизменяемости данных, при пополнении ранее существовавшие свойства не исчезают, по умолчанию используется самое новое определение. Возможно многократной вхождение одного и того же свойства, что соответствует понятию «размеченное множество» и поддерживает полиморфизм.
Гомоиконность	Представление программы в виде списков позволяет чётко видеть структуры данных, представляющие программу. Такое преставление фактически является абстрактным синтаксическим деревом (AST).
Функции. **Рекурсия**	Любое понятие программирования можно рассматривать как функцию или как применение функции. Определения функций подчинены принципам **равноправия и независимости** аргументов и **единственности** результата, поддержана рекурсия.
Конструкторы, **макросы**	Можно без предварительного вычисления аргументов функции производить их вычисления — специальные функции. Это позволяет программировать разные схемы управления вычислениями, конструировать представления выражений и разных категорий функций.
Формы, выражения	Программы и выражения представляются как символьные формы — это список, первый элемент которого содержит представление функции, а остальные — представления аргументов. Пустой список считаются константной формой, всегда имеющей значение Nil.

	Именованные формы допускают само-применимость определений.
Параметры	Обычные функции работают над позиционным списком различимых параметров. Специальные функции могут в качестве параметра получать ассоциативный список, аргументы могут быть серийными (prog) или представлены общим списком (apply).
Пространства имён	Поддержано два пространства имён — раздельно для локальных и глобальных определений с приоритетом локальных. Имена переменных и функций расположены в общем пространстве.
Алгебраические системы функций	Базовые функции над конкретными типами значений группируются в комплекты, обладающие полнотой операций и/или отношений, определены обратные функции и минимальные значения, выполняющие роль нулей или единиц.
Аксиоматика отношений	Поддерживаются аксиомы, позволяющие в отдельных случаях получать результаты функций без выполнения вычислений.
	Базис
Quote	Любую символьную форму можно заблокировать от вычисления, сделав из неё константу.
Eval	Для любой представленной или построенной в программе символьной формы можно вычислять её значение по мере необходимости, в том числе, вычислять ранее заблокированные или сконструированные формы.
Связанные и свободные переменные	При объявлении **безымянных функций** Lambda -конструктор выделяет список **связанных** переменных, они получат значение при вызове функции. Определяющая форма функции может содержать и другие переменные, называемые **свободными**, они получают значение из внешнего контекста или замыкания.
Контекст — области видимости, пространства имён, динамика и статика	Если одна функция вызывает другую, то внутренняя через свободные переменные может использовать значения переменных из объемлющих функций. Возможно размещение определения функции внутри другой, более внешней функции, представляющей контекст для внутренних функций.
Label приоритет локальным определениям. Define, **Defun**	Label освобождает **именование** локальных функций от придумывания уникальных имён. Теоретически глобальные определения можно реализовывать как самые внешние локальные. Кроме создания локальных безымянных и именованных функций имеется практичный компромисс в виде глобальных функций (Define, Defun) с уникальными именами.
Мульти-функции	Некоторые виды функций могут быть определены для работы с произвольным числом параметров.
Полиморфизм	Атом может одновременно обладать разными свойствами, возможно неоднократное вхождение любого свойства. Атом может представлять и значение, и несколько разных категорий определения функции, и ещё что-нибудь по усмотрению программиста. Функция может иметь ряд различных определений, включая одновременное вхождение символьных и кодовых определений, например, результатов компиляции. По умолчанию работает самое новое определение, но и к прежним доступ возможен.
Функциональные переменные	Представления функций — не более чем символьная форма, поэтому нет никаких препятствий существованию функциональных переменных, значением имеющих представление функции, и передаче представлений функций в качестве параметров при вызове функций или выдаче их как результат.
Функции высших порядков	Использование функциональных переменных означает, что можно определять функции высших порядков, аргументами и/или результатами которых могут быть представления функций.
Отображения, свёртки и фильтры	Применение функций, передаваемых через параметры, требует согласования не только типов данных или значений, но и рангов функций и контекста их вызова.
Безымянные определения	Передаваемые через параметры функции часто имеют разовый характер, именование функций осмысленно для многократного использования. Безымянные функции строит конструктор Lambda.
Cond	В Pure Lisp хотя бы одна ветвь должна быть выбрана. В Lisp 1.5 в форме Prog это не обязательно, при отсутствии истинного предиката значением ветвления является Nil.
Eq – сравнение указателей или атомов	Существуют полный комплект предикатов для контроля процесса выполнения программы.
Контроль типов	Каждый тип данных имеет предикат, позволяющий в любой момент вычисления

данных: **Atom**, Null, Number	программы, выяснять принадлежность значений нужному типу данных. Для компиляции программ следует заранее объявлять тип данных, передаваемых через свободные переменные, т. к. их значения могут быть не известны заранее.
Предикаты	Любая функция может выполнять роль предиката.
( ) логика **Nil**, значение «ложь»	Nil выполняет роль значения «ложь», любое другое значение работает как «истина». Поиск в списках при неудаче завершается на пустом списке. Поэтому эффективно пустой список рассматривать как значение «ложь», подобно коду «0» во многих языках.
Компиляция	Компилируются достаточно **отлаженные** функции для профилактики накладных расходов при ограниченных ресурсах памяти и времени.
	Диагностика и отладка
Error – диагностика ошибок	Функция Error - обработчик ошибок, сообщающий диагноз дефекта в определении выражений и позволяющий программировать продолжение вычислений.
Trace/Untrace – **отладка**	Поддержана возможность смотреть аргументы-результаты отлаживаемых функций по ходу их вычисления, список которых может быть при необходимости изменён.
	Расширения и внешний мир
Открытость	Программа имеет доступ ко всем функциям системы программирования (интерпретатора, компилятора, отладчика) и может им давать свои определения. Всё, что понадобилось системе программирования, может быть полезным и программисту. Изменять можно определения любых атомов, кроме Nil.
Print, Read – ввод-вывод, **I/O**	Print формально **является тождественной функцией** вывода данных, он печатает свой аргумент/результат на внешнем носителе. Ввод принимает данное с внешнего устройства, строит эквивалентную ему символьную форму, становящуюся его результатом, что удобно при переходе от обработки констант к обработке произвольных данных из внешнего мира.
	Прагматика, неявные структуры данных и функции
GC – «Сборка мусора», **гибкость** распределения памяти	**Автоматизировано** повторное использование памяти, чтобы **освободить** специалиста для более существенных задач. Возможен вызов GC из программы.
Устройства **ввода-вывода** данных и ОС	Имеются специальные функции, **неявно** поддерживающие доступ к периферийным устройствам и операционной системе.
Замыкание функции	Для передачи функций со свободными переменными в качестве параметров интерпретатор **неявно** выполняет формирование рецептов, однозначно связывающих переменные в точке вызова функции с её контекстом, а не в точках её вычисления, в которых пространства имён могут различаться.
Структуры данных	Хотя на уровне языка Lisp нет ни векторов, ни множеств, ни хэш-таблиц, в реализации **неявно** поддержана работа с векторами для взаимодействия с машинным кодом процедур, со множествами для реализации списков уникальных параметров функции и с хэш-таблицами для идентификации атомов. В языке поддержаны две функциональные модели хэш-таблиц — глобальный список свойств атома и ассоциативный список, локально связывающий атомы с их определениями. Обе модели не нарушают принцип неизменяемости данных.
Присваивания и адреса	Присваивания поддержаны в двух формах. Это псевдо-функции Rplaca и Rplacd, явно изменяющие значение по неявно указанному адресу, и кроме того функции Set и Setq внутри формы Prog, выполняющие изменение значений рабочих переменных без побочного эффекта на внешнем уровне. Это позволяет функцию Prog рассматривать как чистую функцию, совмещающую декларативный ответ на вопрос «Что?» с императивным ответом на вопрос «Как?».
Раскрутка системы программирования: интерпретатора и компилятора	Интерпретатор и компилятор для языка Lisp первоначально были определены на языке Lisp, а затем воспроизведены в машинном коде.
Альтернативность решений, **полное пространство для мысли**	Реализация вычислений с помощью обычных функций, выполняемых после вычисления параметров, дополнена специальными функциями, вместо значений параметров, обрабатывающих их представления. Кроме организации правильных вычислений поддерживается представление обработки ошибочных ситуаций.

	Пространство абстрактных атомов расширено конкретными типами данных, принятыми в основных областях приложения — псевдо-атомы. Автоматизация «сборки мусора» допускает вызов GC из программы. **«Если нельзя, но очень хочется, то можно»**.
Изобразительные средства **( ) . ' ;**	**Бедный синтаксис** освобождает от необходимости размышлять на тему, «где поставить запятую, где не ставить запятой». **( )** - границы точечной пары или списка, **.** - разделитель элементов пары, из которых строятся все структуры данных, **'** - кавычка — префикс, объявляющий представление константы, 'x эквивалент (Quote x). **;** - объявление начала строчного комментария.

Содержание таблицы 2 показывает, что особенности языка Lisp не сводятся к комплекту доступных средств уровня семантики языка программирования, их дополняют реализационные средства и структуры данных уровня прагматики системы программирования. Характерные структуры данных и методы их обработки в языке Lisp подчинены определённым требованиям к их реализации и аксиомам, близкие идеи **абстрактных типов данных** сформулированы в 1974 году Бараброй Лисков (Barbara Liskov).

6. Диалект Scheme для аппаратной реализации

Scheme был разработан в 1976 году в MIT в рамках проекта по созданию Lisp-машин [38, 39]. В США такие разработки велись в 1970-х годах в исследовательском центре Palo Alto. Приаппаратная направленность произвела сдвиг прагматических решений в направлении императивно-операторного программирования, повышения эффективности обработки данных и учёта удобочитаемости программ при учебном программировании, иногда в ущерб продуктивности производственного программирования. Внешне это не очень заметно, но поведение программ может отличаться от ожиданий программиста, работавшего с языком Lisp. Разницу обычно можно компенсировать функциональным моделированием.

6.1. Наследие и отличия от языка Lisp

Язык Scheme[43] наследует от языка Lisp основные типы неделимых данных (числа и строки), списки, а также, общую схему представления выражений и функционирования программ, включая «сборку мусора». Главным изменением, обоснованным доводами эффективности программ на языке компьютера, является совмещение чтения программ с их принудительной компиляцией, что повлекло не только унификацию семантики, но и определённое смещение раскладки понятий, включая изменение грани между примитивными и составными типами данных. Отсутствуют атомы, сопровождаемые программируемыми списками свойств, что снижает продуктивность программирования решений задач, ориентированных на представление знаний. При необходимости можно запрограммировать списочную модель программируемых списков свойств. В целом, язык Scheme дал ответ на вопрос «**КАК** сделать функциональное программирование столь же эффективным как императивное?». Ответ заключается в следующем:

- Немного ограничить универсальность символьных вычислений отказом от абстрактных атомов и программируемых списков свойств символа, оставить только системные свойства.

- За основу базовых структур данных взять вектора, а списки использовать при необходимости как синтаксическое расширение в отдельной библиотеке.

- Чтение списочного представления программы, фактически являющегося представлением её абстрактного синтаксического дерева, совместить с принудительной компиляцией, а интерпретатор eval вынести из базиса во вспомогательную библиотеку.

43 https://www.scheme.com/tspl4/ - R. Kent Dybvig Описание Scheme.

- Макротехнику ограничить выполнением на этапе компиляции, что сводит её потенциал к привычным возможностям препроцессоров во многих системах программирования.

- От равноправия параметров функции перейти к разделению переменных на связанные, получающие значения в точке вызова функции, и свободные, получающие значение в точке определения функции.

- При компиляции автоматически выполнять оптимизацию рекурсий, сводимых к итерациям.

- Отменить право на программируемый вызов «сборки мусора», оставить только автоматический.

- Смягчить неизменяемость данных опираясь на унификацию присваиваний и определений функций, выполненную в своё время при разработке языка Algol 68, а заодно и разрешить изменять значения системных свойств символа (define = set![44]) и даже элементов точечной пары (set-car!, set-cdr!).

- Процедуры работы с устройствами можно рассматривать как функции, вырабатывающие значение #undef, символизирующее, что значение процедуры не имеет смысла.

- Вместо рецептов выполнения функциональных параметров, свободные переменные которых получали значение в точке вызова периода вычислении, формировать замыкания функций в точке их определения при компиляции.

Достойно недоумения, что технический успех реализации языка Scheme ничуть не поколебал общее мнение, что функциональное программирование принципиально не может быть эффективным[45].

6.2. Пресс реализации

На уровне реализации языка одномерные массивы («вектора») рассматриваются в качестве базовой структуры данных как эквивалент списков, перемещённых в расширения. Это изменило роль атома Nil и исказило представление пустого списка, уже не работающего в качестве логического значения «ложь», что мешает переносу программ, но повышает наглядность для преподавателей-математиков введением булева типа данных, содержащего значения (#t и #f[46] вместо T и NIL). Поддержаны на уровне семантики обработка хэш-таблиц и структур, подобных структурам языка C или записям языка Pascal, синтаксическая макротехника в стиле сопоставления с образцом и профилактикой коллизий имён (*hygienic macro*)[47]. Введён явный тип данных «порт» для управления потоками ввода-вывода, возможна передача порта в качестве аргумента.

Scheme стал очередным[48] диалектом языка Lisp, применяющим не динамические, а статические области видимости переменных, подобные блокам языка Algol-60. Это решение обосновали приоритетом компиляции и оптимизацией хвостовой рекурсии, так проще реализация свободных переменных[49]. Впервые появились продолжения и статические замыкания функций, формируемые в позиции определения функции (continuation, closure[50]). Кроме функций работы со списками (cons, car, cdr и eq?) базис

---

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

45 Экспериментальная психология не так давно установила, что оперирование ложной информацией для мозга легче, чем восприятие и воспроизведение достоверных данных.

46 В первых версиях языка Lisp пробовали использовать константу «F» в качестве логического значения «ложь», это как-то не прижилось.

47 Любимая проблема в теории обработки формул. Эта проблема была решена в Lisp 1.5 функцией gensym, вырабатывающей уникальные атомы.

48 В середине 1960-ых появился диалект Lispkit — первая реализация чисто функционального подмножества Pure Lisp с лексической областью видимости. https://github.com/hanshuebner/secd/tree/master/lispkit/LKIT-2

49 Решение этой Funarg-проблемы представлено в определении интерпретатора для Lisp 1.5.

50 Динамический аналог closure под названием FunArg существовал в языке Lisp, работал в динамических областях видимости.

языка Scheme содержит конструкторы выражений и функций (define, lambda, quote, if), оператор присваивания[51] (set!), применяемый к ранее определённым символам, и средства представления синтаксических макросов (define-syntax, let-syntax, letrec-syntax, syntax-rules), функционирующих примерно как препроцессоры многих языков программирования. Расширение базиса содержит средства управления вычислениями и варианты представления функций, выражений и областей видимости (do, let, let*, letrec, cond, case, and, or, begin, named let, delay, unquote, unquote-splicing, quasiquote). Поддержаны ленивые вычисления (delay, force). В списке параметров функции можно выделять параметр для серийных аргументов в форме:

(x y z **. serial**) ; *последний параметр после символа «.» для серийных аргументов*

Примеры программ, начиная с «hello world», могут выглядеть следующим образом:

```
(display '"Hello, World!" )
(define a 10)  ; объявлено значение переменной
(define eval-aplus (delay (+ a 2)))  ; объявлена функция с отложенным суммированием
(set! a 20)   ; присваивание нового значения ранее введённой переменной
(force eval-aplus) ; вызов отложенного вычисления, будет 22
```

Универсальная функция eval не включена ни в базис языка, ни в его расширение, что затрудняет решение новичками задач мета-программирования. В реализации она поддержана на уровне дополнительной библиотеки. Отчасти её роль может выполнять функция force, восстанавливающая отложенные вычисления.

```
(define (evaluate expr)     ; вычисление выражения
   (eval expr (interaction-environment)))
                ; зависит от обстановки, её можно формировать
```

6.3. Изобразительные средства

Кроме построчных «;» и символьных «#» комментариев в языке Scheme имеются блочные, обрамляемые скобками вида «#|» и «|#». Для удобства учёта баланса скобок используются квадратные и, в некоторых реализациях, фигурные скобки[52].

Повышение удобочитаемости программ достигается соглашениями об именовании функций, что можно рассматривать как перенос представления семантики языка на уровень лексики. Имена предикатов заканчиваются символом «?» или «=?».

```
(eq? 'a 'A)              ; равные указатели
(eqv? 5 (+ 2 3))         ; одинаковые значения
(equal? "foo" "bar")     ; совпадение структур
```

В имена обработчиков данных принято включать префикс или суффикс с именем типа данного, например:

(pair? o*bj*))
(port? o*bj*)
(boolean=? *boolean*$_1$ *boolean*$_2$)
(buffer-mode? *Obj*)
(file-exists? *Path*)
(enum-set-constructor *enum-set*)
(hashtable-copy *hashtable*)
(make-vector *n*)

---

51 В те годы специалисты по семантике языков программирования полагали, что если в Lisp включить оператор присваивания, то это всех устроит, не замечая, что Lisp содержит локальные операторы присваивания в форме Prog (set, setq) и оттеснённые на периферию внимания глобальные функции присваивания (rplaca, rplacd).

52 Заодно порождается почва для синтаксических ошибок при несоответствии скобок.

(record-type-name *rtd*)
(string-ci-hash *string*)

Конвертор содержит «->» между именами типов исходных и целевых данных:
(char->integer *char*)
(integer->char *integer*)
(list->string *list*)
(string->list *string*)
(string->symbol *string*)
(string->number *string*)
(list->vector *list*)
(vector->list *vector*)
(bytevector->string *bytevector*)

Имена деструктивных функций завершаются символом «!»:
(set! symbol o*bj*)
(set-car! o*bj*)
(set-cdr! o*bj*)
(vector-sort! *predicate vector*)

Объявление определения функций унифицировано с присваиваниями переменным:
(define f …) ; *определение функции можно потом изменять*
(set! f (lambda (n) (+ n 100)))
;; *ранее объявленной переменной f присваивается новое определение функции*
(map f '(1 2 3)) ; *будет (101 102 103)*

Примеры определения функций:
;; *факториал в рекурсивном стиле*
(define (fact x) ; *объявлено имя функции и её параметра*
(if (< x 2) ; *конструктор условного выражения*
1 ; *при истинном значении предиката, отличном от #f*
(* (fact (- x 1)) x))) ; *при ло;ном значении предиката = #f*
(fact 14) ; *вызов функции*

;; *функция Фибоначчи — требует параллельной рекурсии*
(define (fib n)
(cond ((= n 0) 0) ; *конструктор ветвления*
((= n 1) 1) ; *вторая ветвь*
(else (+ (fib (- n 1)) ; *специальное значение «else» для «иначе»*
(fib (- n 2))))))
(fib 10) ; вызов функции

;; (вспомогательная функция loop реализует цикл с помощью
;; хвостовой рекурсии и накапливающей переменной)
(define (sum-list x)
(let loop ((x x) (n 0)) ; *объявлена внутренняя локальная область видимости*
(if (null? x)
n ; *переменная для накопления результата*
(loop (cdr x) (+ (car x) n)))))
(sum-list '(6 8 100)) *; вызов функции на константном списке*
(sum-list (map fib '(1 2 3 4))) *; вызов функции на вычисленном списке*

Шаблон макроопределения:

```
(define-syntax let
  (syntax-rules ()
    ((let ((var expr) ...) body ...)
      ((lambda (var ...) body ...) expr ...))))
```

Опубликован стандарт де-факто - Report on the Algorithmic Language Scheme (RnRS), что не спасает от заметного разнобоя в реализациях. Похоже, как обычно, разработчики систем программирования кое-в-чём уточняют определение реализуемого языка, подменяют язык его расширенным подмножеством.

Образовательное значение языка Scheme отмечали разработчики языка Java, вспоминавшие в своих первых публикациях о комфорте учебного программирования на языке Scheme в MIT, что повлияло на их решения. Язык Scheme породил заметное число слабо совместимых диалектов и реализаций (GNU Guile, Racket, Chicken Scheme, Jscheme, Kawa и другие), некоторые из которых поддерживают компиляцию на C или в байт-код JVM. В последние годы особое внимание привлекает диалект Racket[53] (ранее — PLTScheme), созданный в 1994 году, предоставляющий среду языково-ориентированного программирования — создание, разработку и реализацию языков программирования. Это симптом перехода практики программирования от накопления правильности вычислений на уровне библиотек к уровню создания проблемно ориентированных языков (DSL).

6.4. Основные отличия от Lisp

Общая характеристика языка Scheme в сравнении с языком Lisp может быть выражена следующим образом:

Язык Scheme заимствовал терминологию и синтаксис языка Lisp, несколько изменяя смысл ряда понятий и сужая трактовку почти всех принципов. Например, от универсальности символьного представления информации происходит возврат к предпочтению процессов обработки чисел и структур над числами. Примитивными данными считаются числа, литеры, булевы значения, порты, строки, символы, замыкания функций, а составными - пары, одномерные вектора, списки, хэш-таблицы и перечни, что следует понимать как изменение границы между базовыми и расширяющими, явными семантическими и неявными прагматическими понятиями. Символы обладают списками свойств для хранения только системной информации о переменных и именах процедур. Системную информацию можно изменять функцией set!. Элементы одной семантической системы могут входить в разные библиотеки (map и for-each - base; exists и for-all - lists). Наличие функций set!, set-car! и set-cdr! допускает создание графов с циклами, поддерживая сдвиг в направлении императивного программирования с нарушением принципа неизменяемости данных, начиная с изначального отсутствия сохраняемой исходной программы при компиляции. Для значения «ложь» введено булево значение #f логического типа данных, для значения «истина» - значение #t. Форма '() понимаются как значение «истина». Синтаксическая макротехника по возможностям уступает специальным функциям языка Lisp. Представление списков кроме круглых скобок использует квадратные.

Программы внешне используют символьные выражения, но при их вводе происходит принудительная компиляция с созданием замыканий функций, хранимых в виде результата компиляции, списочная форма не обязана сохраняться. Кроме того, допускается отклонение от принципа единственного результата функции в виде отсутствия результата процедур. Процедуры вывода и работы с файлами не вырабатывают результат. Связывание переменных с помощью lambda, define, letrec, let-values дополнено функцией присваивания set!, меняющей значения ранее объявленных переменных. Scheme поддерживает не динамические, а статические области видимости

53 https://docs.racket-lang.org/guide/index.html

переменных[54]. Можно откладывать вычисление — delay и отложенное вычисление возобновлять — force. Ряд системных средств, о которых известно программисту, не имеют формы для их применения в программе (GC, compiler). В целом это несколько повышает производительность компилятора и эффективность многократно[55] используемых программ ценой определённого снижения продуктивности экспериментального программирования, особенно отладки.

