Сравнение диалектов языка Lisp

Л.В. Городняя[1, 2]

[1] *Институт систем информатики им. А.П. Ершова СО РАН*
[2] *Новосибирский государственный университет*

Аннотация. Статья посвящена результатам анализа и сравнения особенностей языка Lisp и наиболее успешных его диалектов (Scheme, Common Lisp, Racket, Clojure). Показаны их сходства и различия с оценкой эволюции и энтропии знания в истории языков программирования на примере рассмотренных языков.

Ключевые слова: Lisp, Scheme, Common Lisp, Racket, Clojure

The result of comparison of dialects of the Lisp language

L.V.Gorodnyaya[1,2]

[1] *A.P. Ershov Institute of Informatics Systems*
[2] *Novosibirsk State University*

Abstract. The article 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). 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.

Keywords: Lisp, Scheme, Common Lisp, Racket, Clojure

1. Введение. Статья представляет результаты очередного этапа разработки методики анализа и сравнения языков и парадигм программирования, продолжающейся в лаборатории информационных систем ИСИ СО РАН им. А. П. Ершова в рамках тематики, связанной с исследованием средств и методов преподавания программирования [1]. Начинается проверка методики на конкретных долгоживущих языках программирования (ЯП). Принято во внимание, что термин «язык программирования» в речевой практике понимается как «входной язык системы программирования, обеспечивающей доступ к определённым

аппаратным средствам» (ЯиСП). Поэтому сравниваются не только конструкции собственно ЯП, но и реализационные структуры данных и пространства доступных процессов обработки данных в типовых ЯиСП. Учтено, что ЯП всегда опирается на ряд неявных конструкций, необходимых для реализации системы программирования (СП) и воспринимаемых в практике как неотъемлемая часть языка, превращающегося в целостную ЯиСП.

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

В данной статье освещены результаты сравнения языка Lisp с его диалектами, показывающие, что унаследовано, что изменено и что дополнено. Выбор языка Lisp для первого эксперимента обусловлен не только чёткостью его описания [2], но и ростом интереса к функциональному программированию (ФП), регулярно происходящем при смене элементной базы. Кроме того, Lisp можно характеризовать как язык программировагния одновременно и низкоуровневый, и сверх высокого уровня в зависимости от уровня решаемых задач, причём, адаптирующийся к решению новых задач и изобретению лаконичных и эффективных решений, возможно не соотвествующих шаблонам, навязываемым большинством более популярных языков.

Эксперимент, выполненный на материале диалектов Pure Lisp, Scheme, Common Lisp, Racket и Clojure [2, 3, 4, 5, 6-8], появившихся с шагом 10 лет, показал различие целей создания диалектов и особенности достижения целей при минимальных изменениях семантики и прагматики исходного языка Lisp. Более заметны изменения на уровне лексикона и синтаксиса. При сравнении языка Lisp с его диалектами уделено внимание своду семантических и прагматических принципов ФП и расслоению определений на базис, расширение, средства диагностики (границ вычислимости) и отладки программ, а также связи с внешним миром.

Изложение начинается с краткой справки о языке Lisp, показавшем разницу между новыми задачами компьютерной обработки любой информации и традиционными задачами обработки чисел, и диалекте Pure Lisp, предназначенным для ознакомления с идеями языка Lisp, затем приведены сведения о диалектах Scheme, Common Lisp, Racket и Clojure в порядке их появления. Scheme создал прецедент эффективной реализации ФП, Common Lisp поддержал продуктивность производственного применения ФП, Racket сформировал систему обучения созданию и реализации ЯП, Clojure обеспечил полноценное применение ФП в новых

ИТ. Далее сформулированы выводы о замеченных особенностях формирования диалектов языка Lisp с учётом материалов по языкам ФП, наследующим идеи языка Lisp, таким как F# и Haskell [9, 10, 11]. Это достаточное основание рассматривать Lisp как базовую математику функционального программирования. Для оценки особенностей диалектов использовались примеры реализации отдельных конструкций языка Lisp и его диалектов, проверенные на системе homeLisp[1], платформе jdoodle.com[2] и некоторых онлайн-компиляторах.

2. Немного истории языка Lisp

Идеи языка Lisp при его появлении сразу вызвали ревнивую критику как со стороны программистов, так и со стороны математиков. Математиков смущала противоречивость некоторых построений с точки зрения классической математики, например, различие контекстов определения, вызова и вычисления функций[3]. Программисты не могли смириться с отсутствием в языке привычных понятий, начиная с «изменения состояний памяти», а также с непредсказуемо медленной обработкой списков в сравнении с быстрой обработкой векторов. В середине 1970-ых годов Дана Скотт (Dana S. Scott) опубликовал конструктивную теорию, смягчившую критицизм математиков, построив первую непротиворечивую модель бестипового λ-исчисления[4]. Скептицизм программистов оказался много устойчивее. Например, предрассудок, что Lisp — это интерпретируемый язык, скорее всего, связан с тем, что реализации языка Lisp обычно предоставляют диалоговый стиль работы с программой, при котором не заметно, что фрагменты программного кода компилируется по мере необходимости. Это заблуждение не исчезло при появлении в 1976 году диалекта Scheme, использующего совмещение чтения программы с её полной принудительной компиляцией, сохраняя диалог, и добавляя векторную реализацию списков.

Исторически первой реализацией языка Lisp, включающей все базовые элементы языка, был интерпретатор, работавший на IBM 704, появившийся в октябре 1958 года. 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. Lisp позволяет создавать программы, динамически порождающие код, выполнять любые реализационные трюки, строить

---

[1] http://homelisp.ru/

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

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

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

виртуальные машины, специализированные системы, диалекты и расширяющие язык пакеты.

Такой потенциал языка Lisp сложился как ответ на вопрос: «*НАСКОЛЬКО* новые задачи компьютерной обработки информации отличаются от традиционных задач обработки чисел?». Основные тезисы ответа представлены следующими утверждениями.

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

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

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

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

Концепции языка Lisp со временем кристаллизовались как парадигма функционального программирования [12][5], хотя реализация языка изначально поддерживает основные императивные черты ради привлечения опытных программистов и возможности повышать надёжность и эффективность программ.

Для быстрого ознакомления с идеями языка Lisp Дж. Маккарти выделил семантический базис — диалект Pure Lisp, включающий в себя пять функций обработки списков (CONS, CAR, CDR, EQ, ATOM), четыре конструктора (QUOTE, COND, LAMBDA, LABEL) и универсальную функцию EVAL, способную вычислить любое правильно представленное списком выражение, что определяет границы вычислимости без использования семантики изменения состояний памяти, без глобальных переменных. Пары функций QUOTE и EVAL достаточно для поддержки разных схем вычислений, ленивых вычислений, оптимизации программ и любой макротехники как основы метапрограммирования, включая определение интерпретаторов и компиляторов.

Диалект Pure Lisp дал ответ на вопрос «*Какой* может быть методика обучения программированию на языке Lisp?». Ответ выглядит следующим образом:

---

5 https://alexott.net/ru/fp/books/ Обзор литературы о функциональном программировании

— выделить краткую базовую семантику, достаточную для определения остальных конструкций языка (выражения, ветвления, рекурсивные функции);

— дать примеры их символьного представления и применения при решении знакомых задач;

— показать типовые решения некоторых задач с помощью отображений, ленивых вычислений и метапрограммирования;

— привести определения интерпретатора и компилятора на изучаемом Pure Lisp на уровне калькулятора;

— расширить определения Lisp-калькулятора до обобщённого интерпретатора.

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

- универсальность символьных представлений;
- независимость параметров функции;
- само-применимость определений (рекурсия);
- гибкость распределения памяти;
- неизменяемость данных;
- единственность результата функции.

Дж. 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 [13]. Рассказывая о языке Lisp, Дж. Маккарти подчёркивал, что экспериментируя программист может изменять в языке Lisp всё что угодно, кроме константы Nil. К 1962 году была готова версия «Lisp 1.5» и описание реализации системы, ставшей преемником самого раннего языка Lisp [2]. Описание языка стало основой для создания Lisp-систем как в США, так и за её пределами, в нашей стране на БЭСМ-6, ЕС ЭВМ, СМ-4 и других машинах[6]. Составные данные языка Lisp на уровне синтаксиса выглядят как списки элементов любой природы, хотя неявно в СП на уровне прагматики языка поддержаны 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 и пространств имён, а вектора используются для сопряжения со встроенными машинными процедурами.