6.5. Выводы

Таким образом, из характерных особенностей языка Lisp в языке Scheme поддержана примерно треть. Частично сохранены принципы само-определения, равноправия параметров и гибкости границ блоков памяти. Внешне используются кругло-скобочные списки и общая схема представления и вызова функций, включая представление безымянных функций и передачу функциональных параметров. Можно блокировать вычисление любого выражения. Роль предиката может выполнять любая функция, в качестве значения «истина» допускается любое данное кроме значения — «ложь»[56], оно представлено специальным значением #f. Имеется ветвление с произвольным числом альтернатив на уровне расширения языка и арифметические мультифункции. Поддержано программирование обработки ошибок. На уровне прагматики используется механизм автоматизации повторного использования памяти GC без вызова GC из программы. Можно считать, что так выделилось минимальное ядро грамматики языка Lisp. Не исключено, что при сравнении языка Scheme с языком Pascal обнаружилось бы большее сходство.

Более детальное сравнение представлено в таблице 3.

Таблица 3

Сравнение особенностей Lisp и Scheme

Характеристика языка	*Обоснования и решения Lisp*	*Обоснования и решения Scheme*
	Принципы и выбор решений	***Уточнения и изменения***
1. ***Универсальность***	Любую информацию **для компьютерной обработки** можно представить в символьной форме.	Достаточно информационной обработки как процессов над числами или строками и структурами над ними.
2. ***( )***	На терминале не было других скобок.	Кроме круглых скобок используются квадратные, иногда фигурные, что может повышать наглядность.
3. *Символьные формы*	Символьные формы бывают **неделимыми**, если в языке не предусмотрен разбор их на части, или **составными**, если язык допускает их сборку из частей и разделение на части.	Имеются числа, строки, литеры, булевы и специальные значения, а также символы, представляющие переменные, процедуры и функции, без общего понятия «атом». Составные формы (списки и пары) дополнены представлением и реализацией векторов и портов, а также некоторыми неявными структурами.
4. *Атомы*	Символьные формы для представления неделимых данных называются атомами. Любую сущность, включая ключевые слова, можно представить как атом, именовать в соответствии с особенностями области приложения. Атомы выполняют роль **идентификаторов** для значений любой	Общее понятие «атом» отсутствует, имеются символы, хранящие системную информацию, допускающую изменения, но без возможности программировать свои свойства. Результаты компиляции не конструируются функцией CONS и не разбираются на части, что позволяет их считать примитивными подобно строкам, но это зависит от

54 В середине 1960-ых появился диалект Lispkit — реализация чисто функционального подмножества Pure Lisp с лексической областью видимости.

55 Учебные программы студентов обычно имеют разовый характер.

56 На практике это означает, что фактически в качестве логического типа данных используется множество всех допустимых данных, в котором одно данное представляет значение «ложь», а все остальные данные являются представлениями значения «истина».

	природы.	реализации.
5. Константы	Введены так называемые псевдо-атомы, смысл которых виден по форме их записи, - это **числа и строки**. Строки могут разделяться на символы, но по соглашению они считаются псевдо-атомами. Составные формы строятся из пар элементов произвольной природы, разделённых точкой и заключенных в скобки. **Список строится из пар** и завершается парой, содержащей последний элемент и пустой список.	Числа, строки, литеры, булевы и специальные значения, а также символы, представляющие имена переменных, процедур и функций, определяются по их представлению. Результаты компиляции не имеют внешней формы, их само-определимыми не называют. Вектора можно представлять в виде #(1 2 3 …).
6. Списки, иерархия форм	Основная структура составных данных — список, перечисленные элементы которого заключёны в скобки и при необходимости разделяются пробелами. Нет загромождения запятыми или другими разделителями. **Элементы списка могут быть любой природы и разного вида**.	Базовой структурой является вектор, дающий более эффективный доступ к элементам, представленный как список с префиксом # - #(1 2 3 4 5). Сохраняются списки и пары как основные структуры данных, но они перенесены из базиса языка в расширение. Введены функции set-car! и set-cdr!, аналоги деструктивных rplaca и rplacd языка Lisp. Облегчено создание графов с циклами.
7. *Пустой список* ***Nil = ()***	Пустой список не может быть разделён на части средствами языка, поэтому он является **одновременно списком и атомом**. Его можно считать отдельным типом значений.	Форма '() понимаются как значение «истина». Реализация () и Nil различается в разных реализациях, () не всегда вычисляется, может приводить к попытке извлечь из него функцию, поэтому надёжнее '().
8. **Гибкость распределения памяти**	Имена атомов, числа, стоки, списки **не имеют ограничений на длину**, их размеры не зависят от формата машинных слов и границ блоков памяти..	Как в языке Lisp.
9. *Списки свойств атома*	Атомы, смысл которых не ограничен их именем, сопровождаются списком свойств, часть которых определены в системе программирования, другие **могут программироваться.** Разница между переменными и разными категориями функций представляется в списке свойств атомов. При пополнении списка свойств ранее существовавшие свойства не исчезают, хотя по умолчанию используется **самое новое определение.**	Символы обладают списками свойств только для хранения системной информации о идентификаторах переменных и функций. Программируемых свойств символа обычно не предусмотрено. При необходимости их можно моделировать двухуровневыми списками из пар (индикатор_свойства значение_свойства) или одноуровневыми вида (индикатор_свойства1 значение_свойства1 …).
10. ***Неизменяемость данных***	Обработка списков подчинена принципу неизменяемости данных. Преобразованные списки строятся в новой памяти **без искажения исходных данных**.	При интерпретации программы происходит неявная, принудительная компиляция, возможно использующая иные структуры данных и выполняющая оптимизирующие преобразования. Гомоиконность не поддержана.
11. *Функции и процедуры*	Любое понятие программирования можно рассматривать как функцию или как применение функции. Определения функций допускают **самоопределение (рекурсия)** и подчинены принципам равноправия аргументов и единственности результата.	Программы строятся из процедур, возможно не имеющих результата. В таком случае они, в отличие от функций, не могут быть вызваны внутри выражения. Процедуры выполняют связывание переменных (lambda, define, letrec, let-values), изменение их значений (set!), синтаксические преобразования и взаимодействие с внешним миром (порты).
12. **Равноправие параметров функции**	Аргументы функции **не зависят друг от друга**. Даже если при реализации языка порядок их вычисления определён, результат функции не зависит от этого порядка.	Как в языке Lisp.

13. *Конструкторы, макросы, специальные функции*	Можно без предварительного вычисления аргументов функции производить обработку их представлений внутри функции — **специальные функции**. Это позволяет программировать разные схемы управления вычислениями, конструировать представления выражений и разных категорий функций, преобразовывать программы, например, **компилировать** или оптимизировать.	Существует макротехника (hygienic macros) с профилактикой коллизий имён (define-syntax, let-syntax, letrec-syntax, syntax-rules).
14. ***Единственность результата функции***	При вызове функции в одном и том же контексте, при одних и тех же аргументах получается **один и тот же результат.**	Имеются процедуры, вместо результата вырабатывающие специальное значение #undef.
15. *Выражения (формы)*	Программы и выражения представляются как символьные формы — это список, первый элемент которого содержит **представление** функции, а остальные — **представления** аргументов. Пустой список () или атом Nil считаются константной формой, всегда имеющей значение Nil.	Символьная форма — это **непустой** список, первый элемент которого содержит представление функции, а остальные — представления аргументов. При необходимости использовать пустой список () его следует представлять как константу '()[57], хотя во многих реализациях допускается и ().
16. ***Рекурсия***	Именованные формы выражений допускают **само-применимость определений.**	Как в языке Lisp.
17. *Гомоиконность*	Представление программы в виде списков позволяет чётко видеть структуры данных, представляющие программу. Такое преставление программы фактически является **абстрактным синтаксическим деревом.**	Гомоиконности нет, списочная форма программы теряется при принудительной компиляции программы, совмещённой с вводом программы.
18. *Алгебраические системы функций*	Базовые функции над конкретными типами значений группируются в комплекты, обладающие **полнотой операций** и/или отношений, определены обратные функции и минимальные значения, выполняющие роль нулей или единиц.	Элементы одной семантической системы входят в разные реализационные библиотеки уровня прагматики. (cons , car, cdr, eq, set!), - base (if, define, lambda, quote) - base (delay, force) - r5rs, Eval - eval (do, let, let*, letrec, cond, case, and, or, begin, named let, unquote, unquote-splicing, quasiquote) — расширения.
19. Аксиоматика отношений	Поддерживаются аксиомы, позволяющие в отдельных случаях получать результаты функций **без выполнения вычислений**.	Возможно такие отношения учитываются при оптимизирующей компиляции.
	Семантический базис	
20. **Quote**	Любую форму можно заблокировать от вычисления, сделав из неё константу.	Quote работает только для создания констант. Для объявления ленивых вычислений введена специальная функция delay, а для их возобновления — force.
21. ***Eval***	Для любой представленной или построенной в программе формы можно вычислять её значение по мере необходимости, в том числе, вычислять	В документации голословно предупреждают, что использование eval опасно[58]. При организации ленивых вычислений вместо неё используется force. Тем не менее, хотя eval не

57 Можно подумать, что авторы языка допускают, будто в пустом списке может скрываться представление функции, отсутствие которой трактуется как ошибка, о чём сообщает значение #undef.

58 Не ясно, почему «компиляция на лету», выполняющая не более чем контроль типов данных, безопасна, а «интерпретация на лету», способная выполнять полный контроль типов значений, опасна. Более правдоподобно, что после принудительной компиляции вызывать функцию EVAL не удобно из-за несохранения списочного представления программы.

	ранее заблокированные или сконструированные формы.	входит в базис языка, она доступна в форме (eval *obj environment*).
22. ***Lambda*** - *свободные и связанные переменные*	Lambda при **объявлении безымянных функций** выделяет **список связанных переменных**, они получают значение при вызове функции. Определяющая форма функции может содержать и другие переменные, называемые **свободными**, они получают значение из внешнего контекста.	Изменяется подход к означиванию свободных переменных. Их значения выбираются из области видимости в иерархии вложенности определений функции. Встречается конструктор lambda-case, поддерживающий создание полиморфных определений функции, отличающихся форматом списка параметров.
23. *Пространства имён*	Поддержано два пространства имён — раздельно для локальных и глобальных определений с **приоритетом локальных**. Имена переменных и функций расположены **в общем пространстве.**	Приоритет статическому, лексическому пространству имён, формируемому при компиляции.
24. Контекст — иерархия областей видимости, пространства имён, динамика и статика	Если одна функция вызывает другую, то внутренняя через свободные переменные может использовать **значения переменных из объемлющих функций**.	Scheme применяет не динамические, а статические, лексические области видимости переменных[59]. Это значит, что вместо динамики вызовов функций работает иерархия определений функций, её удобнее анализировать при компиляции — статический анализ.
25. ***Label*** - *приоритет локальным определениям*	Локальные определения освобождают от придумывания уникальных имён. Теоретически глобальные определения можно реализовывать как самые внешние локальные.	Для локальных областей видимости используются процедуры **let, letrec** и т.п.
26. ***Define/Defun*** – *глобальные определения*	Кроме создания локальных безымянных и именованных функций имеется **практичный компромисс** в виде глобальных именованных функций (Define/Defun).	Приоритет глобальным переменным (**Define, Set!**), локальные возникают лишь при генерации отдельных областей видимости (let и тому подобные).
27. *Мульти-функции*	Некоторые функции могут быть определены для работы с произвольным числом параметров.	Как и в языке Lisp поддержаны мультизначения и мультифункции. Определения функций могут иметь список параметров вида (x . y), где правый элемент пары означает список произвольного числа параметров — серийные аргументы.
28. *Полиморфизм*	Атом может одновременно обладать разными свойствами, возможно **неоднократное вхождение** любого свойства. Атом может представлять и значение, и несколько разных категорий определения функции, и ещё что-нибудь по усмотрению программиста. Функция может иметь ряд различных определений, включая одновременное вхождение символьных и кодовых определений, например, результатов компиляции. По умолчанию работает самое новое определение, но и **к прежним определениям доступ возможен.**	Полиморфизм поддержан на уровне сигнатур функций и с помощью lambda-case, поддерживающего создание полиморфных определений функции, отличающихся форматом списка параметров.
29. Функции высших порядков	Представления аргументов функции могут быть **символьными формами определения** или представления функции. Поэтому можно определять	Как и в языке Lisp поддержаны функции высших порядков, встроены отображения, фильтры и свёртки.

59 В середине 1960-ых появился диалект Lispkit — реализация чисто функционального подмножества Pure Lisp с лексической областью видимости.

	функции высших порядков, использующие функциональные переменные. Аргументами и/или результатами функции высших порядков могут быть представления других функций.	
30. Отображения, свёртки и фильтры	Применение функций, передаваемых через параметры, требует согласования не только типов данных или значений, но и рангов функций и контекста их вызова, что можно реализовать конструированием **рецепта применения функциональной переменной**.	Как и в языке Lisp имеются встроенные отображения и фильтры и возможность их программировать.
31. Функциональные переменные	Представления функций — не более чем символьные формы, поэтому нет никаких препятствий существованию функциональных переменных, значением имеющих представление функции, и передаче представлений функций в качестве параметров при вызове функций или **выдаче их как результат.**	Как и в языке Lisp поддержаны функциональные переменные с возможностью их передачи в качестве результата, но без конструирования из элементов, хотя это не исключено.
32. Безымянные определения	Передаваемые через параметры функции часто имеют **разовый характер**, а именование функций осмысленно для многократного использования. Безымянные функции строит конструктор Lambda.	Как и в языке Lisp имеется возможность формировать безымянные функции (lambda), дополненная конструктором lambda-case, поддерживающим создание полиморфных определений функции, отличающихся форматом списка параметров.
33. ***Cond - If*** *ветвление*	В Pure Lisp хотя бы одна ветвь должна быть выбрана. В Lisp 1.5 в форме Prog это не обязательно, при отсутствии истинного предиката значением ветвления является Nil.	Ветвление должно содержать истинный предикат, иначе вырабатывается значение #undef. Имеется предикат else. Введён привычный If.
34. ***Eq*** *– сравнение указателей или атомов*	Существуют полный комплект предикатов для **контроля процесса выполнения** программы.	Различается сравнение указателей, значений и структур данных: eq?, eqv?, equal?.
35. **Atom**, Null, Number - контроль типов значений	**Каждый тип данных имеет предикат**, позволяющий в любой момент интерпретации программы, выяснять принадлежность значений нужному типу данных.	Значениями считаются числа, литеры, булевы значения, порты, пары, списки, строки, одномерные вектора, символы, хэш-таблицы и перечни, для которых встроены распознающие предикаты: pair?, symbol?, number?, string?, null? и другие.
36. *Объявление типов данных*	**Для компиляции функций следует заранее объявлять тип данных**, передаваемых через свободные переменные.	Типы данных устанавливаются при компиляции.
37. *Предикаты*	**Любая функция** может выполнять роль предиката. Ложным значением считается значение (), оно же Nil, остальное — истина.	Принято соглашение, что для наглядности имена предикатов завершаются литерой «?». Ложным значением считается булево значение #f, остальное — истина. Существует булево значение #t, представляющее истину. Возникает необходимость сравнения с #f.
38. ***Логика -*** *( ) Nil, значение «ложь»*	Nil выполняет роль значения «ложь», любое другое значение работает как «истина». Поиск в списках при неудаче **завершается на пустом списке**. Эффективно пустой список рассматривать как значение «ложь», подобно коду «0» во многих языках.	Введён **булев тип данных**, что несколько загромождает представление ветвлений необходимостью сравнивать результаты с логическим значением #f: (cond ((null? xl) '()) ; *проверка пуст ли список* (else (cons al xl))) **; вместо** (cond (xl (cons al xl)) ; *при непустом списке* ; *на пустом списке значение* ()

39. *Компиляция* - ***compile***	Компиляция функций рассматривается как **оптимизация программ**, позволяющая повысить скорость их выполнения примерно в 50 раз ценой существенного расхода времени на обработку программы. Это достаточное основание подвергать компиляции достаточно отлаженные функции, чтобы избежать накладных расходов на компиляцию программ с неотлаженными функциями.	**Принудительная компиляция** программы, выполняемая непосредственно при вводе программы.
	Диагностика и отладка	
40. **Error** – диагностика ошибок	Функция Error - обработчик ошибок, сообщающий диагноз дефекта в определении выражений и позволяющий **программировать продолжение** вычислений.	Поддержана функция реакции на ошибки (error *who msg irritant ...*)
41. ***Trace/Untrace*** – *отладка*	Режим **REPL** (Read-Eval-Print Loop) Поддержана возможность смотреть **аргументы-результаты отлаживаемых функций** по ходу их вычисления, список которых может быть при необходимости изменён. **Trace** – принимает список функций, значения аргументов и результаты которых следует выдавать при каждом вызове. Для этого в списки свойств функций размещается флаг. **Untrace** – удаляет флаг из списка свойств функций, отслеживание которых уже не нужно.	Режим **REPL** (Read-Eval-Print Loop) и процедура display, **не вырабатывающая результата**. При желании на её основе можно смоделировать тождественную функцию print, приспособленную к вставке в любой позиции вычисляемой формы.
	Расширения и внешний мир	
42. ***Открытость***	Программа имеет доступ ко всем функциям системы программирования (интерпретатора, компилятора, отладчика) и может им давать свои определения. Всё, что понадобилось системе, может быть полезным и программисту. **Изменять можно любые определения, кроме Nil.**	Ряд системных средств, о которых известно программисту, **не имеют формы для их применения в программе**. GC, compiler и другие.
43. ***Print, Read*** – *ввод-вывод* ***I/O***	Print формально является **тождественной функцией вывода** данных, он печатает свой аргумент на внешнем носителе. Ввод принимает данное с внешнего устройства, строит эквивалентную ему символьную форму, становящуюся его результатом, что удобно при переходе от обработки констант к обработке произвольных данных.	**Порт** — структурный тип данных. Процедура вывода display не имеет результата, что ограничивает её свободу использования при отладке.
	Прагматика реализации	
44. ***GC*** – *«Сборка мусора»,* ***гибкость распределения памяти***	Автоматизировано **повторное использование памяти**, чтобы освободить специалиста для более существенных задач. Возможен **вызов GC из программы**.	Нет процедуры GC, вызываемой из программы, **только неявная, автоматическая** сборка мусора.
45. ***Структуры данных***	Хотя на уровне языка нет ни векторов, ни хэш-таблиц, в реализации поддержана работа с **векторами для взаимодействия с машинным кодом**	Введена явная возможность вводить **вектора как списки с префиксом #**. Вектор из пяти чисел: #(1 2 3 4 5). Остальные структуры работают неявно, хотя на

	процедур и с **хэш-таблицами для идентификации** атомов. Кроме того, в языке поддержаны две функциональные модели хэш-таблиц — **глобальный список свойств атома и локальный ассоциативный список**, локально связывающий атомы с их определениями, позволяющие не нарушать принцип неизменяемости данных..	уровне семантики к ним поддержан доступ с помощью специальных функций или процедур.
46. ***Присваивания*** *и адреса*	Присваивания поддержаны в двух формах. Это псевдо-функции **Rplaca и Rplacd,** изменяющие значение по указанному адресу, и функции **Set и Setq внутри формы Prog**, выполняющие изменение значений рабочих переменных без побочного эффекта на внешнем уровне, что позволяет Prog рассматривать как **чистую функцию**, совмещающую декларативный ответ на вопрос «Что?» с императивным ответом на вопрос «Как?».	**Присваивания стали явными** процедурами (**set!** и другие), работающие на ранее введённых переменных или структурах данных. Принято соглашение, что имена деструктивных процедур завершаются символом «!».
47. ***Замыкание функции***	Для передачи функций со свободными переменными в качестве функциональных параметров интерпретатор выполняет **формирование рецептов**, однозначно связывающих переменные **в точке вызова функции с её контекстом**, а не в точках её вычисления, в которых пространства имён могут различаться.	Замыкание функций **строится автоматически** для внутренних функций при чтении их определения, т. е. неявно.
48. ***Устройства*** *ввода-вывода данных и ОС*	Имеются функции, поддерживающие доступ к периферийным устройствам и операционной системе.	Как в языке Lisp с дополнительной структурой данных «порт».
49. ***Раскрутка*** *компилятора и интерпретатора*	Интерпретатор и компилятор для языка Lisp первоначально были определены на языке Lisp, а затем воспроизведены в машинном коде.	Известны **эксперименты по самоопределению** языка Scheme, выполнявшиеся в ранге учебных проектов для **обучения мета-программированию**.
50. **Альтернативность** решений, полное пространство для мысли: «Если нельзя, но очень хочется, то можно».	• Реализация вычислений с помощью **обычных функций**, выполняемых после вычисления параметров, дополнена **специальными функциями**, вместо значений параметров, обрабатывающих представления аргументов. • Кроме организации **правильных вычислений** поддерживается представление обработки **ошибочных ситуаций** и продолжения вычислений. • Пространство **абстрактных атомов** расширено конкретными **типами данных**, принятыми в основных областях приложения — псевдо-атомы. • **Автоматизация** «сборки мусора» допускает **вызов GC из программы**. • Кроме интерпретации выражений (**eval**) поддержана компиляция функций (**compile**).	• Реализация вычислений с помощью **обычных функций**, выполняемых после вычисления параметров, дополнена **синтаксической макротехникой и организацией ленивых вычислений.** • Кроме организации **правильных вычислений** поддерживается представление обработки **ошибочных ситуаций и продолжений**. • **Обработка списков** может реализовываться как **обработка векторов,** приспособленная к эффективному изменению значений переменных.
51. Изобразительные средства **( ) . ' ;**	Бедный синтаксис освобождает от необходимости размышлять на тему, «где поставить запятую, где не ставить	() [] {} - границы списков для упрощения контроля баланса скобок и выделения отдельных конструкций — клаузы в ветвлениях, списки параметров и др. **.** - разделитель элементов пары, из которых

	запятой». **( )** - границы точечной пары или списка, **.** - разделитель элементов пары, из которых строятся все структуры данных, **'** - кавычка — префикс, объявляющий представление константы, **'**x эквивалент (Quote x). **;** - объявление начала строчного комментария.	строятся все структуры данных, **'** - кавычка — префикс, объявляющий представление константы, **'**x эквивалент (Quote x). **;** - объявление начала строчного комментария. **#| |# - скобки блочного комментария.** # - специальный префикс для векторов и особых случаев. **Синтаксис обогащён для удобочитаемости.**
	Примеры	
	HomeLisp 1.13.65 (defun next (xl) (cond (xl (cons (1+ (car xl)) (next (cdr xl))))))	https://rextester.com/l/scheme_online_compiler (define (next xl) (cond ((null? xl) '() ; *Проверяем, является ли список пустым* (else (cons (add1 (car xl)) (next (cdr xl))))))

Анализ таблицы 3 показывает, что в языке Scheme несколько ограничены принципы универсальности, неизменяемости данных, независимости параметров и единственности результатов. Представление пустого списка освобождено от роли ложного значения, введен булев тип данных (#f, #t), что влечёт необходимость сравнения с #f, отсутствуют абстрактные атомы, остаются только идентификаторы, для которых программируемые свои свойства не предусмотрены, принудительная компиляция не сохраняет исходный код программы, что означает исчезновение гомоиконности, присваивания стали явными процедурами (**set!** и другие), используются процедуры, не имеющих результата **(**#undef), вместо специальных функций предлагаются синтаксические макросы без конструирования в динамике функций из элементов, ветвления требуют истинного предиката (#undef, else), различаются механизмы означивания связанных (в позиции вызова) и свободных (зависят от иерархия определений) переменных, что приводит к приоритету статического, лексического пространству имён и глобальных, но изменяемых определений (Define, Set!), типы данных для переменных устанавливаются при компиляции, элементы одной семантической системы часто размещены в разные библиотеки, подчёркивается опасность применения универсальной функции eval, ряд функций системы не имеют формы для их применения в программе (GC, compiler и другие). Известны учебные эксперименты по само-определению языка Scheme для студентов, обучаемых разработке языков и систем программирования.