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

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 ФП [14]. Вскоре появился Lispkit — реализация чисто функционального диалекта Pure Lisp с лексической областью видимости, разработанного в качестве испытательного и учебного стенда при изучении концепций ФП, включая ранние эксперименты с ленивыми вычислениями [15, 16]. Полученные компилятор и виртуальная машина были легко переносимы.

В 1974 году в Xerox началась разработка аппаратуры и системы машинных команд для аппаратной реализации языка Lisp. Язык Scheme был разработан в 1976 году в MIT в рамках проекта по созданию Lisp-машин [3]. Появилось доказательство, что так называемая «неэффективность языка Lisp» обусловлена не свойствами языка, а особенностями компьютеров и методов реализации ЯП.

В 1984 году, через 10 лет после Scheme, появился диалект Common Lisp — мультипарадигмальный язык общего назначения, дополняющий традиционные динамические решения языка Lisp механизмами статического связывания переменных и раздельных пространств имён, программирования макросов, функционалов, пакетов и лексических замыканий функций [4]. В 1995 году Common Lisp был стандартизован ANSI.

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

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

3. Диалект Scheme для эффективной реализации

Язык Scheme был разработан в 1976 году в MIT в рамках проекта по созданию Lisp-машин [3]. Для своего времени Lisp-машины были одними из мощнейших компьютеров в классе персональных рабочих станций (около 7 тыс. штук во всём мире на 1988 год).

Scheme дал ответ на вопрос «*КАК* сделать функциональное программирование столь же эффективным как императивное?». Ответ включает следующее:

— Ограничить универсальность символьных вычислений отказом от представления значения любой природы с помощью атомов и программируемых списков свойств символа, оставить только системные свойства переменных и функций.

— За основу реализационных 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 (AST), совместить с принудительной компиляцией, а интерпретатор EVAL вынести из базиса во вспомогательную библиотеку.

— Макротехнику ограничить выполнением на этапе компиляции, что сводит её потенциал к привычным возможностям препроцессоров во многих ЯиСП.

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

— Смягчить неизменяемость данных, опираясь на унификацию присваиваний и определений функций, выполненную к тому времени при разработке языка Algol 68, а заодно и разрешить изменять значения системных свойств символа (define = set![7]) и даже элементов точечной пары (set-car!, set-cdr!) — эквивалентов rplaca, rplacd языка Lisp.

Язык Scheme заимствовал терминологию и синтаксис языка Lisp, несколько изменяя смысл ряда понятий и сужая трактовку почти всех принципов, из характерных особенностей языка Lisp в языке Scheme поддержана примерно треть. Семантика вычислений в Scheme немного дополнена семантикой изменения состояний памяти для наименований функций и глобальных переменных. Переход к векторам пошатнул позиции пустого списка и атома Nil — реализация пустых векторов в те годы не практиковалась. Принудительная компиляция программы, не сохраняющая исходное AST, затрудняет возможности динамической оптимизации программ и процессов. В основном сохранены принципы само-определения, независимости параметров и гибкости границ блоков памяти. Роль предиката может выполнять любая функция, в качестве значения «истина» допускается любое данное кроме значения «ложь»8, представленного специальным значением #f. Можно считать, что так выделилось минимальное ядро грамматики языка Lisp. Теперь существуют реализации Scheme на JVM. С небольшой натяжкой можно сказать, что

---

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

[8] На практике это означает, что 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».

язык Scheme похож на кругло-скобочный Pascal в пределах структурного программирования, доминировавшего в те годы.