Таким образом, язык Scheme наследует менее половины особенностей языка Lisp и добавляет к ним использование квадратных скобок «**[ ]**», блочных комментариев (#| |#) и структуры данных **port**, перемещены вектора с уровня неявной прагматики на уровень явного синтаксиса #(1 2 3 4 5), макротехника решает вопрос профилактики коллизий имён**,** для ленивых вычислений введены функции delay и force, при определении функций поддержан полиморфизм (lambda-case), серийные параметры в виде конструкции (x **.** y) и локальные области видимости (let, letrec), введён привычный if и соглашение об использовании знака «**?**» в именах предикатов, знака «**!**» для меняющих данные процедур и «**->**» для преобразований типов данных.

7. CLISP - Common Lisp для производственного применения

В 1984 году появился диалект Common Lisp — мультипарадигмальный язык общего назначения, фактически являющийся наследником первых версий языка Lisp, обеспечивающий частичную обратную совместимость с оригинальным языком Lisp Джона Маккарти, что позволяет переносить старое программное обеспечение на Common Lisp. Некоторые особенности языка Lisp конкретизированы и/или дополнены с целью повышения продуктивности программирования и производительности программ. CLISP

допускает статическое связывание переменных без отказа от динамического, поддерживает макросы, функционалы и лексические замыкания функций. В 1995 году CLISP был стандартизован ANSI. Министерство обороны США оказало организационную и финансовую поддержку формированию промышленного стандарта языка Lisp, который и приняло в качестве дополнительного средства разработки ПО для военных применений [41, 42].

Этот диалект резко расширил сферу производственного применения языка, используя на уровне семантики средства работы с матрицами, хэш-таблицами, программируемыми структурами данных, подобными структурам в языке C, и мультизначениями, удобными для моделирования потоков. Всё это представляется как списки, но на уровне прагматики реализуется как эффективные структуры данных.

7.1. Наследство Lisp

Common Lisp был разработан в начале 1980-ых с целью объединения полезных механизмов большого числа разрозненных диалектов языка Lisp. Доступно несколько реализаций CLISP, как коммерческих, так и свободно распространяемых. Этот язык, поддерживает функциональное, императивное, рефлексивное, реляционное и объектно-ориентированное программирования (пакет CLOS), а также мета-программирование и обобщённое программирование. В состав наиболее известной системы CLISP входят интерпретатор, компилятор и отладчик, а также средства для настройки лексики и синтаксиса языка, стыковки с другими языками программирования и интернационализации. Большинство современных реализаций поддерживает юникод, что позволяет включать в алфавит кириллицу. CLISP написан на C и Common Lisp, стандартизирован ANSI, запускается на всех Unix-подобных операционных системах, а также на Microsoft Windows. Он испытал влияние языков Lisp, Lisp Machine Lisp, MacLisp, InterLisp и Scheme и повлиял на языки Clojure, Factor, Dylan, Emacs Lisp, EuLisp, ISLisp, *Lisp, AutoLisp, Julia, Moose, R, SKILL, Stella, SubL и Common-lisp.net.

Common Lisp динамический язык программирования, он облегчает эволюционную и поэтапную разработку программного обеспечения с итеративной компиляцией в эффективные программы времени выполнения. Такая поэтапная разработка часто выполняется в интерактивном режиме, не прерывая работающее приложение. CLISP поддерживает как лексическую (статическую — иерархия определений), так и динамическую (иерархия вызовов функций) области видимости. Существуют макросы и специальные функции, формирующие области видимости (lambda, defun, let, let*, flet, labels) — контексты, в которых доступны объявляемые имена атомов. Есть раздельные пространства имен для разных категорий атомов (функций или переменных) или символов, называемые «пакетами». Пакет может использовать другие пакеты.

К скалярным типам значений языка Lisp (атомы, числа и строки), включающим целые произвольной длины, дроби, числа с плавающей запятой и комплексные числа, CLISP добавляет аппаратные типы чисел для эффективной арифметики. Дроби — возможность, редко доступная в других языках. CLISP автоматически подбирает представления числовых значений между этими типами по мере необходимости. В стандарт языка добавлен булев тип данных, содержащий атомы Nil и T. Атом, имеющий свойства, называется «символ». Понятие символа — одно из ключевых для языка CLISP, это уникальный указатель на объект из данных, включающих в себя: имя, значение, функцию, список свойств и пакет. Некоторые символы представляют сами себя — само-определимые символы. Таковы булевы значения — T и NIL Само-определимыми являются числа и строки.

Списки, как и в языке Lisp, строятся из точечных пар функцией cons[60] и разбираются на части операциями car[61] и cdr[62]. Имеются предикаты eq и atom для

60 Consolidation
61 Content of Address part of Register
62 Content of Decrement part of Register

сравнения и отличия атомов от составных данных. Реализация списков свойств, глобальных переменных и деструктивных функций использует неявное понятие «поле». Существуют функции, обеспечивающие доступ к значениям полей по конкретным адресам (setf, set, setq, get, rplaca, rplacd, nconc, nsubst, delete, nreverse, nunion, mapcan, mapcon и другие).

Составные данные кроме списков могут быть многомерными массивами с динамически изменяемыми размерами, хэш-таблицами с любыми объектами в качестве ключа или значения и структурами, не имеющими явного представления, обрабатываемыми с помощью специальных функций на уровне семантики. Они создаются функциями с префиксом «make-».

Среди составных данных выделены последовательности — это списки, векторы, битовые векторы и строки. Существуют функции, которые могут работать с любым видом последовательности.

Массивы могут содержать любой тип значений в качестве элемента, даже смешивать разные типы в одном массиве, или могут быть специализированы, содержать только определённый тип. Есть два стандартных типа массивов: строка является вектором символов (char) и вектор битов. Вектор — это одномерный массив.

Хеш-таблица хранит отображение между данными. Хеш-таблицы могут автоматически изменять размер, как в языке Setl. Структуры, аналогичные структурам в C и записям в Pascal (за исключением вариантных полей), представляют произвольные сложные структуры данных с любым количеством и типом полей. Структуры допускают одиночное наследование. Классы — часть пакета CLOS, — похожи на структуры, но они обеспечивают возможности множественного наследования и более динамичного поведения. Объекты, созданные из классов, называются экземплярами. Особым случаем являются обобщённые функции, являющиеся одновременно функциями и экземплярами.

Другие типы данных представляют файлы и каталоги в файловой системе. Можно конструировать свои типы данных макросом deftype и распознавать типы данных функцией typep.

Common Lisp поддерживает концепцию мультизначений, где любая функция, в соответствии с принципом единственности результата, всегда имеет одно первичное значение, но она также может иметь любое количество вторичных значений, которые можно рассматривать как имитацию параллельных процессов. Мультизначения поддерживаются несколькими специальная функциями с префиксом «multiple-» и функцией «values», дающей полный результат в виде списка основного и вторичных значений.

Определения функций, макросов и их вызовы представляются списками, в которых первый элемент является представлением функции, а остальные, — представлениями аргументов. Макрос defun конструирует функцию, принимает в качестве аргументов имя функции, список имён её параметров и тело функции:

```
(defun square (x)  ; Определение функции square, получающей один аргумент
    (* x x))       ; и возвращающей его квадрат.

(square 42.1)      ; вызывает функцию square.
                   ; и возвращает её квадрат (1772,41).
```

Представления функций и их определений в языке CLISP могут быть значениями аргументов других функций, принимающих эти представления в качестве аргументов или возвращающие такие представления, возможно построенные при вычислении. Из-за разделения областей видимости переменных и функций для вызова функциональных переменных потребовалась функция funcall. В таком случае используется функция function, сокращённо обозначаемая как «#'». Например, функция sort берёт функцию

сравнения и, возможно, функцию ключа, чтобы сортировать структуры данных согласно ключу.

```
;; Сортирует список, используя функции > и <.
(sort (list 5 2 6 3 1 4) #'>)   ; возвращает (6 5 4 3 2 1)
(sort (list 5 2 6 3 1 4) #'<)   ; возвращает (1 2 3 4 5 6)

;; Сортирует список по первым элементам подсписков.
   (sort (list '(9 A) '(3 B) '(4 C)) #'< :key #'first)   ; возвращает ((3 B) (4 C) (9 A))

(defun map-el (fn  xl) ; Поэлементное преобразование XL
                  ; |_____________________ с помощью функции FN
       (cond ; Пока XL не пуст
              (xl (cons (funcall  fn (car  xl)) ; применяем FN к голове XL
                                  (map-el  fn (cdr  xl)) ; и переходим к остальным,
)     )       )   ) ; собирая результаты в список

(defun  next (xl) (map-el #'1+  xl)) ; Очередные числа
(defun  1st (xl)   (map-el #'car  xl)) ; "головы" элементов
(defun  lens (xl) (map-el #'length  xl)) ; Длины элементов
```

При вычислении различают следующие категории функций: специальные функции из определённого в реализации списка, определенный ранее макрос, имя или lambda-определение функции. Анонимные функции, определяемые с помощью lambda, удобны при их однократном применении, например, в отображениях.

```
(lambda (x) (* x x))  ; функция возведения в квадрат.
```

Пространства имен глобальных функций (defun, flet, labels, defmethod и defgeneric) отделены от локальных пространств имен для переменных (let, let*)[63] . Локальные функции с помощью flet и labels создают раздельные пространство имён (области видимости) для простых и рекурсивных функций соответственно. Для рекурсивных вызовов строится свой собственный экземпляр лексической среды — свой контекст, своя область видимости, защищающая поколения параметров рекурсии.

```
(flet ((square (x)
   (* x x)))
   (square 3))
```

Макрос в языке Common Lisp выглядит как функция, но вместо выражения, которое вычисляется, выполняется преобразование исходного кода программы.[64] Макрос получает представления аргументов и вычисляет полученную новую исходную форму определения. Макрос defgeneric определяет обобщённые функции. Такие функции представляют собой набор методов. Макрос defmethod определяет методы. В случае вызова обобщённой функции можно выбирать эффективный метод.

```
(defgeneric add (a b))
(defmethod add ((a number) (b number))
  (+ a b))
(defmethod add ((a vector) (b number))
```

---

63 http://www.nhplace.com/kent/Papers/Technical-Issues.html Является ли отдельное пространство имен для функций преимуществом, является источником разногласий в сообществе Lisp. Обычно это называют дебатами о Lisp-1 и Lisp-2. Lisp-1 относится к модели Scheme, а Lisp-2 относится к модели Common Lisp. Эти названия были придуманы в статье 1988 года Ричарда П. Габриэля и Кента Питмана, в которой два подхода подробно сравниваются.

64 Подобно работе специальных функций языка Lisp

```
  (map 'vector (lambda (n) (+ n b)) a))
 (defmethod add ((a vector) (b vector))
  (map 'vector #'+ a b))
(defmethod add ((a string) (b string))
   (concatenate 'string a b))

(add 2 3)              ; результат 5
(add #(1 2 3 4) 7)              ; результат #(8 9 10 11)
(add #(1 2 3 4) #(4 3 2 1))     ; результат #(5 5 5 5)
(add "COMMON " "LISP")          ; результат "COMMON LISP"
```

Дуг Хойт в книге «Let Over Lambda» при обсуждении макросов, утверждает, что «Макросы — это единственное величайшее преимущество lisp как языка программирования и единственное величайшее преимущество любого языка программирования». Хойт приводит несколько примеров итеративной разработки макросов. Макросы позволяют создавать новые синтаксические формы, например, новые структуры управления. Ниже пример макроса until, предусматривающего цикл «до тех пор, пока». Синтаксис:

```
    (until test form*)
;; Макро-определение для  until:
  (defmacro until (test &body body)
   (let ((start-tag (gensym "START"))
       (end-tag   (gensym "END")))
    `(tagbody ,start-tag
       (when ,test (go ,end-tag))
       (progn ,@body)
       (go ,start-tag)
       ,end-tag)))
```

Императивное программирование поддержано в лексической области видимости (неявный ProgN) и внутри форм Prog, где можно назначать в TAGBODY метки, передавать на них управление (go x), объявлять глобальные переменные (defvar *stashed*) и устанавливать их значения (setf *stashed* (lambda () (go some-label))). Проблемы с коллизиями имён помогает решать создающая уникальные символы функция gensym и пакеты, обладающие независимыми пространствами имён.

В реализациях CLISP, поддерживающих много-поточность, динамические области видимости разделены для каждого процесса выполнения. Таким образом, специальные переменные служат абстракцией для локальной памяти процессов.

Кроме организации регулярных вычислений, можно программировать обработку прерываний и ошибок (error, catch, throw). Имеются функции слежения за ходом вычислений, включая пошаговое выполнение (trace, untrace, step).

Функции или вся программа могут быть скомпилированы функцией *compile*. Определения функций могут включать директивы (*declare)*, которые дают подсказки компилятору о типах свободных переменных или об оптимизациях. CLISP позволяет компилировать как отдельные функции Lisp в памяти, доступные через индикатор SUBR. так и целые файлы во внешний хранимый скомпилированный код (fasl-файлы). Исходная списочная структура определения функции сохраняется в её списке свойств и доступна для дальнейшего применения через индикатор EXPR.

Язык CLISP проводит различие между временем чтения, временем компиляции, временем загрузки и временем выполнения и позволяет пользовательскому коду также проводить это различие для выполнения желаемого типа обработки на нужном этапе.

В системе Clisp реализована полная поддержка ООП пакетом CLOS. Восходящие к λ-исчислению механизмы наследования в языке Lisp на базе свободных переменных на основе иерархии определений и вызовов функций для поддержи ООП могут быть уточнены иерархией списков свойств. Пример создания собственной модели ООП в 8 строк на языке Lisp опубликован в книге Пола Грэма (*Paul Graham*) «ANSI Common Lisp» на базе иерархии хэш-таблиц[65]. Пол Грэм является автором книг «On Lisp»[66] (1993), «ANSI Common Lisp» (1995), и «Hackers & Painters» (2004). В книге «On Lisp» Пол Грэм подробно описал использование макросов в языке CLISP. Одним из успешных проектов Пола Грэма и компании Y Combinator стало создание социально-новостного сайта Hacker News и участие в разработке программного обеспечения Viaweb, написанное на Common Lisp, позволяющего пользователям создавать свои собственные интернет-магазины. Летом 1998 года Viaweb был приобретён компанией Yahoo!. Журнал BusinessWeek включил Пола Грэма в список 25 наиболее влиятельных людей (*The 25 Most Influential People on the Web*) за 2008 год.

Таким образом Common Lisp даёт ответ на вопрос: «**ЧТО** может дать функциональное программирование программной индустрии?». Ответ получен следующими решениями:

- Универсальность символьной обработки и структур данных неограниченного размера дополнена средствами обработки конечных чисел и неявных структур данных, типичных для большинства языков программирования.
- Компиляция отдельных функций допускает и компиляцию полной программы без потери исходного списочного представления.
- Равноправие независимых параметров функции, допускающих вычисление в любом порядке, уточняется их спецификацией на позиционные, факультативные, ключевые и серийные.
- Поддержаны раздельные пространства имён для разных категорий атомов, форм, моделей вычислений и приложений.
- Самоопределение в форме рекурсии обогащено разнообразием схем циклов, по возможностям превосходящих типовые схемы.
- 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 данных ограничивается введением понятия «поле» для работы с системными свойствами символа и для обременительных функций предоставлением их деструктивных аналогов, приспособленных 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.
- Кроме макротехники имеются средства настройки алфавита, лексики, диагностических сообщений и фиксации модифицированных версий системы для их независимого применения в виде проблемно-ориентированных версий.
- К обычным средствам отладки добавлен режим пошагового выполнения вычислений с показом не только аргументов-результатов функции, но и состояния стека с возможностью манипулирования значениями переменных, определениями функций и методами продолжения вычислений.

7.2. Сравнение со Scheme

Common Lisp часто сравнивают и противопоставляют языку Scheme — это два самых популярных диалекта Lisp. Scheme предшествовал CLISP и исходит не только из

65 http://www.paulgraham.com/avg.html, http://www.paulgraham.com/rootsoflisp.html
66 https://paulgraham.com/onlisp.html

той же традиции Lisp, но и от одних и тех же разработчиков: Гай Стил, вместе с которым Джеральд Джей Сассман разработал Scheme, возглавлял комитет по стандартизации Common Lisp. Common Lisp иногда называют Lisp-2, а Scheme — Lisp-1, имея в виду использование CLISP отдельных пространств имен для функций и переменных[67]. (На самом деле, CLISP имеет и другие пространства имен, например, для тегов go, имен блоков и ключевых слов цикла, для разных пакетов).

CLISP существенно отличается от Scheme общей схемой обработки программ, в нём компиляция отделена от интерпретации, и сохранено традиционное соглашение языка Lisp о роли T и NIL в качестве булевых значений и то, что NIL также обозначает пустой список. Scheme использует специальные значения #t и #f для представления истины и ложности. В CLISP любое значение, отличное от NIL, рассматривается как истинное при выполнении условных операторов, например if, тогда как в Scheme все значения, отличные от #f, рассматриваются как истинные. Значение (), которое эквивалентно NIL и рассматривается как ложь в Common Lisp, Scheme рассматривает как true в логическом выражении.

Заблуждение о том, что Lisp — это чисто интерпретируемый язык, возможно, связано с тем, что реализации языка Lisp обычно предоставляют диалоговый стиль работы с программой, при котором фрагменты программного кода компилируется по мере необходимости. Это заблуждение не исчезло при появлении диалекта Scheme, использующего совмещение чтения программы с её компиляцией. В CLISP широко используется инкрементальная компиляция. Некоторые компиляторы компилируют код CLISP в код C.

7.3. Применение и реализации

Наиболее популярны сейчас реализации CLISP, CMUCL, Mocl.

CLISP — компактная версия Lisp-системы, поддерживающая длинную целочисленную арифметику, создание исполняемых файлов и интерфейс для вызова низкоуровневых функций, например, написанных на C и т. п., включая компиляцию байт-кода.

CMUCL — высокоэффективная версия [68], разработана в Университете Карнеги-Меллона, сейчас поддерживается как бесплатное программное обеспечение с открытым исходным кодом группой добровольцев. CMUCL использует более быстрый компилятор собственного кода. Он доступен в Linux и BSD для Intel x86; Linux для Альфа; macOS для Intel x86 и PowerPC; и Solaris, IRIX и HP-UX на их родных платформах.

Mocl - для iOS, Android и macOS.

Common Lisp используется для разработки исследовательских приложений, часто в области искусственного интеллекта (ИИ), при быстрой разработке прототипов или для развернутых приложений, используется во многих коммерческих приложениях, включая Yahoo. Сайт интернет-торговли Store, в котором первоначально участвовал Пол Грэм, затем был переписан на C++ и Perl.

Другие упомянутые в Википедии примеры включают в себя крупные системы для American Express, Orbitz и Kayak.com, American Airlines, Continental Airlines и US Airways. Boeing, Airbus и Northrop Grumman (ITA Software, Piano и др.), а также SigLab для обработки сигналов в противоракетной обороне и систему планирования миссии НАСА Mars Pathfinder.

Кроме собственно ИИ, Common Lisp активно применяется в разработках систем массового назначения, основанных на экспертном знании и на здравом смысле в языках спецификаций, при создании видео-игр Naughty Dog, пакетов 3D-графики, среды

67 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.

68 https://www.cons.org/cmucl – особо эффективная реализация Clisp.

визуального программирования для компьютерной сборки и синтеза звука, работы с музыкой, создан мозаичный оконный менеджер X11, модуль для улучшения письма на английском языке (ACT-R, Cyc, Gensym G2, Genworks GDL, Jak и Daxter, Mirai для анимации лица Голлума в фильме «Властелин колец: Две башни», PWGL, Opusmodus, Reddit, Stumpwm, Grammarly).

Заметное число применений посвящено профессиональной поддержке программирования, включая систему проверки моделей (PVS), прототип сборщика мусора Microsoft .NET Common Language Runtime. загрузчик данных для PostgreSQL (Pgloader), средства динамического анализа и перепланирования (DART), автоматизированное средство доказательства теорем (ACL2) и системы компьютерной алгебры (Аксиома, Maxima).

7.4. Выводы

Таким образом, сохраняя или восстанавливая основные концептуальные решения языка Lisp, диалект Common Lisp их дополняет, расширяя пространство типов значений и структур данных, поддерживая определение своих структур данных, конкретизируя представление функциональных параметров, допуская использование лексической области видимости без потери динамической, упрощая форматы ветвлений, пополняя состав деструктивных аналогов малоэффективных функций и возможности компиляции, повышая естественность работы с макросами, предоставляя средства отладки программ, поддерживая организацию прерываний и взаимодействие с низкоуровневыми средствами и ОС, включая функции работы с файлами. Введены понятия поле, пакет, символ, мультизначения. Сохранены без изменений общая схема представления списками других структур данных, механизм работы с функциями, реляционная работа с атомами и списками свойств, возможность приостановки и возобновления вычислений, использование любых функций в качестве предикатов, мультифункции, функциональная природа обмена данными, возможность программировать обращения к любыми средствам системы, включая GC. 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.

Таблица 4

Различия Lisp и Common Lisp

Характеристика языка	*Обоснования и решения Lisp*	*Обоснования и решения CLISP*
	Принципы и выбор решений	***Уточнения и изменения***
1. **Универсальность**	Любую информацию **для компьютерной обработки** можно представить в символьной форме.	Как в языке Lisp.
2. **( )**	На терминале не было других скобок. Зато и теперь они набираются на любом регистре, в отличие от фигурных и квадратных	Как в языке Lisp.
3. Символьные формы	Символьные формы бывают **неделимыми**, если в языке не предусмотрен разбор их на части, или **составными**, если язык допускает их сборку из частей и разделение на части.	Кроме символьных форм языка Lisp, CLISP на уровне семантики поддерживает **многомерные вектора и хэш-таблицы** и позволяет строить **свои типы данных**, внешне выглядящие как списки, допускающие более эффективные функции доступа.
4. Атомы	Символьные формы для представления неделимых данных называются атомами. Любую сущность, включая ключевые слова, можно представить как атом, именовать в соответствии с	В отличие от списков свойств в языке Lisp, подчинённых принципу неизменяемости данных, в CLISP системные свойства обрабатываются деструктивно. Для этого введено неявное понятие «**поле»**

	особенностями области приложения. Атомы выполняют роль **идентификаторов** для значений любой природы.	Атом со списком свойств называется **символ.**
5. *Константы*	Введены так называемые псевдо-атомы, смысл которых виден по форме их записи, - это **числа и строки**. Строки могут разделяться на символы, но по соглашению они считаются псевдо-атомами. Составные формы строятся из пар элементов произвольной природы, разделённых точкой и заключенных в скобки. **Список строится из пар** и завершается парой, содержащей последний элемент и пустой список.	К общему представлению чисел добавлены **аппаратно-представимые числа** ради эффективности их обработки.
6. Списки, иерархия форм	Основная структура составных данных — список, перечисленные элементы которого заключёны в скобки и при необходимости разделяются пробелами. Нет загромождения запятыми или другими разделителями. **Элементы списка могут быть любой природы и разного вида**.	Как в языке Lisp.
7. Пустой список **Nil = ()**	Пустой список не может быть разделён на части средствами языка, поэтому он является **одновременно списком и атомом**. Его можно считать отдельным типом значений.	Как в языке Lisp., но введен **логический тип данных** без отмены роли Nil и пустого списка () в качестве значения «ложь».
8. **Гибкость распределения памяти**	Имена атомов, числа, стоки, списки **не имеют ограничений на длину**, их размеры не зависят от формата машинных слов и границ блоков памяти..	Как в языке Lisp., но с добавлением аппаратно-представимых чисел, имеющих ограничения на размер ради эффективного применения машинных команд. Другие данные не имеют таких ограничений.
9. Списки свойств атома	Атомы, смысл которых не ограничен их именем, сопровождаются списком свойств, часть которых определены в системе программирования, другие **могут программироваться.** Разница между переменными и разными категориями функций представляется в списке свойств атомов. При пополнении списка свойств ранее существовавшие свойства не исчезают, хотя по умолчанию используется **самое новое определение.**	Как в языке Lisp для программируемых свойств. Реализована деструктивная работа с системными свойствами.
10. **Неизменяемость данных**	Обработка списков подчинена принципу неизменяемости данных. Преобразованные списки строятся в новой памяти **без искажения исходных данных**.	Как в языке Lisp, но приоритет неизменяемости данных сопровождается деструктивными аналогами обременительных функций ради программирования более эффективной обработки.
11. *Функции*	Любое понятие программирования можно рассматривать как функцию или как применение функции. Определения функций допускают **самоопределение (рекурсия)** и подчинены принципам равноправия аргументов и единственности результата.	В дополнение к пониманию функций как в языке Lisp, введены мультизначения, не нарушающие принцип единственного результата, и пакеты, обладающие своими пространствами имён.
12. **Равноправие параметров функции**	Аргументы функции **не зависят друг от друга**. Даже если при реализации языка порядок их вычисления определён, результат функции не зависит от этого порядка.	Как в языке Lisp.
13. *Конструкторы,*	Можно без предварительного	Make- создает экземпляры значений

макросы, специальные функции	вычисления аргументов функции производить обработку их представлений внутри функции — **специальные функции**. Это позволяет программировать разные схемы управления вычислениями, конструировать представления выражений и разных категорий функций, преобразовывать программы, например, **компилировать** или оптимизировать.	конкретного типа данных. Функции defmacro, defmethod и т. д. поддерживают макротехнику.
14. **Единственность результата функции**	При вызове функции в одном и том же контексте, при одних и тех же аргументах получается **один и тот же результат.**	Как в языке Lisp.
15. Выражения (формы)	Программы и выражения представляются как символьные формы — это список, первый элемент которого содержит **представление** функции, а остальные — **представления** аргументов. Пустой список () или атом Nil считаются константной формой, всегда имеющей значение Nil.	Как в языке Lisp, с добавлением префикса # для представления функциональных значений параметров, что означает вызов функции function.
16. **Рекурсия**	Именованные формы выражений допускают **само-применимость определений.**	Как в языке Lisp.
17. Гомоиконность	Представление программы в виде списков позволяет чётко видеть структуры данных, представляющие программу. Такое преставление программы фактически является **абстрактным синтаксическим деревом.**	Как в языке Lisp.
18. Pure Lisp	cons, car, cdr, eq, atom, lambda, label, cond, quote, eval.	Как в языке Lisp, но eval без явного ассоциативного списка.
19. Алгебраические системы функций	Базовые функции над конкретными типами значений группируются в комплекты, обладающие **полнотой операций** и/или отношений, определены обратные функции и минимальные значения, выполняющие роль нулей или единиц.	Как в языке Lisp.
20. Аксиоматика отношений	Поддерживаются аксиомы, позволяющие в отдельных случаях получать результаты функций **без выполнения вычислений**.	Как в языке Lisp.
	Семантический базис	
21. **Quote**	Любую форму можно заблокировать от вычисления, сделав из неё константу.	Как в языке Lisp.
22. **Eval**	Для любой представленной или построенной в программе формы можно вычислять её значение по мере необходимости, в том числе, вычислять ранее заблокированные или сконструированные формы.	Как в языке Lisp с добавлением функции **funcall** из-за разницы в пространствах имён для функций и переменных
23. **Lambda** - *свободные и связанные переменные*	Lambda при **объявлении безымянных функций** выделяет **список связанных переменных**, они получают значение при вызове функции. Определяющая форма функции может содержать и другие переменные, называемые **свободными**, они получают значение из внешнего контекста.	В Lambda-определениях позиционные параметры дополнены возможностью указывать иные виды параметров — факультативные, ключевые и серийные. . Некоторые аргументы могут иметь значения по умолчанию.

24. *Пространства имён*	Поддержано два пространства имён — раздельно для локальных и глобальных определений с **приоритетом локальных**. Имена переменных и функций расположены **в общем пространстве.**	Добавлена статическая, лексическая область видимости переменных[69], реализуемая как неявный progN, и ряд форм для объявления локальных пространств имён и пакетов.
25. Контекст — иерархия областей видимости, динамика и статика	Если одна функция вызывает другую, то внутренняя через свободные переменные может использовать **значения переменных из объемлющих функций**.	Добавлена статическая, лексическая областЬ видимости переменных без отмены динамической.
26. ***Label*** *- приоритет локальным определениям*	Локальные определения освобождают от придумывания уникальных имён. Теоретически глобальные определения можно реализовывать как самые внешние локальные.	Имеется возможность формировать локальные пространства имён с помощью Let, Let*, Flet, Labels для простых и взаимосвязанных переменных, подпрограмм и рекурсивных фукнций.
27. **Define/Defun –** глобальные определения	Кроме создания локальных безымянных и именованных функций имеется **практичный компромисс** в виде глобальных именованных функций (Define/Defun).	Как и в языке Lisp, имеется возможность объявлять глобальные определения с помощью **Defun**.
28. Мульти-функции	Некоторые функции могут быть определены для работы с произвольным числом параметров.	Как в языке Lisp с добавлением мультизначений.
29. *Полиморфизм*	Атом может одновременно обладать разными свойствами, возможно **неоднократное вхождение** любого свойства. Атом может представлять и значение, и несколько разных категорий определения функции, и ещё что-нибудь по усмотрению программиста. Функция может иметь ряд различных определений, включая одновременное вхождение символьных и кодовых определений, например, результатов компиляции. По умолчанию работает самое новое определение, но и **к прежним определениям доступ возможен.**	Возможности полиморфизма ограничены для системных свойств, т к. они поддерживаются без сохранения прежних значений при изменениях.
30. Функции высших порядков	Представления аргументов функции могут быть **символьными формами определения** или представления функции. Поэтому можно определять функции высших порядков, использующие функциональные переменные. Аргументами и/или результатами функции высших порядков могут быть представления других функций.	Как в языке Lisp/
31. Отображения, свёртки и фильтры	Применение функций, передаваемых через параметры, требует согласования не только типов данных или значений, но и рангов функций и контекста их вызова, что можно реализовать конструированием **рецепта применения функциональной переменной**.	Как и в языке Lisp имеется возможность программировать отображения и фильтры. Система включает стандартные отображения map, mapcon, mapcar, map-into и другие, часть которых деструктивны..
32. *Функциональные переменные*	Представления функций — не более чем символьные формы, поэтому нет никаких препятствий существованию функциональных переменных, значением	Как и в языке Lisp, но с использованием функции funcall при необходимости из пространства имён функций переходить в пространство имён переменных.

69 В середине 1960-ых появился диалект Lispkit — реализация чисто функционального подмножества Pure Lisp с лексической областью видимости.

	имеющих представление функции, и передаче представлений функций в качестве параметров при вызове функций или **выдаче их как результат.**	
33. Безымянные определения	Передаваемые через параметры функции часто имеют **разовый характер**, а именование функций осмысленно для многократного использования. Безымянные функции строит конструктор Lambda.	Как и в языке Lisp имеется возможность формировать безымянные функции (lambda), дополненная конструктором lambda-case, поддерживающим создание полиморфных определений функции, отличающихся форматом списка параметров.
34. **Cond - If** ветвление	В Pure Lisp хотя бы одна ветвь должна быть выбрана. В Lisp 1.5 в форме Prog это не обязательно, при отсутствии истинного предиката значением ветвления является Nil.	К универсальному Cond как в языке Lisp добавлен обычный If.
35. **Eq** – сравнение указателей или атомов	Существуют полный комплект предикатов для **контроля процесса выполнения** программы.	Как в языке Lisp.
36. **Atom**, Null, Number - контроль типов значений	**Каждый тип данных имеет предикат**, позволяющий в любой момент интерпретации программы, выяснять принадлежность значений нужному типу данных.	Как в языке Lisp. Кроме того, есть общая функция typep - она выясняет тип данных.
37. Объявление типов данных	**Для компиляции функций следует заранее объявлять тип данных**, передаваемых через свободные переменные (declare).	.Как в языке Lisp.
38. Предикаты	**Любая функция** может выполнять роль предиката. Ложным значением считается значение (), оно же Nil, остальное — истина.	Как в языке Lisp.
39. **Логика -** ( ) Nil, значение «ложь» -	Nil выполняет роль значения «ложь», любое другое значение работает как «истина». Поиск в списках при неудаче **завершается на пустом списке**. Эффективно пустой список рассматривать как значение «ложь», подобно коду «0» во многих языках.	Благодаря неявной лексической области (неявный PROGN) можно не требовать истинного предиката в ветвлениях, что дает лаконичное представление COND, выполняющих перебор списка.
40. Компиляция - **compile**	Компиляция функций рассматривается как **оптимизация программ**, позволяющая повысить скорость их выполнения примерно в 50 раз ценой существенного расхода времени на обработку программы. Это достаточное основание подвергать компиляции достаточно отлаженные функции, чтобы избежать накладных расходов на компиляцию программ с неотлаженными функциями.	Компиляция отлаженных функций, в отличие от компиляции программ, позволяет избегать накладных расходов на компиляцию не созревших для компиляции фрагментов.
	Диагностика и отладка	
41. **Error** – диагностика ошибок	Функция Error - обработчик ошибок, сообщающий диагноз дефекта в определении выражений и позволяющий **программировать продолжение** вычислений.	Как в языке Lisp.
42. **Trace/Untrace** – отладка	Режим **REPL** (Read-Eval-Print Loop) Поддержана возможность смотреть **аргументы-результаты отлаживаемых функций** по ходу их вычисления, список	Как в языке Lisp есть Trace/Untrace, кроме того, для отладка имеются try/catch для программирования прерываний по ситуациям и отладочный режим step, позволяющий

	которых может быть при необходимости изменён. **Trace** – принимает список функций, значения аргументов и результаты которых следует выдавать при каждом вызове. Для этого в списки свойств функций размещается флаг. **Untrace** – удаляет флаг из списка свойств функций, отслеживание которых уже не нужно.	наблюдать за ходом вычислений пошаговым образом с возможностью смотреть стек, перемещаться по уровням и вносить изменения в обстановку. Есть функции apropos обзора состава системы и dribble для вывода протоколов работы.
	Расширения и внешний мир	
43. **Открытость**	Программа имеет доступ ко всем функциям системы программирования (интерпретатора, компилятора, отладчика) и может им давать свои определения. Всё, что понадобилось системе, может быть полезным и программисту. **Изменять можно любые определения, кроме Nil.**	Как в языке Lisp, причём, возможны вставки кода и выходы на уровень ОС. Можно фиксировать автономные, модифицированные, проблемно-ориентированные версии системы, настраивать синтаксис, диагностические сообщения и алфавит.
44. **Print, Read** – ввод-вывод **I/O**	Print формально является **тождественной функцией вывода** данных, он печатает свой аргумент на внешнем носителе. Ввод принимает данное с внешнего устройства, строит эквивалентную ему символьную форму, становящуюся его результатом, что удобно при переходе от обработки констант к обработке произвольных данных.	Как в языке Lisp с добавлением работы с файлами и взаимодействие с ОС.
	Прагматика реализации	
45. **GC** – «Сборка мусора», **гибкость распределения памяти**	Автоматизировано **повторное использование памяти**, чтобы освободить специалиста для более существенных задач. Возможен **вызов GC из программы**.	Как в языке Lisp.
46. **Структуры данных**	Хотя на уровне языка нет ни векторов, ни хэш-таблиц, в реализации поддержана работа с **векторами для взаимодействия с машинным кодом** процедур и с **хэш-таблицами для идентификации** атомов. Кроме того, в языке поддержаны две функциональные модели хэш-таблиц — **глобальный список свойств атома и локальный ассоциативный список**, локально связывающий атомы с их определениями, позволяющие не нарушать принцип неизменяемости данных..	Введены на уровне семантики явные символьные модели для векторов и хэш-функций, без изменения синтаксиса. Добавлены конструкторы с префиксом make-.для создания разных типов значений.
47. **Присваивания** и адреса	Присваивания поддержаны в двух формах. Это псевдо-функции **Rplaca и Rplacd,** изменяющие значение по указанному адресу, и функции **Set и Setq внутри формы Prog**, выполняющие изменение значений рабочих переменных без побочного эффекта на внешнем уровне, что позволяет Prog рассматривать как **чистую функцию**, совмещающую декларативный ответ на вопрос «Что?» с императивным ответом на вопрос «Как?».	Как в языке Lisp с некоторым расширением области действия неявной формы ProgN на глобальный уровень, разрешающий использование присваиваний — это делает глобальной статическую область видимости.

48. **Замыкание функции**	Для передачи функций со свободными переменными в качестве функциональных параметров интерпретатор выполняет **формирование рецептов**, однозначно связывающих переменные **в точке вызова функции с её контекстом**, а не в точках её вычисления, в которых пространства имён могут различаться.	Как в языке Lisp.
49. **Устройства** ввода-вывода данных и ОС	Имеются функции, поддерживающие доступ к периферийным устройствам и операционной системе.	Как в языке Lisp.
50. **Раскрутка** компилятора и интерпретатора	Интерпретатор и компилятор для языка Lisp первоначально были определены на языке Lisp, а затем воспроизведены в машинном коде.	Системы реализована на Си и самом Common Lisp.
51. **Альтернативность** решений, полное пространство для мысли: «Если нельзя, но очень хочется, то можно».	• Реализация вычислений с помощью **обычных функций**, выполняемых после вычисления параметров, дополнена **специальными функциями**, вместо значений параметров, обрабатывающих представления аргументов. • Кроме организации **правильных вычислений** поддерживается представление обработки **ошибочных ситуаций** и продолжения вычислений. • Пространство **абстрактных атомов** расширено конкретными **типами данных**, принятыми в основных областях приложения — псевдо-атомы. • **Автоматизация** «сборки мусора» допускает **вызов GC из программы**. • Кроме интерпретации выражений (**eval**) поддержана компиляция функций (**compile**).	Как в языке Lisp.
52. Изобразительные средства **( ) . ' ;** #	Бедный синтаксис освобождает от необходимости размышлять на тему, «где поставить запятую, где не ставить запятой». **( )** - границы точечной пары или списка, **.** - разделитель элементов пары, из которых строятся все структуры данных, **'** - кавычка — префикс, объявляющий представление константы, 'x эквивалент (Quote x). **;** - объявление начала строчного комментария.	Как в языке Lisp с добавлением префикса # для вызова функциональных констант, эквивалент функции Function.. Поддержана возможность использования кириллицы, настройки лексики, синтаксиса и диагностических сообщений. **:** - используется в ключевых словах.

Таблица 3 показывает, что принципиальное отличие диалекта CLISP от языка Lisp выражается в появлении раздельных пространств имён для переменных и функций, а также для других категорий данных. Это удобно для реализации проблемно-ориентированных пакетов. В результате несколько изменяется синтаксис работы с функциональными параметрами, а именно, появляется префикс # для представления функциональных значений параметров, что означает вызов функции function, и функция funcall для вызова функции со списком аргументов.

Кроме того, добавлена статическая, лексическая область видимости переменных[70], реализуемая как глобальный неявный progN, и булев тип данных (T, Nil) без отмены роли

70 В середине 1960-ых появился диалект Lispkit — реализация чисто функционального подмножества Pure Lisp с лексической областью видимости.

Nil и пустого списка () в качестве значения «ложь». Поддержана работа с многомерными векторами, хэш-таблицами и можно строить свои типы данных, Атом со списком свойств называется «символ», системные свойства атомов обрабатываются деструктивно, для этого введено неявное понятие «поле», поэтому системные свойства не могут быть полиморфными. Кроме данных произвольных размеров поддержаны аппаратно-представимые числа. Имеется функция typep – она выясняет тип данных. Добавлен обычный If. Можно формировать локальные пространства имён с помощью Let, Let*, Flet, Labels раздельно для независимых и взаимосвязанных переменных, простых и рекурсивных функций. Ведено понятие «мультизначение». Параметры функций могут быть позиционными, факультативными, ключевыми и серийными, некоторые могут иметь значения по умолчанию. Имеются деструктивные аналоги ресурсозатратных функций. Конструкторы Defmacro и Defmethod поддерживают макротехнику. Доступен отладочный режим Step, позволяющий наблюдать за ходом вычислений пошаговым образом и функции Apropos для обзора состава системы и Dribble для вывода протоколов работы. Поддержана возможность настраивать синтаксис, диагностические сообщения, алфавит, включая использование кириллицы, настройки лексики, синтаксиса и диагностических сообщений. Первые системы CLISP реализованы на Си и самом Common Lisp.

8. Диалект Racket для будущих языков программирования

В 2010 году название Racket получила очередная версия учебной среды программирования на языке Scheme, разрабатываемой с 1995 в Университете Дьюка для обучения созданию, разработке и реализации новых языков программирования [58]. Можно вспомнить, что первые описания языка Logo, иногда называемого «Lisp без скобок», завершались призывом «Сделай свой язык»[71].

Диалект Racket является мультипрадигмальным языком программирования общего назначения, поддерживающим функциональное, языково-ориентированное, рефлективное, логическое, объектно-ориентированное, процедурное и мета-программирование. Racket успешно используется как скриптовый язык, в обучении информатике и в научных исследованиях. Образовательная направленность повлияла на общую структуру языка Racket как системы 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.

8.1. Возможности

Racket поддерживает мощную макросистему для создания предметно-ориентированных языков программирования, использующих языковые конструкции с различной семантикой в отличие от популярных синтаксически управляемых конструкторов анализаторов, что созвучно целям программистского образования, высказанным Р. Флойдом (*Robert W. Floyd*) в его Тьринговской лекции[72]. Реализация Racket стала качественным образовательным продуктом, предназначенным для продвижения начинающих программистов до уровня разработчиков больших систем и исследований в области мягкой типизации - совмещения статической и динамической типизации, характерной для многих диалектов языка Lisp. Производительность обеспечена JIT-компилятором и механизмом сборки мусора с поддержкой поколений объектов. Включена поддержка мелкозернистого параллелизма. Появились версии Minimal Racket без пакетов, поддержка байт-кода и JIT-компиляции для архитектуры

71 Пейперт С. Переворот в сознании: Дети, компьютеры и плодотворные идеи. Москва, Педагогика, 1989.

72 R. W. Floyd. The paradigms of Programming. / R. W. Floyd. //Communications of the ACM 22 (N 8), 1979: p. 455 http://rkka21.ru/docs/turing-award/rf1978r.pdf

ARM, а также быстродействующий Typed Racket и другие. Имеется собственная виртуальная машина. В экспериментах по разработке разных версий диалекта Racket обнаружилось, что компиляция не всегда повышает производительность программ, но иногда способствует продуктивности программирования. Необходимость ждать результат компиляции даёт программисту защищённое время подумать о задаче и программе. Был провозглашён манифест языка Racket, детально описывающий принципы, лежащие в основе его разработки, описывающий основу вычислений, стоящих за процессом проектирования программ, и раскрывающий возможности для будущего их улучшения[73].

Реализация языка представлена как комплект модулей и диалектов, допускает подключение иноязычных библиотек и улучшение системы используемых классов. Кроме влияния Scheme (macros, modules, lexical closures, tail calls, continuations) на Racket повлияли идеи объектно-ориентированного языка Eiffel[74] (software contracts), современные потребности в организации параллельных вычислений (green threads, OS threads) и многое другое. Racket способен работать как ОС и управлять другими программами.