5. Common Lisp для производственного применения

Common Lisp был разработан в начале 1980-ых с целью объединения полезных механизмов большого числа разрозненных диалектов языка Lisp [4]. Common Lisp часто сравнивают и противопоставляют языку Scheme — это два самых популярных диалекта Lisp. Scheme предшествовал Common Lisp и исходил не только из той же традиции Lisp, но и от тех же разработчиков: Гай Стил, вместе с которым Джеральд Джей Сассман разработал Scheme, возглавлял комитет по стандартизации Common Lisp, в котором преодолено сужение потенциала языка Lisp. Сохраняя и восстанавливая основные концептуальные решения языка Lisp, диалект Common Lisp их дополняет, расширяет пространство допустимых процессов.

Common Lisp иногда называют Lisp-2, а Scheme — Lisp-1, имея в виду использование CLISP отдельных пространств имен для функций и переменных[9]. 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 программ на нужном этапе. В системе CLISP реализован пакет CLOS, дающий полную поддержку популярной парадигмы ООП.

Common Lisp поддерживает функциональное, императивное, рефлексивное, реляционное и объектно-ориентированное программирование (пакет CLOS), а также метапрограммирование 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, а также фиксации специализированных версий языка. Этот диалект резко расширил сферу производственного применения языка Lisp, используя в лексиконе семантику доступа к матрицам, хэш-таблицам, программируемым структурам данных, подобным структурам в языке C, и мультизначениям, удобными для моделирования потоков. Всё это внешне представляется как списки, но реализовано как эффективные структуры данных, доступные через функции. Массивы могут содержать любой тип значений в качестве элемента, смешивать разные типы в одном массиве, или могут быть специализированы, содержать только определённый тип.

---

[9] 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.

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

— Универсальность символьной обработки и структур данных неограниченного размера дополнена средствами обработки конечных чисел и структур данных, типичных для большинства ЯП и приложений.

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

— Самоопределение в форме рекурсии обогащено разнообразием схем циклов, 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 уточняется введением понятия «поле» для работы с изменяемыми системными свойствами символа. Для потребляющих много памяти функций предоставляются их деструктивные аналоги, приспособленные к более эффективной обработке данных (conc-nconc, subst-nsubst, reverse-nreverse, union-nunion, mapcar-mapcon и др.)

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

Введены понятия поле, пакет и мультизначения. 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 операционной 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.

Common Lisp используется для разработки исследовательских приложений в области искусственного интеллекта (ИИ), при быстрой разработке прототипов или для развернутых приложений, используется во многих коммерческих приложениях, включая Yahoo. 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 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).

6. Racket для создания новых языков программирования

В 2010 году название Racket получила очередная версия учебной среды программирования на языке Scheme, разрабатываемая с 1995 в Университете Дьюка для обучения созданию, разработке и реализации новых языков программирования. Это мультипрадигмальный ЯП общего назначения, поддерживающий функциональное, языково-ориентированное, рефлективное, логическое, объектно-ориентированное, процедурное и метапрограммирование. Образовательная направленность повлияла на общую структуру языка Racket как систему диалектов, соответствующих уровням обучения. Это заодно позволило в реализации языка сохранить решения языка Scheme, принятые под прессом компьютерных характеристик середины 1970-ых годов, и дополнить системную поддержку языка Racket в соответствии со значительно усовершенствованными возможностями современной элементной базы и новыми требованиями ИТ.

Диалект Scheme был сознательно ограничен по возможностям в сравнении с языком Lisp, жертвуя универсальностью, гибкостью и продуктивностью ради простоты языка и эффективности СП. Он оказался не удобен для промышленных проектов. Разработчики диалекта Racket выявили ключевые недостатки Scheme, затрудняющие разработку крупных и надежных систем, а именно, отсутствие модульной системы, слабую поддержку обработки исключений, метапрограммирования и расширяемости, ограничения в типизации и структурах данных, отсутствие способов для построения безопасных и масштабируемых систем. Кроме того, в Scheme гигиеничные макросы менее выразительны, чем специальные функции в языке Lisp или макросы Common Lisp. Для преодоления таких недостатков Racket, при развитии языка Scheme, был создан как диалект языка Lisp и используется для реализации ЯП, компиляторов и интерпретаторов, образовательных систем и платформ, языков для обучения и преподавания, включая обучающие и экспериментальные языки, Оставаясь наследником Scheme, Racket делает ставку на практичность, расширяемость и богатство инструментов, уделяя больше внимания удобству использования и обучения, поддержку метапрограммирования и инструментальных средств для разработки

ЯиСП. Racket включает макросы и на этапе компиляции, и фазы выполнения с удобными функциями для разработки и отладки. Поддержан диалект Scribble — язык разметки для документации и ряд диалектов, связанных с основными парадигмами программирования, проблемно-ориентированными языками (DSL) и приложениями ИТ.