8.2. Компромисс Lisp-1 и Lisp-2

Возможности расширения Racket четко отличают его от других языков семейства Lisp-1, приближают его к языкам семейства Lisp-2. Это интегрированные возможности расширения языка, поддерживающие создание новых предметно-ориентированных языков и диалектов. Они встраиваются в систему модулей, что позволяет осуществлять контекстно-зависимый и модульный контроль над синтаксисом, а также переопределять семантику применения функции. Из особенностей Lisp-1 остаётся отсутствие программируемых свойств символа и наличие булевых значений #f, #t.

Любой модуль можно использовать как язык, посредством спецификатора #lang, это фактически означает, что практически любой аспект языка может быть запрограммирован как отдельный диалект.

Во многих языках макросистема — это не более чем тщательно настроенный интерфейс прикладного программирования (API) для расширений синтаксиса компилятора. Используя такой интерфейс, программисты могут добавлять новые возможности и целые предметно-ориентированные языки так, что они становятся внешне совершенно неотличимы от встроенных языковых конструкций, но без принципиального развития семантики языка. Система макросов в Racket используется для создания полных языковых диалектов, затрагивая семантику. В их числе - Typed Racket, который является диалектом Racket с постепенной типизацией, облегчающей переход от нетипизированного к типизированному коду программ, Lazy Racket - диалект с ленивыми вычислениями, а также Hackett, который объединяет Haskell и Racket. Педагогический язык программирования Pyret изначально был реализован в Racket. Известны и другие языки, полученные с помощью Racket.

Редактор системы DrRacket обеспечивает подсветку синтаксиса и ошибок времени выполнения, сопоставление скобок, отладчик и алгебраический пошаговый вычислитель. Кроме того, Racket имеет встроенную поддержку библиотек и сложные инструменты анализа для опытных программистов. Поддерживается модульное программирование с браузером модулей, контурным обзором, тестированием и измерениями кода, а также поддержкой рефакторинга. Racket предоставляет интегрированный, контекстно-зависимый доступ к обширной гиперссылочной "Службе поддержки" и включает диалект поддерживающий разработку документации. Уместно вспомнить разработанную Д. Кнутом технологию грамотного программирования (*literate programming*)[75], по которой

---

73 *Felleisen, M.; Findler, R.B.; Flatt, M.; Krishnamurthi, S.; Barzilay, E.; McCarthy, J.; Tobin-Hochstadt, S. (2015). "The Racket Manifesto" (PDF). Proceedings of the First Summit on Advances in Programming Languages: 113–128.*

74 Бертран Мейер Основы ООП https://intuit.ru/studies/courses/71/71/info

75 http://webplanet.ru/interview/soft/2008/05/06/knuth.html — интервью с Д.Кнутом.

процесс разработки и отладки программ программист должен совмещать с созданием и уточнением документации.

Дистрибутив языка Racket включает в себя обширную библиотеку, охватывающую системное и сетевое программирование, веб-разработку, интерфейс к операционной системе и для работы с внешними устройствами, несколько разновидностей регулярных выражений, генераторы лексеров/парсеров, логическое программирование и полноценный фреймворк для графического интерфейса. При конструировании языков программирования можно использовать отображения образцов, подобные грамматикам с переводом, что позволяет делать диалекты не только на Racket, его байт-коде и JVM, но и на других языках. Для организации параллельных вычислений поддержаны разные модели параллелизма, включая средства синхронизации процессов.

8.3. Учебные диалекты

Для студентов включена поддержка ряда "уровней языка" (Начинающий студент, Продвинутый студент и т.д.). Уровень начинающего студента выделен в диалект Minimal Racket. Он включает в себя базовый синтаксис (#lang[76] racket), выражения, создание и обработку списков (cons, car, cdr, list, length, null?), вызов функций, квотирование (') и комментарии, базовые типы данных (числа, строки, символы, списки, #t, #f) и операции (+, -, *, /, quotient, remainde, =, <, >, <=, >=, and, or, not), определения рекурсивных функций с параметрами и возвращаемыми значениями (define, lambda), ветвления (if, cond), области видимости и модули (scope, require, локальные и глобальные переменные), ввод/вывод (display, newline, read) и основы обработки ошибок (error, check-expect), что примерно соответствует возможностям Pure Lisp без EVAL. Добавление к Minimal Racket доступной в языке Racket функции eval достаточно для обучения мета-программированию.

Примеры:

```
#lang racket
(eval (eval '(list 'car  '(a b c)) ))
(define (square x) (* x x))
(if (> x 0) "positive" "non-positive")
(define expr '(list 'car '(a b c)))
(eval (eval expr))
```

Переход к уровню продвинутого студента нацелен на базовые навыки работы с функциями высокого порядка, замыканиями, созданием собственных типов данных, определением образцов и отображений, компиляции, а также введением новых синтаксических конструкций, использованием наследования и полиморфизма, обработкой хэш-таблиц, деревьев, экспериментами с асинхронным программированием, параллельными вычислениям, много-поточностью и асинхронными процессами. После этого формируются приёмы работы с изменяемыми данными (mutable data: set-car!, set-cdr!, vector-set!, hash-set!, box, unbox) и побочными эффектами, изучаются такие механизмы как монады (monads), модули (modules), импорт и экспорт (Import, export), работа с файлами (File I/O: open-input-file, open-output-file, read, write, close-input-port, close-output-port) и с сетью (Networking - net/tcp). Применяется графический интерфейс пользователя (GUI), методы отладки (Debugging) и тестирование (Testing: rackunit). Исследуются проблемы производительности и оптимизации программ, включая хвостовую рекурсию (Tail recursion) и профилирование (Profiling: open-input-file, close-input-port, read-line, write-string), взаимодействие с командной строкой и аргументами командной строки (check-expect, check-assert). Программа «hello world» будет выглядеть следующим образом:

```
#lang racket
```

76 Спецификатор #lang указывает наименование используемого языка или диалекта.

"Hello, World!"

Пример факториала на Typed Racket, диалекте со статической типизацией:

```
#lang typed/racket
    (: fact (Integer -> Integer))
    (define (fact n)
          (cond [(zero? n) 1]
                [else (* n (fact (- n 1)))]
    )       )
```

Пример реализации языка, поддерживающего арифметические операции, такие как сложение, вычитание, умножение и деление, может выглядеть следующим образом:

```
#lang racket
;; Определение грамматики языка
(define-syntax-rule (define-language)
(begin
(define-syntax-class expr
   (pattern (+ expr expr)
               #:with type 'add)
   (pattern (- expr expr)
                #:with type 'sub)
   (pattern (* expr expr)
                #:with type 'mul)
    (pattern (/ expr expr)
                #:with type 'dif)
(pattern (number:real))
(pattern (id:identifier))
(pattern ((expr)))
(pattern (- expr)
           #:with type 'neg)

) ))
```

В этом примере можно проследить аналогию представленных в программе конструкций с мета-лингвистическими формами типа БНФ, что позволяет такие программы получать методами синтаксически управляемого конструирования. Определение синтаксиса языка можно рассматривать как представление сложного типа данных. Получается, что диалект Racket отвечает на вопрос «**КТО** будет определять лицо программирования в будущем?». Ответ формируется следующими решениями:

- Универсальность символьных вычислений распространена на архитектуру языка, рассматриваемого как расширяемый комплект диалектов (#lang) согласно манифесту языка Racket, а также полноте приобретения профессиональных навыков, включающих разработку документации, что соответствует предложенной Д. Кнутом концепции грамотного программирования (*literate programming*)[77].

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

- Равноправие параметров поддержано при обучении мета-программированию включением механизмов сопоставления с образцами, достаточными для представления грамматик с переводом (БНФ), а также привлечением графического интерфейса.

77 http://www.literateprogramming.com/ - о грамотном программировании.

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

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

- Неизменяемость данных на начальных этапах обучения постепенно сменяется навыками применения изменяемых данных, использования побочных эффектов и типизации данных.

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

Более детально сходства и различия языков Lisp и Racket показаны в таблице 5. Следует обращать внимание, что в конкретных диалектах решения Racket могут отличаться.

8.4. Выводы

Можно отметить, что диалект Racket примерно на треть наследует решения языка Scheme и на две трети возвращается к решениям языка Lisp. Основные различия сводятся к общей организации языка в виде комплекта диалектов, семантика которых обладает некоторыми различиями, отражающими разнообразие целей обучения программированию, пошагового продвижение к решению вопросов разработки новых языков программирования и мета-программирования. Совмещение компиляции с чтением программы допускает восстановление исходной структуры программы. Поддержаны динамические области видимости. Можно программировать свои свойства символа, но без принципа неизменяемости данных. Изменена граница между неявными и явными представлениями структур данных, можно представлять на уровне синтаксиса константные вектора и хэш-таблицы. Введены специальные значения для характеристики типичных ситуаций в динамике вычислений. Расширен комплект средств отладки программ и подготовки комплектов поставки программного продукта. Сформированы диалекты для разного уровня квалификации пользователей. 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.

Таблица 5

Сравнение особенностей Lisp и Racket

Характеристика языка	*Обоснования и решения Lisp*	*Обоснования и решения* Racket
	Принципы и выбор решений	***Уточнения и изменения***
1. ***Универсальность***	Любую информацию **для компьютерной обработки** можно представить в символьной форме.	Как в Scheme, на базе которого он создан, информационная обработка ограничена процессами обработки чисел или строк и структур над ними.
2. ***()***	На терминале не было других скобок. Зато и теперь они набираются на любом регистре, в отличие от фигурных и квадратных	Кроме круглых скобок используются квадратные и фигурные.
3. *Символьные формы*	Символьные формы бывают **неделимыми**, если в языке не предусмотрен разбор их на части, или **составными**, если язык допускает их сборку из частей и разделение на части.	Имеются неделимые числа, строки, литеры, булевы и специальные значения, атомы и символы как в языке Lisp. Составные формы (списки и пары) дополнены представлением и реализацией векторов, множеств, хэш-таблиц, структур, классов и

		объектов, функций и потоков. Кроме того, существуют порты-потоки (Ports), обещания (Promises), продолжения (Continuations), регулярные выражения (regular expressions), дата и время (date/time)
4. Атомы	Символьные формы для представления неделимых данных называются атомами. Любую сущность, включая ключевые слова, можно представить как атом, именовать в соответствии с особенностями области приложения. Атомы выполняют роль **идентификаторов** для значений любой природы.	Как в языке Lisp атомы и символы, числа, строки, литеры. Кроме того есть булевы и специальные значения, такие как #<void>.
5. *Константы*	Введены так называемые псевдо-атомы, смысл которых виден по форме их записи, - это **числа и строки**. Строки могут разделяться на символы, но по соглашению они считаются псевдо-атомами. Составные формы строятся из пар элементов произвольной природы, разделённых точкой и заключенных в скобки. **Список строится из пар** и завершается парой, содержащей последний элемент и пустой список.	Числа, строки, литеры, булевы, специальные значения, комплексные числа (Complex numbers). Racket различает точные (exact) и неточные (inexact) числа.
6. Списки, иерархия форм	Основная структура составных данных — список, перечисленные элементы которого заключёны в скобки и при необходимости разделяются пробелами. Нет загромождения запятыми или другими разделителями. Э**лементы списка могут быть любой природы и разного вида**.	Как в языке Lisp.
7. Пустой список **Nil = ()**	Пустой список не может быть разделён на части средствами языка, поэтому он является **одновременно списком и атомом**. Его можно считать отдельным типом значений.	Как в языке Lisp, Nil и пустой список () - отдельный тип данных, используются в качестве расширения значения «ложь».
8. **Гибкость распределения памяти**	Имена атомов, числа, стоки, списки **не имеют ограничений на длину**, их размеры не зависят от формата машинных слов и границ блоков памяти..	Как в языке Lisp ограничений нет с добавлением функции memory-consumed, дающей сведения о потреблении памяти для различных объектов.
9. Списки свойств атома	Атомы, смысл которых не ограничен их именем, сопровождаются списком свойств, часть которых определены в системе программирования, другие **могут программироваться.** Разница между переменными и разными категориями функций представляется в списке свойств атомов. При пополнении списка свойств ранее существовавшие свойства не исчезают, хотя по умолчанию используется **самое новое определение.**	Символы обладают списками свойств для хранения информации о идентификаторах переменных и функций. Можно программировать свои свойства символа и восстанавливать их после изменений.
10. ***Неизменяемость данных***	Обработка списков подчинена принципу неизменяемости данных. Преобразованные списки строятся в новой памяти **без искажения исходных данных**.	Изменяемы хеш-таблицы (hash tables), ячейки (Boxes), векторы (Vectors), структуры с изменяемыми полями и выделена категория функций мутаторы (Mutators). mcons, mcar, mcdr, set-mcar!, set-mcdr! vector, vector-ref, vector-set!, box, unbox, set-box!
11. *Функции*	Любое понятие программирования	При необходимости многие результаты

	можно рассматривать как функцию или как применение функции. Определения функций допускают **самоопределение (рекурсия)** и подчинены принципам равноправия аргументов и единственности результата.	упаковываются в список, отсутствие результата для процедур - #<void>.
12. **Равноправие параметров функции**	Аргументы функции **не зависят друг от друга**. Даже если при реализации языка порядок их вычисления определён, результат функции не зависит от этого порядка.	Как в языке Lisp. если не учитывать разницу между связанными и свободными переменными
13. *Конструкторы, макросы, специальные функции*	Можно без предварительного вычисления аргументов функции производить обработку их представлений внутри функции — **специальные функции**. Это позволяет программировать разные схемы управления вычислениями, конструировать представления выражений и разных категорий функций, преобразовывать программы, например, **компилировать** или оптимизировать.	Существует макротехника с профилактикой коллизий имён. (define-syntax, let-syntax, letrec-syntax, syntax-rules, syntax-case, quasiquote/unquote). Макрос заменяет исходный код в процессе компиляции. racket/match, parser-tools/lex, parser-tools/yacc
14. **Единственность результата функции**	При вызове функции в одном и том же контексте, при одних и тех же аргументах получается **один и тот же результат.**	Как в языке Lisp.
15. *Выражения (формы)*	Программы и выражения представляются как символьные формы — это список, первый элемент которого содержит **представление** функции, а остальные — **представления** аргументов. Пустой список () или атом Nil считаются константной формой, всегда имеющей значение Nil.	Как и в языке Lisp символьная форма — это список, первый элемент которого содержит представление функции, а остальные — представления аргументов. Допустима инфиксная форма для арифметических выражений.
16. **Рекурсия**	Именованные формы выражений допускают **само-применимость определений.**	Как в языке Lisp.
17. Гомоиконность	Представление программы в виде списков позволяет чётко видеть структуры данных, представляющие программу. Такое преставление программы фактически является **абстрактным синтаксическим деревом.**	Как в языке Lisp, но требует средств восстановления символьной формы после компиляции.
18. Pure Lisp	cons, car, cdr, eq, atom, lambda, label, cond, quote, eval.	Minimal Racket
19. *Алгебраические системы функций*	Базовые функции над конкретными типами значений группируются в комплекты, обладающие **полнотой операций** и/или отношений, определены обратные функции и минимальные значения, выполняющие роль нулей или единиц.	Элементы одной семантической системы входят в разные реализационные модули уровня прагматики. Eval не водит в базис, хотя признаётся ключевой.
20. Аксиоматика отношений	Поддерживаются аксиомы, позволяющие в отдельных случаях получать результаты функций **без выполнения вычислений**.	Возможно такие отношения учитываются при оптимизирующей компиляции. (Atom? (Cons x y)) = #f
	Семантический базис	
21. **Quote**	Любую форму можно заблокировать от вычисления, сделав из неё константу.	Как и в языке Lisp Quote работает для создания констант. *Как и в языке Scheme для объявления*

		ленивых вычислений есть специальная функция delay, а для их возобновления — force.
22. **Eval**	Для любой представленной или построенной в программе формы можно вычислять её значение по мере необходимости, в том числе, вычислять ранее заблокированные или сконструированные формы.	Как и в языке Lisp eval является одной из ключевых функций, позволяет интерпретировать и выполнять произвольные выражения во время выполнения программы. Предостерегают, использование eval требует осторожности.
23. **Lambda** - свободные и связанные переменные	Lambda при **объявлении безымянных функций** выделяет **список связанных переменных**, они получают значение при вызове функции. Определяющая форма функции может содержать и другие переменные, называемые **свободными**, они получают значение из внешнего контекста.	Как и в языке Lisp имеется возможность формировать безымянные функции (lambda).
24. Пространства имён	Поддержано два пространства имён — раздельно для локальных и глобальных определений с **приоритетом локальных**. Имена переменных и функций расположены **в общем пространстве.**	Как и в языке Lisp – динамика. Кроме того, каждый модуль, библиотека и др. имеет свое собственное пространство имён как пакеты в языке Lisp (require, provide, define-namespace-anchor, namespace-anchor->namespace)
25. Контекст — иерархия областей видимости, динамика и статика	Если одна функция вызывает другую, то внутренняя через свободные переменные может использовать **значения переменных из объемлющих функций**.	Добавлена статическая, лексическая область видимости переменных без отмены динамической и ряд форм для объявления локальных пространств имён и пакетов.
26. ***Label*** - *приоритет локальным определениям*	Локальные определения освобождают от придумывания уникальных имён. Теоретически глобальные определения можно реализовывать как самые внешние локальные.	Имеется возможность формировать локальные пространства имён с помощью Let и т.п.
27. **Define/Defun –** глобальные определения	Кроме создания локальных безымянных и именованных функций имеется **практичный компромисс** в виде глобальных именованных функций (Define/Defun).	Как и в языке Lisp приоритет локальным переменным (Lambda, Define)
28. Мульти-функции	Некоторые функции могут быть определены для работы с произвольным числом параметров.	Как и в языке Lisp поддержаны мультизначения и мультифункции, но с более удобным синтаксисом. (define (sum-all **.** Numbers) (apply + numbers))
29. *Полиморфизм*	Атом может одновременно обладать разными свойствами, возможно **неоднократное вхождение** любого свойства. Атом может представлять и значение, и несколько разных категорий определения функции, и ещё что-нибудь по усмотрению программиста. Функция может иметь ряд различных определений, включая одновременное вхождение символьных и кодовых определений, например, результатов компиляции. По умолчанию работает самое новое определение, но и **к прежним определениям доступ возможен.**	Только однократное вхождение свойства. Нарушает принцип неизменяемости.
30. Функции высших	Представления аргументов функции	Как и в языке Lisp поддержаны функции высших

порядков	могут быть **символьными формами определения** или представления функции. Поэтому можно определять функции высших порядков, использующие функциональные переменные. Аргументами и/или результатами функции высших порядков могут быть представления других функций.	порядков.
31. Отображения, свёртки и фильтры	Применение функций, передаваемых через параметры, требует согласования не только типов данных или значений, но и рангов функций и контекста их вызова, что можно реализовать конструированием **рецепта применения функциональной переменной**.	Как и в языке Lisp имеются встроенные отображения и фильтры и возможность их программировать.
32. Функциональные переменные	Представления функций — не более чем символьные формы, поэтому нет никаких препятствий существованию функциональных переменных, значением имеющих представление функции, и передаче представлений функций в качестве параметров при вызове функций или **выдаче их как результат.**	Как и в языке Lisp поддержаны функциональные переменные с возможностью их передачи в качестве результата, но без конструирования из элементов, хотя это не исключено.
33. Безымянные определения	Передаваемые через параметры функции часто имеют **разовый характер**, а именование функций осмысленно для многократного использования. Безымянные функции строит конструктор Lambda.	Как и в языке Lisp имеется возможность формировать безымянные функции (lambda), дополненная конструктором lambda-case, поддерживающим создание полиморфных определений функции, отличающихся форматом списка параметров.
34. ***Cond - If*** *ветвление*	В Pure Lisp хотя бы одна ветвь должна быть выбрана. В Lisp 1.5 в форме Prog это не обязательно, при отсутствии истинного предиката значением ветвления является Nil.	Ветвление должно содержать истинный предикат, иначе вырабатывается значение #undef. Имеется предикат else – тождественная истина. Иначе результат #<undefined>. Это несколько загромождает представление ветвлений необходимостью сравнивать результаты при исчерпании списка
35. ***Eq*** *– сравнение указателей или атомов*	Существуют полный комплект предикатов для **контроля процесса выполнения** программы.	Различается сравнение указателей, значений и структур данных: eq?, eqv?, equal? как в Scheme
36. **Atom**, Null, Number - контроль типов значений	**Каждый тип данных имеет предикат**, позволяющий в любой момент интерпретации программы, выяснять принадлежность значений нужному типу данных.	встроены распознающие предикаты.: Number?, integer? real?rational?complex? string? symbol? list ? pair?null? vector? procedure? boolean? void? eof-object? Типы переменных проверяются во время выполнения программы, как в языке Lisp. Диалект Typed Racket является статически типизированным языком
37. *Объявление типов данных*	**Для компиляции функций следует заранее объявлять тип данных**, передаваемых через свободные переменные (declare).	принудительная компиляция выясняет что может
38. *Предикаты*	**Любая функция** может выполнять роль предиката. Ложным значением считается значение (), оно же Nil, остальное — истина.	Ложным значением считается булево значение #f, остальное — истина. Существует булево значение #t, представляющее истину. Возникает необходимость сравнения с #f.
39. ***Логика -*** *( ) Nil, значение «ложь»*	Nil выполняет роль значения «ложь», любое другое значение работает как «истина».	Введён булев тип данных **(#f, #t)**, что влечёт необходимость сравнивать результаты с логическим значением #f:

	Поиск в списках при неудаче **завершается на пустом списке**. Эффективно пустой список рассматривать как значение «ложь», подобно коду «0» во многих языках.	(if (null? ls) #f…) Но Nil и пустой список () используются в качестве расширения значения «ложь». Истинными считаются любые другие значения.
40. *Компиляция* - ***compile***	Компиляция функций рассматривается как **оптимизация программ**, позволяющая повысить скорость их выполнения примерно в 50 раз ценой существенного расхода времени на обработку программы. Это достаточное основание подвергать компиляции достаточно отлаженные функции, чтобы избежать накладных расходов на компиляцию программ с неотлаженными функциями.	Компиляция программы совмещена с чтением как в Scheme.
	Диагностика и отладка	
41. **Error** – диагностика ошибок	Функция Error - обработчик ошибок, сообщающий диагноз дефекта в определении выражений и позволяющий **программировать продолжение** вычислений.	Поддержана функция реакции на ошибки (error who msg irritant ...) try/except и try-catch
42. **Trace/Untrace** – отладка	Режим **REPL** (Read-Eval-Print Loop) Поддержана возможность смотреть **аргументы-результаты отлаживаемых функций** по ходу их вычисления, список которых может быть при необходимости изменён. **Trace** – принимает список функций, значения аргументов и результаты которых следует выдавать при каждом вызове. Для этого в списки свойств функций размещается флаг. **Untrace** – удаляет флаг из списка свойств функций, отслеживание которых уже не нужно.	В дополненбие к диагностике, отображению стека вызовов, просмотру переменных и пошаговому выполнению имеются функции trace и untrace, printf и displayln, break, DrRacket, Racket Debugger, Lint-подобные инструменты, raco check-syntax и библиотека rackunit. log-info "...", log-debug "..." и log-error "…", а также time ... и profile …, racket-tcp-server и racket-tcp-client.
	Расширения и внешний мир	
43. **Открытость**	Программа имеет доступ ко всем функциям системы программирования (интерпретатора, компилятора, отладчика) и может им давать свои определения. Всё, что понадобилось системе, может быть полезным и программисту. **Изменять можно любые определения, кроме Nil.**	Большое количество библиотек для различных задач, включая графику, сетевое программирование, обработку данных и т. д. system, subprocess, process-wait, process-kill, get-os, get-hostname, get-environment-variable, get-time-of-day
44. ***Print, Read*** – *ввод-вывод* ***I/O***	Print формально является **тождественной функцией вывода** данных, он печатает свой аргумент на внешнем носителе. Ввод принимает данное с внешнего устройства, строит эквивалентную ему символьную форму, становящуюся его результатом, что удобно при переходе от обработки констант к обработке произвольных данных.	Функции open-input-file, open-output-file, read, display, printf, tcp-listen, tcp-accept, tcp-connect. Port-count-bytes, Newline, read-char, read-line, read-string, file-exists?, close-input-port, close-output-port, read-line, Read-string, read-char, write, delete-file и директории. Порт — структурный тип данных поток. Процедура вывода не имеет результата.
	Прагматика реализации	
45. **GC** – «Сборка мусора», **гибкость распределения памяти**	Автоматизировано **повторное использование памяти**, чтобы освободить специалиста для более существенных задач.	Процедура GC из программы выполняет сборку мусора. Функция memory-use (из модуля racket/base) сообщает общее количество памяти,

	Возможен **вызов GC из программы**.	используемой в данный момент в байтах. Create-a-lot-of-objects и Gc-stats — для экспериментов.
46. ***Структуры данных***	Хотя на уровне языка нет ни векторов, ни хэш-таблиц, в реализации поддержана работа с **векторами для взаимодействия с машинным кодом** процедур и с **хэш-таблицами для идентификации** атомов. Кроме того, в языке поддержаны две функциональные модели хэш-таблиц — **глобальный список свойств атома и локальный ассоциативный список**, локально связывающий атомы с их определениями, позволяющие не нарушать принцип неизменяемости данных..	Для векторов введена **явная возможность** вводить их как списки с префиксом #. Вектор из пяти чисел: #(1 2 3 4 5). Хэш-таблицы #hash(1 #t 2 #t 3 #t). #hash((1 . 2) (3 . 4)). Остальные структуры работают неявно, конструируются, к ним поддержан доступ с помощью специальных функций на уровне семантики.
47. ***Присваивания и*** *адреса*	Присваивания поддержаны в двух формах. Это псевдо-функции **Rplaca и Rplacd,** изменяющие значение по указанному адресу, и функции **Set и Setq внутри формы Prog**, выполняющие изменение значений рабочих переменных без побочного эффекта на внешнем уровне, что позволяет Prog рассматривать как **чистую функцию**, совмещающую декларативный ответ на вопрос «Что?» с императивным ответом на вопрос «Как?».	Присваивания стали явными (set! и другие) set! define define-values let Различаются присваивания констант и переменных make-pointer, pointer-ref и pointer-set!, malloc, free и memcpy, bytes-ref и bytes-set!
48. ***Замыкание функции***	Для передачи функций со свободными переменными в качестве функциональных параметров интерпретатор выполняет **формирование рецептов**, однозначно связывающих переменные **в точке вызова функции с её контекстом**, а не в точках её вычисления, в которых пространства имён могут различаться.	Замыкание функций строится автоматически для внутренних функций при их определении, т. е. неявно как в Scheme.
49. ***Устройства*** *ввода-вывода данных и ОС*	Имеются функции, поддерживающие доступ к периферийным устройствам и операционной системе.	Как в языке Lisp.
50. **Раскрутка** компилятора и интерпретатора	Интерпретатор и компилятор для языка Lisp первоначально были определены на языке Lisp, а затем воспроизведены в машинном коде.	Базовый компилятор (Bootstrapping) Racket разработан с учетом самокомпиляции. Racket Compiler (raco compile) - встроенный в Racket компилятор, написанный на Racket Racket Interpreter (racket) - встроенный в Racket интерпретатор, написанный на Racket
51. **Альтернативность** решений, полное пространство для мысли: «Если нельзя, но очень хочется, то можно».	• Реализация вычислений с помощью **обычных функций**, выполняемых после вычисления параметров, дополнена **специальными функциями**, вместо значений параметров, обрабатывающих представления аргументов. • Кроме организации **правильных вычислений** поддерживается представление обработки **ошибочных ситуаций** и продолжения вычислений. • Пространство **абстрактных атомов** расширено конкретными **типами данных**, принятыми в основных	• Процедуры с побочными эффектами" (side-effect procedures). (void) • Реализация вычислений с помощью обычных функций, выполняемых после вычисления параметров, дополнена синтаксической макротехникой и организацией ленивых вычислений.

	областях приложения — псевдо-атомы. • **Автоматизация** «сборки мусора» допускает **вызов GC из программы**. • Кроме интерпретации выражений (**eval**) поддержана компиляция функций (**compile**).	
52. Изобразительные средства **( ) . ' ; # ! : …**	Бедный синтаксис освобождает от необходимости размышлять на тему, «где поставить запятую, где не ставить запятой». **( )** - границы точечной пары или списка, **.** - разделитель элементов пары, из которых строятся все структуры данных, **'** - кавычка — префикс, объявляющий представление константы, **'**x эквивалент (Quote x). **;** - объявление начала строчного комментария.	() **[]** {} **- границы списков для упрощения контроля баланса скобок.** **.** - разделитель элементов пары, из которых строятся все структуры данных, **'** - кавычка — префикс, объявляющий представление константы, 'x эквивалент (Quote x). **;** - объявление начала строчного комментария. **|# #| - блочный комментарий** # - специальный префикс особых случаев. **: - ключевые слова и пространства имен.** **|** - идентификаторы, содержащие пробелы или специальные символы **! - макросы** **… - макросы** Синтаксис обогащён для удобочитаемости

Анализ таблицы 5 показывает, что в диалекте Racket, как и в диалекте Scheme, универсальность ограничена возможностями JVM, неизменяемость данных нарушается функциями, работающими как присваивания (set!), изменяемы хеш-таблицы (hash tables), ячейки (Boxes), векторы (Vectors), структуры с изменяемыми полями (mcons, mcar, mcdr, set-mcar!, set-mcdr! Vector, vector-ref, vector-set!, box, unbox,s et-box!) и выделена категория функций мутаторы (Mutators). Отсутствие значений некоторых функций маскируется значением #<void>, макротехника работает на уровне лексики, допускается лишь однократное вхождение свойства атома, применение функций высших порядков не предусматривает конструирование функциональных значений в динамике, используется булев тип данных (#f, #t), ветвления требуют истинного предиката, иначе формируется значение #undef, имеется константа Else и предикаты eq?, eqv?, equal?. Функция Eval доступна, но не водит в базис языка. Принудительная компиляция выполняется, но допускает восстановление исходного кода. Для объявления ленивых вычислений есть специальная функция delay, а для их возобновления — force.

Диалект Racket дополняет составные формы (списки и пары) возможностью инфиксного представления отдельных категорий формул, представлением векторов, множеств и хэш-таблиц, реализацией на уровне семантики структур, классов и объектов, функций и потоков, существуют порты-потоки (Ports), обещания (Promises), продолжения (Continuations), регулярные выражения (regular expressions), дата и время (date/time). Для реализации такого разнообразия принято во внимание, что атом по существу является указательным типом данных, с которым могут быть связаны разные дисциплины функционирования. Можно программировать свои свойства символа и восстанавливать их после изменений. Символьные формы для представления векторов, множеств и хэш-таблиц переходят с уровня неявной прагматики на уровень явного синтаксиса и семантики (#(1 2 3 4 5) — вектор, #hash(1 #t 2 #t 3 #t), #hash((1 . 2) (3 . 4)) — хэш-таблицы, #{1 2 3} - множество), хотя списочное представление остаётся основным. Порты или потоки рассматриваются как структуры данных, представления числовых констант связываются с указанием точности (exact, inexact), кроме лексической области поддержаны свои пространства имён, имеется конструктор lambda-case для полиморфных определений функций, реализован ряд специализированных диалектов, включая Typed Racket, являющийся статически типизированным языком. Заметное внимание уделено проблеме отладки программ. Кроме try/except и try-catch имеются специальные

диалекты DrRacket, Racket Debugger и другие. Функция memory-consumed даёт сведения о потреблении памяти для различных объектов, memory-use (из модуля racket/base) сообщает общее количество памяти, используемой в данный момент в байтах. Для экспериментов предоставлены Create-a-lot-of-objects и Gc-stats. Диалект Racket активно используется как инструмент самоопределения. Базовый компилятор (Bootstrapping) Racket разработан с учетом самокомпиляции, Racket Compiler (raco compile) - встроенный в Racket компилятор, написанный на Racket, Racket Interpreter (racket) - встроенный в Racket интерпретатор, написанный на Racket.

9. Clojure — новые горизонты

Clojure — современный диалект языка Lisp, появился в 2007 году, разработан Ричем Хикки, независимым разработчиком ПО, ранее разработавшим _dotLisp, в рамках проекта .NET Framework [59-71]. Clojure наследует основные особенности языка Lisp, обеспечивающие гибкое и мощное мета-программирование, поддерживает синтаксическое расширение[78] и надежную семантику, дополненную механизмами для программирования приложений на базе распределённых и параллельных процессов.

Clojure - мультипарадигмальный язык общего назначения с поддержкой разработки в интерактивном режиме, ориентированный на функциональное программирование и упрощающий использование многопоточности (functional, concurrent, macro, agent-oriented, logic, pipeline). Испытал влияние языков C#, C++, **Common Lisp,** Erlang, Haskell, Java, ML, Prolog, **Racket,** Ruby, **Scheme,** Wolfram, повлиял на языки Elixir, Hy, Janet, LFE, Pixie, Rhine.

9.1. Наследство Lisp и его обобщение

Clojure придерживается философии языка Lisp «код как данные»[79] (гомоиконность) и поддерживает развитую систему Lisp-макросов,. Выражения в программе выглядят как списки, в которых первый элемент — представление функции, остальные являются аргументами этой функции. По существу это абстрактное синтаксическое дерево.

Clojure позиционируется как язык семейства Lisp-1[80], он изначально не предназначен для совместимости по коду с диалектами семейства Lisp-2, использует собственный набор структур данных, идентификаторы в Clojure регистрозависимы, символы не имеют списков свойств — их роль выполняют метаданные, некоторые традиционные функции поменяли имена, например, car и cdr заменены на first и rest, часть оставшихся неизменными синтаксических элементов изменили смысл, есть и другие отличия.[81]

Общий ход работы программы определяет цикл REPL (read eval print loop). Вся семантика вычислений сосредоточена в функции eval, которая выясняет границы вычислимости выражений по их списочному представлению. Функция read используется для создания значений из их текстовых представлений — отображение текста в структуры данных Clojure. Функция print отображает данные в их текстовое представление на внешнем устройстве. Синтаксис Clojure основан на S-выражениях и помимо обычных списков поддерживает синтаксис отображений, множеств и векторов. Специальная функция quote (или ') выполняет блокировку вычислений.

---

78 https://clojure.org/reference/reader — описание языка Clojure.

79 http://www.paulgraham.com/rootsoflisp.html - Книга Пола Грэма «Корни Лиспа», описывает точность фундаментальных вычислительных операций, первоначально обнаруженных и описанных Джоном Маккарти, это можно видеть в Clojure.

80 Семейство языков Lisp в середине 1980-ых разделилось на Lisp-1 и Lisp-2. Различия сводятся к особенностям реализации логических значений, взаимодействия статических областей видимости определений функций с динамическими областями вызовов функций, возможностями списков свойств атома и ряда других аспектов стиля представления и обработки программ, начиная с принудительной компиляции.

81 https://clojure.org/reference/lisps — отличия от языка Lisp.

9.2. Синтаксис данных

Подобно языку Common Lisp к скалярным типам (атомы, числа и строки), добавлены аппаратные типы чисел для эффективной арифметики, выбираемые системой автоматически. Поддержана расширяемость структур данных с использованием # как тега. Расширен синтаксис непосредственно представимых структур данных, таких как отображения, вектора, множества:

booleans: true, false[82]
strings: "foo bar"
characters: \c, \tab
symbols: name [83]
keywords: :key [84]
integers: 123
floating point numbers: 3.14
lists: (a b 42)
vectors: [a b 42]
maps: {:a 1, "foo" :bar, [1 2 3] four} [85]
sets: #{a b [1 2 3]}
nil: nil (a null-like value)

Ключевые слова и символы являются атомами (идентификаторами), но если ключевые слова означают сами себя, то символы представляют значения (переменные или функции). Отсутствие программируемых свойств символа при необходимости можно компенсировать использованием отображений. Значения символов зависят от контекста и пространств имен, доступных в этом контексте. Текущее пространство имен всегда связано с символом *ns*[86]. Переменные — это места хранения, в которых могут храниться любые данные. Все локальные переменные неизменяемы.

```
(def x 1)
   ;= #'user/x — рабочее пространство имён
x  ;  доступ к значению переменной
   ; = 1 результат
```

Синтаксис для наиболее распространенных структур данных использует квадратные и фигурные скобки (вектора, отображения, множества), кроме того, поддержаны метаданные — коды из битов информации, которая может быть связана со значением или ссылкой.

```
'(a b :name 12.5) ;; list - список
['a 'b :name 12.5] ;; vector - вектор
{:name "Chas" :age 31,[87] :home "World"} ;; map — отображение, хэш-таблица
#{1 2 3} ;; set — множество — элементы уникальны
```

9.3. Определение функций и выражений

Операции конструирования выражений в языке Clojure называют, как и в языке Lisp, специальными функциями (def, defn, if ,fn, #, let, case, cond, loop/recur, try/catch, quote ('), do), формы с их применением могут иметь свой собственный синтаксис и

---

82 Тип «booleans» по существу работает как супертип, объединяющий конечный тип данных «false» и бесконечно расширяемый тип данных «true», в который автоматически включается любое данное, возникающее при вычислении, если оно не входит в тип «false».

83 Символ — это атом, обладающий кроме имени системными или программируемыми свойствами.

84 Ключ — это атом, обозначающий сам себя, не имеющий других свойств.

85 Запятая используется для наглядности, эквивалент пробелу.

86 Обрамление звёздочками принято для системных переменных.

87 Запятая эквивалентна пробелу, используется для наглядности.

семантику и быть реализованы как макросы, т. е. без предварительного вычисления аргументов.

Специальная форма fn выполняет создание представлений функций.

```
(fn [x]
(+ 10 x))
```

Определения функций, создаваемые fn, могут неформально сопровождаться пред- и пост-условиями, что обеспечивает поддержку проверки утверждений об аргументах и полученных значениях, а также позволяет при тестировании проверять инварианты функций и использовать методы верификации программ.

Другие функции создают или переопределяют переменные, неявно используя функцию def внутри себя. Они имеют префикс «def» (defn, defn-, defprotocol, defonce, defmacro, deftype, defrecord и defmethod и т. д.).

Имеются функции, принимающие и/или вырабатывающие последовательности или отображения. Векторы являются функциями своих индексов. Возможны функции с несколькими арностями, кроме позиционных аргументов бывают необязательные, ключевые, серийные. Некоторые параметры могут иметь значения по умолчанию. Специальная форма letfn позволяет определить несколько именованных функций одновременно, такие функции могут использовать друг друга.

Возможна деструктуризация структур данных, работающая с любой последовательностью, включая:

- Списки, векторы и последовательности Clojure.
- Любая коллекция, реализующая java.util.List (например, ArrayLists и LinkedLists).
- Java-массивы.
- Строки, которые деструктурированы на символы.
- Списки аргументов функций.

Деструктуризация означает переход от ранее созданной структуры данных к структуре с другим методом доступа при сохранении её наполнения[88], допускается использование подчеркивания (_) для обозначения игнорируемой позиции. Исходная структура сохраняется.

Деструктуризация последовательности аргументов функции позволяет использовать нумерацию аргументов, список которых можно рассматривать как вектор[89]. К первому аргументу функции можно обращаться просто используя %.

```
#(Math/pow % %2)  :эквивалентно
#(Math/pow %1 %2).

((fn [x y z] (+ x y z)) 3 4 12)  ;= 19
; эквивалентно:
(let [x 3
      y 4
      z 12]
     (+ x y z)   )
```

Многие формы (включая fn, let, цикл и любые производные, такие как defn) заключают свое тело в неявное выражение do[90]. Специальная форма do вычисляет все перечисленные в нём выражения по порядку и возвращает значение последнего выражения.

```
(do
```

88 Напоминает организацию работы с векторами в языках Fortran и APL.

89 Подобно некоторым языкам заданий и макропроцессорам.

90 Эквивалент ProgN в языке Lisp.

```
(println "hi")
(apply * [4 5 6]))
; hi
;= 120
```

recur — это более низкоуровневая специальная функция цикла и рекурсии, она передает управление самому локальному заголовку цикла, не занимая пространство стека. Имеются и другие формы императивного цикла (doseq, dotimes и др.).

Специальная функция if реализует условный оператор. Условные выражения Clojure определяют, что логическая истина — это что угодно, кроме Nil[91] или ложности.

Специальная функция cond — аналогична конструкции «else if».

Специальная функция var даёт доступ к значению переменной:

```
(var x)
;= #'user/x
```

Имеются специальные функции try/catch для обработки исключений.

9.4. Параллелизм

Параллельное программирование использует транзакционную память как в базах данных, агентов и разные виды динамических 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. Акцент на исключении общих ошибок и сбоев, возникающих при использовании изменяемых состояний, ради быстрой отладки программ, что позволяет на языке Clojure решать задачи, требующие параллелизма или распараллеливания.

Хотя Clojure делает упор на использование неизменяемых структур данных и значений, существуют контексты и приложения, в которых необходимо производить изменения состояния (специальная функция set!) или синхронизовать выполнение потоков: откладывание, ожидание, обещание, передача, готовность, блокировка (delay, future, promise, deliver, is-done?, deref), позволяющие контролировать ход вычисления.

В качестве компромисса между идеями чистого ФП и необходимостью изменения состояний при организации параллельных процессов введены ссылочные типы — структуры, обеспечивающие разные виды дисциплин доступа к памяти и стратегий многопоточности (atom, ref, agent). Возможные проблемы с коллизиями имён решаются с помощью пространств имен или пакетов подобно автономным модулям. Экземпляры ссылочных типов могут быть разыменованы путем добавления префикса @ к символу, обозначающему экземпляр.

Примеры[92]

Hello world:

```
(println "Привет, мир!")
```

Потокобезопасный генератор уникальных серийных номеров:

```
(let [i (atom 0)]
 (defn generate-unique-id
  "Возвращает различные числовые ID для каждого вызова."
  []
  (swap! i inc)))
```

91 Но не пустого списка ( ).

92 Скопированы из источника

Анонимный подкласс java.io.Writer, который ничего не выводит, и макрос, используемый чтобы заглушить весь вывод внутри него:

```
(def bit-bucket-writer
  (proxy [java.io.Writer] []
    (write [buf] nil)
    (close []    nil)
    (flush []    nil)))

(defmacro noprint
  "Вычисляет заданные выражения, заглушая весь *вывод* на экран".
  [& forms]
  `(binding [*out* bit-bucket-writer]
     ~@forms))

(noprint
 (println "Hello, nobody!"))
```

Здесь 10 потоков манипулируют одной общей структурой данных, которая состоит из 100 векторов, каждый из которых содержит 10 (изначально последовательных) уникальных чисел. Каждый поток многократно выбирает две случайных позиции в двух случайных векторах и обменивает местами их значения. Все изменения векторов происходят в единой транзакции путём использования системы транзакционной памяти clojure. Поэтому даже после 1000 итераций в каждом из потоков числа не теряются.

```
(defn run [nvecs nitems nthreads niters]
  (let [vec-refs (vec (map (comp ref vec)
                           (partition nitems (range (* nvecs nitems)))))
        swap #(let [v1 (rand-int nvecs)
                    v2 (rand-int nvecs)
                    i1 (rand-int nitems)
                    i2 (rand-int nitems)]
                (dosync
                 (let [temp (nth @(vec-refs v1) i1)]
                   (alter (vec-refs v1) assoc i1 (nth @(vec-refs v2) i2))
                   (alter (vec-refs v2) assoc i2 temp))))
        report #(do
                  (prn (map deref vec-refs))
                  (println "Distinct:"
                           (count (distinct (apply concat (map deref vec-refs))))))]
    (report)
    (dorun (apply pcalls (repeat nthreads #(dotimes [_ niters] (swap)))))
    (report)))

(run 100 10 10 100000)
```

Вывод предыдущего примера:

```
([0 1 2 3 4 5 6 7 8 9] [10 11 12 13 14 15 16 17 18 19] ...
 [990 991 992 993 994 995 996 997 998 999])
Distinct: 1000

([382 318 466 963 619 22 21 273 45 596] [808 639 804 471 394 904 952 75 289 778] ...
 [484 216 622 139 651 592 379 228 242 355])
Distinct: 1000
```

Есть основания полагать, что диалект Clojure отвечает на вопрос «**ГДЕ** новые горизонты, в освоении которых помогает продуктивность и моделирующая сила языка Lisp?». Ответ заключается в следующем:

- Универсальность символьных вычислений распространена на явный синтаксис отображений, множеств и векторов. Используются пространства имен или пакетов подобно автономным модулям. Введены метаданные — кодированный аналог списка свойств, который может быть связан со значением или ссылкой.
- Равноправие параметров функции допускает несколько арностей, кроме позиционных аргументов можно объявить необязательные, ключевые, серийные.
- Самоопределение рекурсивных определений, семантика вычислений которых сосредоточена в функции eval, распространяется на представление параллелизма или распараллеливания, можно синхронизовать выполнение потоков: откладывание, ожидание, обещание, передача, готовность, блокировка (delay, future, promise, deliver, is-done?, deref), контролировать ход вычисления, а также эффективные формы циклов, не использующих стек.
- Гибкость распределения памяти обобщена до деструктуризации структур данных, работающей с любыми последовательностями, включая списки аргументов.
- Неизменяемость данных при необходимости изменений поддержана как транзакционная память в базах данных, используются агенты и разные виды динамических переменных, реализованных как ссылочные типы — структуры, обеспечивающие разные виды дисциплин доступа к памяти и стратегий многопоточности (atom, ref, agent).
- Единственность результата функции ужесточается возможностью представления пред- и пост-условий, проверки утверждений об аргументах и полученных значениях, при тестировании можно проверять инварианты функций и использовать методы верификации программ.

9.5. Выводы

Диалект Clojure сохраняет примерно 80% особенностей языка Lisp, уточняет рад 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. Он стал явным указательным типом данных, позволяющим поддерживать различные дисциплины обработки данных, возникающие в разных моделях параллельных вычислений, разнообразие которых непредсказуемо велико. Кроме того, механизм реструктуризации данных распространён на список параметров, что расширяет форматы вызова функций. Более детально различия описаны в таблице 6.

Таблица 6

Сравнение особенностей Lisp-Clojure [93]

Характеристика языка	*Обоснования и решения Lisp*	*Обоснования и решения* Clojure
-10+30+20+18	***Принципы и выбор решений***	***Уточнения и изменения***
1. ***Универсальность***	Любую информацию **для компьютерной обработки** можно представить в символьной форме.	Символьные формы распространены на представление векторов, множеств и хэш-таблиц. Методы обработки ограничены возможностями JVM.
2. ***()***	На терминале не было других скобок. Зато и теперь они набираются на любом регистре, в отличие от	Кроме круглых скобок используются квадратные для векторов и фигурные для отображений и множеств.

93https://clojure.org/reference/lisps

	фигурных и квадратных	
3. *Символьные формы. Типы данных*	Символьные формы бывают **неделимыми**, если в языке не предусмотрен разбор их на части, или **составными**, если язык допускает их сборку из частей и разделение на части.	Примитивными данными считаются атомы, числа, строки, булевы значения, ключевые слова, пустой список. Составными данными считаются списки, вектора, символы, отображения, множества, и **указательные переменные**. Различаются функции (functions), потоки (streams), ссылки (refs - **указательные переменные**.), агенты (agents) . Поддержана **деструктуризация** составных данных, включая списки параметров функций.
4. *Атомы*	Символьные формы для представления неделимых данных называются атомами. Любую сущность, включая ключевые слова, можно представить как атом, именовать в соответствии с особенностями области приложения. Атомы выполняют роль **идентификаторов** для значений любой природы.	Выделены из атомов символы, ключевые слова и указательные переменные. Нет программируемых списков свойств атома, вместо них используются метаданные. (atom value) - создает новый атом с начальным значением value. @atom - получает значение атома. (reset! atom new-value) - устанавливает новое значение атома. (swap! atom function & args) - применяет функцию к текущему значению атома и обновляет его. Имеется **gensym.** В Clojure атом - это структура данных, которая: хранит одно значение. Функции atom, ref, agent, promise и future можно использовать для управления состоянием данных и параллелизмом. Обеспечены атомарные (thread-safe) обновления значений. Несколько потоков могут пытаться изменить значение атома одновременно, но изменения будут выполнены последовательно, без гонок данных. Для сложных состояний часто используют ссылочные типы - ref.
5. Константы	Введены так называемые псевдо-атомы, смысл которых виден по форме их записи, - это **числа и строки**. Строки могут разделяться на символы, но по соглашению они считаются псевдо-атомами. Составные формы строятся из пар элементов произвольной природы, разделённых точкой и заключенных в скобки. **Список строится из пар** и завершается парой, содержащей последний элемент и пустой список.	Кроме чисел и строки, можно представлять вектора, множества и хэш-таблицы.
6. Списки, иерархия форм	Основная структура составных данных — список, перечисленные элементы которого заключёны в скобки и при необходимости разделяются пробелами. Нет загромождения запятыми или другими разделителями. **Элементы списка могут быть любой природы и разного вида**.	Иерархия может быть из любых структур данных, а также из областей видимости и пакетов. *Допускается использование запятой в качестве разделителя элементов списка.*
7. *Пустой список* ***Nil = ()***	Пустой список не может быть разделён на части средствами языка, поэтому он является **одновременно**	Булевы значения False, True. Nil является представлением ложного значения, но он не равен пустому списку.

	списком и атомом. Его можно считать отдельным типом значений.	Пустой список — это отдельный тип данных.
8. **Гибкость распределения памяти**	Имена атомов, числа, стоки, списки **не имеют ограничений на длину**, их размеры не зависят от формата машинных слов и границ блоков памяти..	Кроме данных без ограничений на размер возможны данные ограниченного размера.
9. *Списки свойств атома*	Атомы, смысл которых не ограничен их именем, сопровождаются списком свойств, часть которых определены в системе программирования, другие **могут программироваться.** Разница между переменными и разными категориями функций представляется в списке свойств атомов. При пополнении списка свойств ранее существовавшие свойства не исчезают, хотя по умолчанию используется **самое новое определение.**	Вместо списков свойств атома — метаданные Метаданные неизменяемы,
10. ***Неизменяемость данных***	Обработка списков подчинена принципу неизменяемости данных. Преобразованные списки строятся в новой памяти **без искажения исходных данных**.	Существуют изменяемые структуры данных, такие как: Хеш-таблицы (hash tables), Ячейки (Boxes) - изменяемые контейнеры, Векторы (Vectors), Структуры с изменяемыми полями и функции-мутаторы (Mutators): mcons, mcar, mcdr , vector, vector-ref, vector-set! -., box, unbox, set-box!
11. Функции	Любое понятие программирования можно рассматривать как функцию или как применение функции. Определения функций допускают **самоопределение (рекурсия)** и подчинены принципам равноправия аргументов и единственности результата.	Программы строятся из выражений с помощью специальных функций и макросов. Изменены названия: lambda – fn, car – first, cdr - rest и т. п.
12. ***Равноправие параметров функции***	Аргументы функции **не зависят друг от друга**. Даже если при реализации языка порядок их вычисления определён, результат функции не зависит от этого порядка.	Свободные переменные зависят от **контекста, разного** в Lisp и Clojure.
13. *Конструкторы, макросы, специальные функции*	Можно без предварительного вычисления аргументов функции производить обработку их представлений внутри функции — **специальные функции**. Это позволяет программировать разные схемы управления вычислениями, конструировать представления выражений и разных категорий функций, преобразовывать программы, например, **компилировать** или оптимизировать.	def defn if fn # let case cond loop/recur try/catch quote (') do defn, defn-, defprotocol, defonce, defmacro, deftype, defrecord и defmethod и т. д. Макросы выполняются при компиляции. Можно сохранять списочные формы программы
14. **Единственность результата функции**	При вызове функции в одном и том же контексте, при одних и тех же аргументах получается **один и тот же результат.**	Как в языке Lisp.
15. Выражения (формы)	Программы и выражения представляются как символьные формы — это список, первый элемент которого содержит **представление** функции, а остальные — **представления** аргументов. Пустой	Как и в языке Lisp символьная форма — это список, первый элемент которого содержит представление функции, а остальные — представления аргументов.

	список () или атом Nil считаются константной формой, всегда имеющей значение Nil.	
16. **Рекурсия (3)**	Именованные формы выражений допускают **само-применимость определений.**	Как в языке Lisp.
17. Гомоиконность	Представление программы в виде списков позволяет чётко видеть структуры данных, представляющие программу. Такое преставление программы фактически является **абстрактным синтаксическим деревом.**	Как в языке Lisp,
18. *Pure Lisp*	cons, car, cdr, eq, atom, lambda, label, cond, quote, eval.	def let if fn apply map filter reduce loop/recur - нет сравнений и quote, - для отладки используют throw/try/catch
19. Алгебраические системы функций	Базовые функции над конкретными типами значений группируются в комплекты, обладающие **полнотой операций** и/или отношений, определены обратные функции и минимальные значения, выполняющие роль нулей или единиц.	(cons, first, rest, empty?) map, filter (=, not=, <, >) (+, -, *, /) (fn, defn, apply), partial (def, defn, defmacro, macroexpand) (quote, eval) (and, or, not)
20. Аксиоматика отношений	Поддерживаются аксиомы, позволяющие в отдельных случаях получать результаты функций **без выполнения вычислений**.	Возможно такие отношения учитываются при оптимизирующей компиляции.
	Семантический базис	
21. ***Quote***	Любую форму можно заблокировать от вычисления, сделав из неё константу.	Вместо **Quote** можно отложить вычисление — try/catch или синхронизовать future, promise, deliver, is-done?, deref
22. **Eval**	Для любой представленной или построенной в программе формы можно вычислять её значение по мере необходимости, в том числе, вычислять ранее заблокированные или сконструированные формы.	Как и в языке Lisp.
23. **Lambda** - свободные и связанные переменные	Lambda при **объявлении безымянных функций** выделяет **список связанных переменных**, они получают значение при вызове функции. Определяющая форма функции может содержать и другие переменные, называемые **свободными**, они получают значение из внешнего контекста.	**Fn** – с точностью до обозначений как и в языке Lisp.
24. *Пространства имён*	Поддержано два пространства имён — раздельно для локальных и глобальных определений с **приоритетом локальных**. Имена переменных и функций расположены **в общем пространстве.**	Встроены: clojure.core clojure.string clojure.math clojure.java.io clojure.test clojure.data clojure.walk clojure.repl
25. Контекст — иерархия областей видимости, пространства имён, динамика и статика	Если одна функция вызывает другую, то внутренняя через свободные переменные может использовать **значения переменных из объемлющих функций**.	Применяются и динамические, и статические области видимости переменных[94].

94 В середине 1960-ых появился диалект Lispkit — реализация чисто функционального подмножества Pure Lisp с лексической областью видимости.

26. **Label** - приоритет локальным определениям	Локальные определения освобождают от придумывания уникальных имён. Теоретически глобальные определения можно реализовывать как самые внешние локальные.	let: letfn: binding: with-local-vars:
27. **Define/Defun –** глобальные определения	Кроме создания локальных безымянных и именованных функций имеется **практичный компромисс** в виде глобальных именованных функций (Define/Defun).	приоритет отдается локальным переменным над глобальными.
28. Мульти-функции и параметры	Некоторые функции могут быть определены для работы с произвольным числом параметров.	Возможны функции с несколькими арностями, кроме позиционных аргументов бывают необязательные, ключевые, **серийные.** Некоторые аргументы могут иметь значения по умолчанию. Возможна деструктуризация последовательности аргументов, что позволяет использовать их нумерацию как элементов вектора.
29. Полиморфизм	Атом может одновременно обладать разными свойствами, возможно **неоднократное вхождение** любого свойства. Атом может представлять и значение, и несколько разных категорий определения функции, и ещё что-нибудь по усмотрению программиста. Функция может иметь ряд различных определений, включая одновременное вхождение символьных и кодовых определений, например, результатов компиляции. По умолчанию работает самое новое определение, но и **к прежним определениям доступ возможен.**	Таким образом, в зависимости от количества аргументов, вызываемое определение функции будет различаться. Диспетчеризация (multimethods). Протоколы (protocols). Реализация интерфейсов Java. Функциональный полиморфизм.
30. Функции высших порядков	Представления аргументов функции могут быть **символьными формами определения** или представления функции. Поэтому можно определять функции высших порядков, использующие функциональные переменные. Аргументами и/или результатами функции высших порядков могут быть представления других функций.	Как и в языке Lisp поддержаны функции высших порядков.
31. Отображения, свёртки и фильтры	Применение функций, передаваемых через параметры, требует согласования не только типов данных или значений, но и рангов функций и контекста их вызова, что можно реализовать конструированием **рецепта применения функциональной переменной**.	Map: filter: reduce: take, drop, take-while, drop-while:
32. Функциональные переменные	Представления функций — не более чем символьные формы, поэтому нет никаких препятствий существованию функциональных переменных, значением имеющих представление функции, и передаче представлений функций в качестве параметров при вызове функций или **выдаче их как результат.**	def ops [(* 2) (+ 3) (- 1)]) (map #(% 5) ops) ; => (10 8 4)
33. Безымянные	Передаваемые через параметры	Как и в языке Lisp имеется возможность

определения	функции часто имеют **разовый характер**, а именование функций осмысленно для многократного использования. Безымянные функции строит конструктор Lambda.	формировать безымянные функции (fn), дополненная конструктором defn, подобным lambda-case, поддерживающим создание полиморфных определений функции, отличающихся форматом списка параметров.
34. ***Cond - If*** *ветвление*	В Pure Lisp хотя бы одна ветвь должна быть выбрана. В Lisp 1.5 в форме Prog это не обязательно, при отсутствии истинного предиката значением ветвления является Nil.	true – булево значение If – условная форма ***When*** *– цикл пока истинно условие*
35. **Eq – сравнение указателей или атомов**	Существуют полный комплект предикатов для **контроля процесса выполнения** программы.	Различается сравнение указателей, значений и структур данных: eq?, eqv?, equal? как в Scheme
36. **Atom**, Null, Number - контроль типов значений	**Каждый тип данных имеет предикат**, позволяющий в любой момент интерпретации программы, выяснять принадлежность значений нужному типу данных.	Контроль типов данных в Clojure основан на динамической типизации, проверке типов во время выполнения, встроенных типах данных, протоколах и реализациях, а также возможности аннотирования типов функций. Числа: number? int? double? string? char? Коллекции: coll? vector? list? set?map? Логические значения: boolean? (true или false). Функции: fn? Атомы и агенты: atom? agent? Java-объекты: instance?
37. Объявление типов данных	**Для компиляции функций следует заранее объявлять тип данных**, передаваемых через свободные переменные (declare).	В Clojure типы значений выводятся и проверяются во время выполнения, а не типы данных во время компиляции.
38. *Предикаты*	**Любая функция** может выполнять роль предиката. Ложным значением считается значение (), оно же Nil, остальное — истина.	true или false/Nil
39. ***Логика -*** *( ) Nil, значение «ложь»*	Nil выполняет роль значения «ложь», любое другое значение работает как «истина». Поиск в списках при неудаче **завершается на пустом списке**. Эффективно пустой список рассматривать как значение «ложь», подобно коду «0» во многих языках.	Введен логический тип значений, с сохранением ложного Nil, но без ().
40. Компиляция - **compile**	Компиляция функций рассматривается как **оптимизация программ**, позволяющая повысить скорость их выполнения примерно в 50 раз ценой существенного расхода времени на обработку программы. Это достаточное основание подвергать компиляции достаточно отлаженные функции, чтобы избежать накладных расходов на компиляцию программ с неотлаженными функциями.	В Clojure, типы значений выводятся и проверяются во время выполнения, а не во время компиляции. При компиляции сохраняется исходная списочная форма программы.
	Диагностика и отладка	
41. ***Error*** *– диагностика ошибок*	Функция Error - обработчик ошибок, сообщающий диагноз дефекта в определении выражений и позволяющий **программировать продолжение** вычислений.	try, catch, и finally. Leiningen - запуска тестов, анализа кода и проверки качества. Clojure.tools.namespace:зависимостями между пространствами имен

		Clojure.spec: спецификации для данных и функций, Клиенты для IDE: IntelliJ IDEA, Emacs, Vim и VS Code, Анализаторы кода: Eastwood и Kibit, статический анализ кода Clojure Написание юнит-тестов и интеграционных тестов
42. ***Trace/Untrace*** – *отладка*	Режим **REPL** (Read-Eval-Print Loop) Поддержана возможность смотреть **аргументы-результаты отлаживаемых функций** по ходу их вычисления, список которых может быть при необходимости изменён. **Trace** – принимает список функций, значения аргументов и результаты которых следует выдавать при каждом вызове. Для этого в списки свойств функций размещается флаг. **Untrace** – удаляет флаг из списка свойств функций, отслеживание которых уже не нужно.	REPL Stacktrace: clojure.tools.trace/trace, Debugger: Cider и CLJR clojure.tools.trace/trace, позволяют пошагово выполнять код, устанавливать точки останова и просматривать состояние переменных. Logging: clojure.tools.logging. Profiling:clojure.core/time и clojure.tools.namespace.track-deps. clojure.test, для создания тестов и проверки правильности работы своих функций. try...catch core.async
	Расширения и внешний мир	
43. **Открытость**	Программа имеет доступ ко всем функциям системы программирования (интерпретатора, компилятора, отладчика) и может им давать свои определения. Всё, что понадобилось системе, может быть полезным и программисту. **Изменять можно любые определения, кроме Nil.**	JVM JIT
44. ***Print, Read*** – *ввод-вывод,* ***I/O***	Print формально является **тождественной функцией вывода** данных, он печатает свой аргумент на внешнем носителе. Ввод принимает данное с внешнего устройства, строит эквивалентную ему символьную форму, становящуюся его результатом, что удобно при переходе от обработки констант к обработке произвольных данных.	Java Interoperability: java.io.File, java.io.BufferedReader, java.io.BufferedWriter java.io.InputStream, java.io.OutputStream clojure.java.io System.out System.in: java.net.Socket, java.net.ServerSocket: java.net.URL, java.net.URLConnection: clojure.java.shell: http-kit и aleph print выполняет вывод и **возвращает nil.**
	Прагматика реализации	
45. ***GC*** – *«Сборка мусора»,* ***гибкость распределения памяти***	Автоматизировано **повторное использование памяти**, чтобы освободить специалиста для более существенных задач. Возможен **вызов GC из программы**.	persistent data structures и transients, System/gc. VisualVM, JProfiler) для выявления утечек памяти и неэффективного использования памяти
46. ***Структуры данных***	Хотя на уровне языка нет ни векторов, ни хэш-таблиц, в реализации поддержана работа с **векторами для взаимодействия с машинным кодом** процедур и с **хэш-таблицами для идентификации** атомов. Кроме того, в языке поддержаны две функциональные модели хэш-таблиц — **глобальный список свойств атома и локальный ассоциативный список**, локально связывающий атомы с их определениями,	Persistent Data Structures (radix trees, vector trees,структуры данных JVM, Compiler-Generated Structures Closure, Objects: Замыкания (closures) - это функции, которые "захватывают" переменные из области видимости, в которой они были определены. Metadata. Threading and Concurrency Structures. Atoms: Атомы (atoms), Ссылочные типы (refs), Агенты (agents), Index Structures — БД Memory Management Structures.

	позволяющие не нарушать принцип неизменяемости данных..	
47. ***Присваивания*** *и адреса*	Присваивания поддержаны в двух формах. Это псевдо-функции **Rplaca и Rplacd,** изменяющие значение по указанному адресу, и функции **Set и Setq внутри формы Prog**, выполняющие изменение значений рабочих переменных без побочного эффекта на внешнем уровне, что позволяет Prog рассматривать как **чистую функцию**, совмещающую декларативный ответ на вопрос «Что?» с императивным ответом на вопрос «Как?».	Присваивания стали явными (set! Def) Различаются константы и переменные.
48. ***Замыкание функции***	Для передачи функций со свободными переменными в качестве функциональных параметров интерпретатор выполняет **формирование рецептов**, однозначно связывающих переменные **в точке вызова функции с её контекстом**, а не в точках её вычисления, в которых пространства имён могут различаться.	Замыкание функций строится автоматически для внутренних функций при их определении, т. е. неявно как в Scheme.
49. ***Устройства*** *ввода-вывода данных и ОС*	Имеются функции, поддерживающие доступ к периферийным устройствам и операционной системе.	Java Virtual Machine (JVM), предоставляет доступ к широкому спектру библиотек и функций, которые позволяют взаимодействовать с операционной системой и периферийными устройствами.
50. **Раскрутка** компилятора и интерпретатора	Интерпретатор и компилятор для языка Lisp первоначально были определены на языке Lisp, а затем воспроизведены в машинном коде.	Компиляция в байт-код не является принудительной. Самодостаточность и само-интерпретация. eval является основным строительным блоком для интерпретатора из модулей. clojure.walk: обход структуры данных Clojure. Лексический анализ (Lexing): токены. Синтаксический анализ (Parsing): Строит (AST) из токенов. Анализ семантики: Генерация байткода: AST в байткод Java Эти этапы реализованы в коде Clojure. Функции для генерации байткода используют библиотеки для работы с Java bytecode.
51. **Альтернативность** решений, полное пространство для мысли: «Если нельзя, но очень хочется, то можно».	• Реализация вычислений с помощью **обычных функций**, выполняемых после вычисления параметров, дополнена **специальными функциями**, вместо значений параметров, обрабатывающих представления аргументов. • Кроме организации **правильных вычислений** поддерживается представление обработки **ошибочных ситуаций** и продолжения вычислений. • Пространство **абстрактных атомов** расширено конкретными **типами данных**, принятыми в основных областях приложения — псевдо-атомы. • **Автоматизация** «сборки мусора» допускает **вызов GC из программы**.	В дополнение к альтернативам языка Lisp: • *пред- и пост-условия,* • *указательные переменные.*

	• Кроме интерпретации выражений (**eval**) поддержана компиляция функций (**compile**).	
52. *Изобразительные средства* **( ) . ' ;** #	Бедный синтаксис освобождает от необходимости размышлять на тему, «где поставить запятую, где не ставить запятой». **( )** - границы точечной пары или списка, **.** - разделитель элементов пары, из которых строятся все структуры данных, **'** - кавычка — префикс, объявляющий представление константы, **'**x эквивалент (Quote x). **;** - объявление начала строчного комментария.	Синтаксис обогащен префиксами, что можно рассматривать как краткие имена функций: #"" - регулярные выражения =>: Стрелка (для ассоциативных массивов и map). ->, ->>, ..: Операторы цепочки (threading macros). ?: Тернарный оператор (в макросах, например if). #: Префикс для комментариев, а также для литералов (например, #{} для множества, #() для анонимных функций, #_ для исключения). @: Dereference (разворачивание) атомов, ссылок и переменных, содержащих значения, которые могут меняться. %: Placeholder (заполнитель) в анонимных функциях и макросах. $: Используется в java-интерфейсе. , (запятая): Разделитель между элементами в списках, векторах, хэш-таблицах и других структурах данных. В Clojure запятые, в основном, **игнорируются компилятором.** ", \": Кавычки для строк. \ (обратный слеш): Для экранирования специальных символов в строках и символах. |: Используется для разделения в регулярных выражениях. : (двоеточие): Префикс для ключевых слов (keywords). / (слеш): Используется для именования пространств имен (namespaces) и в дробях. ...: Эллипсис (для variadic функций и в макросах). ...: (в макросах) Для вариативных аргументов. ^: для метаданных. ~: для цитирования и нецитирования (unquote).