Производительность Racket обеспечена JIT-компилятором и механизмом сборки мусора с поддержкой поколений объектов. Включена поддержка мелкозернистого параллелизма. Появились версии Minimal Racket без пакетов, поддержка байт-кода и JIT-компиляции для архитектуры ARM, а также быстродействующий Typed Racket и другие диалекты. Имеется собственная виртуальная машина. В экспериментах по разработке разных версий Racket обнаружилось, что компиляция не всегда повышает производительность программ, но иногда способствует продуктивности программирования[10]. Система макросов в Racket используется для создания полных языковых диалектов, затрагивая семантику.

Racket отвечает на вопрос «*КТО* будет определять лицо программирования в будущем?». Ответ достигается следующими решениями:

— Универсальность символьных вычислений распространена на приобретение профессиональных навыков, включающих разработку документации, что соответствует предложенной Д. Кнутом парадигме грамотного программирования (literate programming)[11].

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

— Независимость параметров поддержана на уровне сопоставления с образцами, достаточными для представления грамматик с переводом (БНФ), образцы могут сопоставляться в произвольном порядке за исключением образца «Любое данное», выбираемое лишь если никакой другой образец не подтверждён12.

---

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

11 http://www.literateprogramming.com/ — методика профилактики кризиса ухода исполнителя и разбухания описаний, создаваемых техническими писателями. Не исключено, что методика не стала массовой из-за высокого темпа развития ИТ. Возможно Д. Кнут хотел привлечь внимание к тому, что искусство программирования основывается и на владении естественным языком.

12 Подобно решению про пустую сроку в БНФ, варианту ESC в логическом программировании и пустому переходу в сетях Петри.

— 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.

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

Racket примерно на треть наследует решения языка Scheme и на две трети возвращается к исходным решениям языка Lisp. Совмещение компиляции с чтением программы допускает восстановление символьной 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.

7. Clojure — новые горизонты ИТ

Clojure — современный диалект языка Lisp, появился в 2007 году, разработан Ричем Хикки (Rich Hickey), независимым разработчиком ПО, ранее разработавшим dotLisp в рамках проекта .NET Framework. Clojure наследует особенности языка Lisp, обеспечивающие гибкое и мощное метапрограммирование, поддерживает синтаксическое расширение [6-8] и надежную семантику, дополненную механизмами для программирования приложений на базе распределённых и параллельных процессов.

В этом языке определения функций могут неформально сопровождаться пред- и пост-условиями, что помогает проверке утверждений об аргументах и полученных значениях, а также позволяет при тестировании проверять инварианты функций и использовать внешние методы верификации программ. Clojure, как язык программирования, не предоставляет встроенных методов верификации программ, но для обеспечения корректности программ используются различные подходы и инструменты, включающие средства типизации и спецификации (clojure.spec, malli), динамический контроль данных и тестирование **(**clojure.test, test.check**),** помогающие обнаруживать неожиданности. Возможна интеграция с внешними инструментами для формальной

верификации с возможностью автоматического тестирования свойств, такими как Rosette, Z3, SMT-солвер (satisfiability modulo theories) или ACL2, проверки типов данных (clj-kondo, Eastwood), а также статические анализаторы для проверки безопасности кода или соблюдения определённых стандартов.

Динамическая типизация означает, что типы переменных проверяются во время выполнения, это обеспечивает гибкость и быструю разработку, так как не требуется явного объявления типов. Clojure уделяет большое внимание тестированию (clojure.test, test.check, midje, kaocha, Unit Testing, Integration Testing, Property-based testing), вызывающему доверие у практиков.