Анализ таблицы 6 показывает, что основные отличия от языка Lisp обладают сходством с диалектом Scheme. Нет программируемых списков свойств атома, вместо списков свойств предложены неизменяемые метаданные, атом хранит одно значение. Введены булевы значения False, True. Хотя Nil признаётся представлением ложного значения, он не равен пустому списку. В базис диалекта не включены сравнения и блокировка quote. Макросы выполняются при компиляции. Замыкания функций строятся по иерархии определений, а не вызовов функций. Кроме присваиваний введены функции-мутаторы (Mutators). Функция print выполняет вывод и возвращает Nil.

Унаследованные от языка Lisp и диалекта Scheme решения дополнены средствами для верификации программ и организации параллельных процессов. Более конкретно, на уровне синтаксиса предложены формы представления векторов, множеств и хэш-таблиц с использованием квадратных скобок для векторов и фигурных для отображений и множеств. Допускается использование запятой в качестве разделителя элементов списка. Пустой список — это отдельный тип данных. Используются данные ограниченного размера. Ключевые слова не имеют метаданных.

Понятие символа и атома трактуется подобно контейнерам как указательная переменная для представления различных стратегий много-поточности, дисциплин управления состоянием общей памяти и параллелизмом (atom, ref, agent, promise и future)

в предположении, что изменения состояний будут выполнены последовательно, с поддержкой транзакционной памяти, без гонок данных, допуская диспетчеризацию (multimethods) и протоколы (protocols). Можно синхронизовать выполнение потоков: откладывание, ожидание, обещание, передача, готовность, блокировка (delay, future, promise, deliver, is-done?, deref).

Определения функций можно сопровождать пред- и пост-условиями, letfn позволяет определить несколько именованных функций одновременно, такие функции могут использовать друг друга. Имеются и другие конструкторы определений с префиксом «def» (defn, defn-, defprotocol, defonce, defmacro, deftype, defrecord и defmethod и т. д.). Механизм ветвлений кроме if и cond может использовать When.

Поддержаны и динамические, и статические области видимости, иерархия может быть из любых структур данных, а также из областей видимости и пакетов, много разных встроенных пространств имён, функциональный полиморфизм допускает кроме позиционных аргументов необязательные, ключевые и серийные. Кроме того, предложен механизм деструктуризация структур данных, включая деструктуризацию последовательности параметров. Имеется реализация интерфейсов Java, JVM, JIT.

Значительно расширен инструментарий отладки, кроме try, catch, и finally доступен запуск тестов, анализ кода и проверка качества (Leiningen), проверка зависимостей между пространствами имен (Clojure.tools.namespace), спецификации для данных и функций (Clojure.spec), для создания тестов и проверки правильности работы своих функций (clojure.test), анализаторы кода (Eastwood и Kibit), статический анализ кода Clojure и многое другое (Stacktrace, clojure.tools.trace/trace, Debugger, Logging, Profiling, track-deps.core.async, и др.).

Значительные дополнения произошли на уровне синтаксиса символьных форм, их можно рассматривать как краткие имена функций.

Заключение

Отмечая разницу в целях создания диалектов языка Lisp, можно заметить, что рассмотренные диалекты обладают стабильным реализационным ядром, основанным на конкретном комплекте структур данных и механизмов их обработки. Вариации сводятся к границам между синтаксисом, семантикой и прагматикой языка, между периодом компиляции и выполнения программы, в трактовке значений «ложь». Более детально эта разница представлена в Таблице 7. Можно видеть смягчение программистского скептицизма по отношению к принятым в языке Lisp решениям по мере прогресса элементной базы и развития ИТ.

Таблица 7

Сравнение вариаций в определении системного ядра диалектов языка Lisp.

Конструкции	**1958 Lisp**	**1976 Scheme**	**1984 Clisp**	**1995 Racket**	**2007 Clojure**
Символьные формы.	(A 2 …)	(1 2 …) [1 2 ...]	(A 2 …)	(1 2) [1 2] {1 2 3}	(1 2) [1 2] {1 2 3}
Списки	Синтаксический базис	Деструктуризация с векторами	Синтаксический базис	Деструктуризация с векторами	Деструктуризация с векторами
Вектора	Неявная прагматика параметров процедур	Базовая деструктуризация списков в вектора	Семантика реорганизации списков в вектора и обратно	Синтаксис и базовая деструктуризация списков в вектора	Синтаксис и базовая деструктуризация списков в вектора
Множества	Неявная прагматика параметров функций	Неявная прагматика параметров функций	Явная семантика функций работы с множествами	Синтаксис и семантика	Синтаксис и семантика допускают деструктуризацию через вектора.
Размеченные множества	Неявная прагматика	В пределах системных	Неявная прагматика	Синтаксис и семантика хэш-	Синтаксис и семантика хэш-

	списков свойств атома	свойств.	программируемых списков свойств символа	таблиц, возможны модели.	таблиц, возможны модели
Хэш-таблицы	Неявная прагматика списка глобальных атомов и значений	Семантика областей видимости	Семантика пространств имён и пакетов	Синтаксис и семантика хэш-таблиц	Синтаксис и семантика хэш-таблиц, варианты представления
Ассоциативный список атома	Семантика вычислений	Прагматика вычислений	Семантика вычислений	Прагматика вычислений	Синтаксис хэш-таблиц и прагматика вычислений,
Атомы	Символы, числа и строки произвольной длины	Числа, строки и коды произвольной длины	Символы, числа и строки произвольной длины с добавление конечных чисел	Числа, строки и коды произвольной длины	Символы, числа и строки произвольной или конечной длины. Указательные переменные.
( )	() = Nil, отдельный тип данных, работает как значение «ложь»	() ≠ Nil	() = Nil, отдельный тип данных, работает как значение «ложь»	() = Nil работают как значение «ложь» наряду с «#f».	() ≠ Nil, отдельный тип данных. Nil работает как значение «false»
Символы	Атомы, обладающие списком программируемых свойств в любом количестве.	Идентификаторы переменных и функций без программируемых свойств.	Атомы, обладающие списком любого числа системных и программируемых свойств.	Символы с программируемыми свойствами.	Атомы, сопровождаемые метаданными вместо списка свойств
Списки свойств	Можно любые свойства вводить и программировать многократно.	Системные свойства функций и переменных могут изменяться присваиваниями.	Однократное вхождение системных свойства можно изменять. Программируемые свойства можно вводить многократно.	Системные свойства функций и переменных могут изменяться, программироватьс я макротехникой. Другим символам можно программировать свои свойства	Системные свойства функций и переменных неизменяемы, они хранятся в метаданных. Понятие атома трактуется как указательная переменная для доступа к разным программируемым стратегиям и дисциплинам обработки данных.
Иерархия	Вызовы функций	Определения функций	Вызовы функций и определения функций	Определения функций	Вызовы функций и определения функций
Пространства имён	Приоритет локальным и динамике	Приоритет статике — лексической области видимости	Разнообразие пространств имён. Взаимодействие лексических и динамических областей	Приоритет статике — лексической области видимости	Приоритет лексической области видимости, но доступна и динамика
Рецепты/ замыкания функций	При вычислении динамический рецепт по вложенности вызовов функций	При компиляции лексическое замыкание по вложенности определений	При компиляции лексическое замыкание по вложенности определений с использованием пространств имён	При компиляции лексическое замыкание по вложенности определений	При компиляции лексическое замыкание по вложенности определений
eval	Универсальная базовая функция доступна в любой момент.	Объявлена опасной, вынесена в отдельную	Универсальная базовая функция доступна в любой момент, но без	Универсальная базовая функция доступна, но предостерегают,	Универсальная базовая функция\ объявлена основной.

		библиотеку.	ассоциативного списка.	что она затрудняет отладку.	
Значение «ложь»	Nil = ()	#f	Nil = ()	#f, Nil, ()	false, Nil
Гомоиконность	Представление AST доступно в любое время, хранится независимо от компиляции	Представление AST теряется при принудительной компиляции	Представление AST доступно в любое время, хранится независимо от компиляции	Представление AST доступно после компиляции с помощью бэктрэкинга.	Представление AST доступно в любое время
Компиляция - **compile**	Компиляция функций по мере необходимости, исходный код сохраняется	Принудительная компиляция программы при чтении с потерей исходного кода	Компиляция функций по мере необходимости, но возможна и полная исходный код сохраняется,	Принудительная компиляция программы при её чтении без потери исходного кода	Автоматическая компиляция на лету с привлечением JVM и JIT.
GC – «Сборка мусора»	Автоматическая GC и вызов из программы	Только автоматическая GC без вызова из программы	Автоматическая GC и вызов из программы	Автоматическая GC и вызов из программы	Автоматическая GC и вызов из Java-библиотеки
присваивания и адреса	На периферии внимания	Базовая функция	На периферии внимания	Базовая функция	Кроме присваиваний используются структуры данных с управляемыми состояниями.
Print, Read – ввод-вывод, **I/O, ОС**	Функции взаимодействия с устройствами вырабатывают результаты, удобные для комбинаторики действий.	Процедуры взаимодействия с устройствами без выработки результата.	Функции взаимодействия с устройствами, включая любые файлы, вырабатывают результаты.	Развитый комплект средств без выработки результата.	Развитый комплект средств с выработкой результата, для вывода это Nil.
Раскрутка компилятора и интерпретатора	Компилятор и интерпретатор определены и отлажены на языке Lisp, затем закодированы.	Учебные проекты	Компилятор и интерпретатор определены на Common Lisp и C.	Ряд диалектов на самом языке.	Имеется самоопределение с декомпозицией по этапам компиляции.

Можно видеть тенденцию движения структур данных из неявной прагматики на уровень семантики в виде функций доступа, затем на уровень синтаксиса с введением специальных символьных форм. Другие вариации связаны с отношением к неизменяемости данных, этапом обработки программы и программированием стратегии или дисциплины доступа к общим данным. Определённые вариации связаны с реализацией значения «ложь» и разными подходами к организации параллельных процессов.

Нередко при сравнении языков программирования выстраивают довольно объёмные таблицы.[72] Отдельные штрихи вопросов сравнения языков программирования отмечены в источниках по другим языкам программирования и учебной литературе, включая бенчмарки для экспериментальной проверки отдельных спорных утверждений [75-89]. Интересно такой же анализ выполнить для наследников языка Lisp, а также для других семейств языков программирования. Представленные результаты анализа и сравнения диалектов языка Lisp образуют основу для определения номенклатуры семантических систем ядра языков функционального программирования.

Литература и источники

1. McCarthy John. Recursive Functions of Symbolic Expressions and Their Computation by Machine. Communications of the ACM, April 1960.
2. http://www-formal.stanford.edu/jmc/history/lisp/node2.html — LISP prehistory - Summer 1956 through Summer 1958.
3. McCarthy J. LISP 1.5 Programming Manual.- The MIT Press., Cambridge, 1963, 106p.
4. Lisp 1. Programmer's Manual. Massachusetts, 1960. 156 p.
5. http://www-formal.stanford.edu/jmc/history/lisp/node3.html — The implementation of LISP
6. http://www-formal.stanford.edu/jmc/history/lisp/node4.html — From LISP 1 to LISP 1.5
7. http://www-formal.stanford.edu/jmc/history/lisp/node5.html — Beyond LISP 1.5
8. Mitchell, R.W., "LISP 2 Specifications Proposal", Stanford Artificial Intelligence Laboratory Memo No. 21, Stanford, Calif., 1964.
9. http://www-formal.stanford.edu/jmc/index.html — John McCarthy's Home Page
10. http://www-formal.stanford.edu/jmc/mcc59.pdf — John McCarthy PROGRAMS WITH COMMON SENSE
11. Stoyan H. The Influence of the Designer on the Design – J. McCarthy and LISP. *Artificial Intelligence and Mathematical Theory of Computation*, p.409-426, Academic Press, 1991.
12. https://www.gnu.org/gnu/rms-lisp.html — Запись речи Ричарда Столмена на Международной конференции по Лиспу, 28 октября 2002
13. https://dl.acm.org/doi/10.1145/154766.155373 Guy L. Steele, Jr., Richard P. Gabriel The evolution of Lisp http://www.dreamsongs.com/NewFiles/Hopl2.pdf - Steele G.L.Jr, Gabriel R.P. The Evolution of Lisp
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. https://blog.cleancoder.com/uncle-bob/2017/07/11/ Martin, Robert (2017-07-11). "Pragmatic Functional Programming". CleanCoder.com. https://habr.com/ru/articles/478354/ (рус)
16. Functional Programming Languages in Education. LNCS 1022. Springer
17. Хендерсон П. Функциональное программирование. Применение и реализация: Пер. с англ.—М.: Мир, 1983.—349 с.
18. Филд А., Харрисон П. Функциональное программирование. Перевод под редакцией В.А.Горбатова. – М. Мир, 1993, 638с
19. Лавров С.С. Функциональное программирование. // Компьютерные инструменты в образовании. – 2002, N 2-4.
20. Городняя Л.В., Лавров С.С. Функциональное программирование. Принципы реализации языка Лисп. // Компьютерные инструменты в образовании. – 2002, N5, с. 49-58
21. http://www.intuit.ru/studies/courses/29/29/info - Городняя Л.В. Основы функционального программирования. Курс лекций. Учебное пособие. Серия «Основы информационных технологий» /М.: ИНТУИТ.РУ «Интернет-университет Информационных Технологий», 2004. – 280 с.
22. Городняя Л.В. Функциональный подход к описанию парадигм программирования //Новосибирск, ИСИ СО РАН. Препринт 152. 2009. 66 с.
23. Городняя Л.В. Функциональный подход к системному представлению прикладных программ учебного назначения. Новосибирск. (Препр./РАН, Сиб. отд-ние. ИСИ; N 16), 1993, 25 с.
24. Булычев Д. Снова о функциональном программировании. – М.: Открытые системы. N 5 (73), 2002, с. 38-44
25. Стивен Р. Палмер, Джон М.Фелсинг. Практическое руководство по функционально-ориентированной разработке ПО. − М.: Вильямс, 2002. − 299 с.
26 Лавров С.С., Силагадзе Г.С. Входной язык и интерпретатор системы программирования на базе языка ЛИСП для машины БЭСМ-6. - М.: ВЦ АН СССР, 1967.
27. Лавров С.С., Силагадзе Г.С. Входной язык и интерпретатор системы программирования на базе языка ЛИСП для машины БЭСМ-6. - М.: ИТМ и ВТ АН СССР, 1969.
28. Лавров С.С., Силагадзе Г.С. Автоматическая обработка данных. Язык Лисп и его реализация. - М.: Наука, 1978, 176 с.
29. Юфа В.М. Развитие системы программирования ЛИСП ВЗСМ-6, обработка символьной информации, вып. I, М.: ВЦ АН СССР, 1973.
30. Городняя Л.В. Реализация Лисп-интерпретатора. ВЦ СО РАН СССР, Новосибирск, 1974, - с. 24-35
31. Лозовский В.С. 0 некоторых аспектах человеко-машинного диалога. Изв. АН СССР, тех.кибернетика, 1981, с.147-156.
32. http://www.iis.nsk.su/news/events/mccarthy/ - Поздравление Дж. Маккарти с 75-летием на сайте ИСИ СО РАН
33. Юфа В.М. О новых функциях в системе ЛИСП - ВЗСМ-6. // В сб. «Обработка символьной информации», вып. 4, М.: ВЦ АН СССР, 1978, с. 26-50.
34. Пантелеев А. Г. Об интерпретаторе с языка Лисп для ЕС ЭВМ. — Программирование, 1980, No 3, с. 86-87
35. Семенова Е. Т., Чернов ПЛ. Использование диалоговой системы LISP/EC в учебном процессе. Тр./ Моск. энерг. ин-та, 1981, вып. 525, с. 23 — 25.

36. http://homelisp.ru/ - Файфель Б.Л. Рабочая версия системы программирования для языка Homelisp
37. http://www.intuit.ru/studies/courses/1026/158/info - Городняя Л.В., Н.А.Березин Введение в программирование на Лиспе. – М.: Интернет-Университет Информационных технологий. -: http://www.intuit.ru, 2007
38. R. Kent Dybvig The Scheme Programming Language - https://www.scheme.com/tspl4/ -
39. http://cemerick.com/2009/03/24/why-mit-now-uses-python-instead-of-scheme-for-its-undergraduate-cs-program/
40. Хьювенен Э., Сеппанен Й. Мир Лиспа., т.1,2, М.: Наука, 1994
41. Грэм П. ANSI Common Lisp. – Пер. с англ. – СПб.: Символ-Плюс, 2012. – 448 с., ил. ISBN 978-5-93286-206-3
42.Paul Graham: *ANSI Common Lisp*, Prentice Hall, 1995, ISBN 0-13-370875-6, Graham P. ANSI Common Lisp. //Prentice Hall, 1996. – 432p
43. http://machine-building.conf.nstu.ru/wp-content/uploads/2013/11/ANSI-Common-Lisp.pdf
44. http://www.clisp.org/ - GNU Clisp
45. http://www.lisp.org/HyperSpec/FrontMatter/ - ANSI стандарт Common Lisp
46. http://www.paulgraham.com/onlisptext.html - Интересные материалы по Clisp Пола Грэма, автора книги по стандарту ANSI Clisp
47. http://www.cons.org/cmucl/ - Домашняя страница CMUCL
48. Хейфец, А. Л. Инженерная компьютерная графика. AutoCAD // СПб. : БХВ-Петербург, 2007. - 316 с.
49. Гладков С.А. Программирование на языке автолисп в системе САПР Автокад – М.: «Диалог-МИФИ», 1991. – 96 с.
50. Lisp as an Alternative to Java. URL: https://flownet.com/gat/papers/lisp-java.pdf
51. Городняя Л.В. Некоторые диалекты Лиспа и сферы их приложения. - В сб. Трансляция и преобразования программ. - Новосибирск, 1984, с. 60-71
52. Paul Graham On Lisp *(англ.)* https://paulgraham.com/onlisp.html
53. Graham, Paul What Make Lisp Different https://paulgraham.com/diff.html
54. Graham, Paul Yackers & Painters/ https://paulgraham.com/hackpaint.html
55. Graham, Paul Lisp: побеждая посредственность (рус.) https://nestor.minsk.by/sr/2003/07/30710.html
56. Graham, Paul https://news.ycombinator.com/item?id=33983232 Graham, Paul (2016-05-06). "Paul Graham on Twitter". Twitter.com.
57. О logo // https://web.archive.org/web/20041013130519/http://www.int-edu.ru/logo/logo.html
58. The Racket Reference https://docs.racket-lang.org/reference/ -
59. Rich Hickey Simple Made Easy - *https://habr.com/ru/articles/496802/* (с русским переводом)
60. Rich Hickey "[ANN] dotLisp: A Lisp dialect for .Net" https://groups.google.com/g/comp.lang.scheme/c/ibf6C-C6V66o?pli=1 (2002-10-16).
61. https://ru.hexlet.io/blog/posts/clojure - Как устроен функциональный диалект Лиспа Clojure и почему использующие его программисты восхищаются им
62. Введение в Clojure - https://alexott.net/ru/clojure/clojure-intro/
63. Алекс Отт. Clojure, или «Вы все ещё используете Java? Тогда мы идем к вам!» в 4 выпуске журнала «Практика функционального программирования» (Обновленная версия статьи, с описанием версии 1.2) - - https://alexott.net/ru/clojure/clojure-intro/index.html#sec2
64. Описание языка Clojure - https://cdn.oreillystatic.com/oreilly/booksamplers/9781449394707_sampler.pdf
65. Описание языка Clojure - https://clojure.org/reference/reader
66. A History of Clojure - https://clojure.org/about/history
67. Differences Clojure with other Lisps - https://clojure.org/reference/lisps
68. https://stackoverflow.com/questions/6008313/clojure-vs-other-lisps
69. https://zeemly.com/compare/clojure-vs-lisp
70. Notes on Common Lisp VS Clojure https://gist.github.com/vindarel/3484a4bcc944a5be143e74bfae1025e4
71. One Major Difference Between Clojure and Common Lisp - https://news.ycombinator.com/item?id=10206827
72. Сравнение по 11 признакам более 100 языков - https://rosettacode.org/wiki/Language_Comparison_Table
73. Маурер У. Введение в программирование на языке ЛИСП. – М.: Мир, 1976.
74. Оллонгрен А. Определение языков программирования интерпретирующими автоматами. Пер.с англ. Серия: Математическое обеспечение ЭВМ. М. Мир 1977г. 288 с. ил.
75. Лавров С.С. Расширяемость языков. Подходы и практика. В сб.: Прикладная информатика, вып. 2. М.: Финансы и статистика, 1984, с. 17 — 22.
76. Городняя Л.В. Сравнение учебных языков программирования Бейсик, Паскаль и Рапираа // ИнфО, 1990
77. https://rapirareborn.com/ РАПИРА
78. Евстигнеев В.А., Городняя Л.В., Густокашина Ю.В. "Язык функционального программирования SISAL" // Интеллектуализация и качество программного обеспечения, Новосибирск, 1994, с. 21-42
79. Сошников Д. В. Программироване на F#. – М.: ДМК Пресс, 2011. – 192 с.
80. http://www.marstu.mari.ru/mmlab/home/lisp/title.htm – Дистанционный учебник М.Н.Морозова

81. http://grimpeur.tamu.edu/~colin/lp/ - Небольшой учебник для начального знакомства с Лиспом
82. Сергеев Л.О. Удивительный мир языка Лисп. Введение в язык и задачи, задачи, задачи. // Информатика – 2000, N 29.
83. Городняя Л.В. Об одном подходе к синтезу транслятора на примере языка Литтл. // Теория и практика системного программирования. − Новосибирск, 1977. − С. 60-71.
84. Городняя Л.В. Макетирование программ с помощью тестов и описаний. //В сб «Языки и системы программирования», Новосибирск, 1981, с. 115-123.
85. Городняя Л.В. От дискретной математики к семантике языков программирования // Всероссийская научная конференция «Математические основы информатики и информационно-коммуникационных систем»
86. Gouy, Isaac. The Computer Language Benchmarks Game. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/.
87. https://www.jdoodle.com/ - доступ примерно к 70 компиляторам, включая рассмотренные диалекты языка Lisp
88. https://learnxinyminutes.com/docs/common-lisp/ - Learn X in Y minutes Where X=Common Lisp.
89. https://paulgraham.com/bel.html
90. https://alexott.net/ru/fp/books/