Статический анализ выявляет потенциальные ошибки, нарушения стиля кодирования, неиспользуемый код, поддерживает автодополнения для более раннего обнаружения ошибок. Существует регулярный рефакторинг кода (улучшение структуры кода без изменения его функциональности) помогает поддерживать код в хорошем состоянии, упрощает понимание и уменьшает вероятность ошибок. Доступны библиотеки, позволяющие определять контракты (core.contracts, lucid.policy) для функций (пред-условия, пост-условия, инварианты) и автоматическая генерация документации.

Возможна явная 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, деструктурированные на списки символов.
- Списки аргументов функций.

Деструктуризация означает переход от ранее созданной структуры данных к структуре с другим методом доступа при сохранении её наполнения[13], допускается использование подчеркивания «_» для обозначения игнорируемой позиции. Исходная структура сохраняется в соответствии с принципом неизменяемости данных.

Деструктуризированная последовательность аргументов функции позволяет вместо имён связанных переменных использовать нумерацию аргументов, список которых можно рассматривать как вектор[14]. К первому аргументу функции можно обращаться, просто используя «%».

Параллельное программирование использует транзакционную память как в базах данных, а также агентов и разные виды ссылочных переменных. Поддержаны ленивые последовательности, вспомогательные

[13] Похоже на реорганизацию векторов в языке APL.

[14] Подобно некоторым языкам заданий и макропроцессоров.

процессы и введено несколько неявных понятий для поддержки параллелизма и программирования своих структур данных с учётом проблем надёжности и безопасности.

В качестве компромисса между идеями чистого ФП и необходимостью изменения состояний для организации параллельных процессов введены ссылочные переменные — атомы рассматриваются как структуры, обеспечивающие разные дисциплины доступа к памяти и стратегий многопоточности (atom, ref, agent).

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

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

— Можно синхронизовывать выполнение потоков: откладывание, ожидание, обещание, передача, готовность, блокировка (delay, future, promise, deliver, is-done?, deref), контролировать ход вычисления, а также представлять эффективные формы циклов, не использующих стек.

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

— Единственность результата функции дополнена возможностью представления пред- и пост-условий, проверки утверждений об аргументах и полученных значениях, при тестировании можно проверять инварианты функций и использовать внешние методы верификации программ.

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. Атом 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 аргументов, что расширяет форматы вызова функций.

8. Итог сравнения диалектов

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, что показано в Таблице 1.

Таблица 1.

Вариации в статусе структур данных и понятий диалектов языка Lisp

Конструкции	**1958 Lisp**	**1976 Scheme**	**1984 Clisp**	**1995 Racket**	**2007 Clojure**
Вектора	Неявная прагматика параметров процедур	Базовая деструктуризация списков в вектора	Лексикон реорганизации списков в вектора и обратно	Синтаксис и базовая деструктуризация списков в вектора	Синтаксис и базовая деструктуризация списков в вектора
Хэш-таблицы	Неявная прагматика списка глобальных атомов и значений	Семантика областей видимости	Семантика пространств имён и пакетов и лексикон	Синтаксис и семантика хэш-таблиц	Синтаксис и семантика хэш-таблиц, варианты представления
Ассоциативный список атома	Семантика вычислений, программируемые свойства	Прагматика вычислений, системные свойства	Семантика вычислений, программируемые свойства	Прагматика вычислений, системные свойства	Синтаксис хэш-таблиц и прагматика вычислений
Атомы и символы	Символы, числа и строки произвольной длины. Атомы обладают списком программируемых свойств в любом количестве.	Числа, строки и коды произвольной длины. Символы представляют переменные и функции, нет программируемых свойств	Атомы, числа и строки произвольной длины с добавлением конечных чисел. Символом называется атом, обладающий списком свойств.	Числа, строки и коды произвольной длины. Символы с программируемыми свойствами.	Символы, числа и строки. Указательные переменные. Атомы, сопровождаемые метаданными вместо списка свойств.
Пространства имён	Приоритет локальным и динамике вызовов функций	Приоритет статике — лексической области видимости определений функций	Разнообразие пространств имён. Взаимодействие лексических и динамических областей	Приоритет статике — лексической области видимости	Приоритет лексической области видимости, но доступна и динамика
eval	Универсальная базовая функция доступна в любой момент	Объявлена опасной, вынесена в отдельную библиотеку	Универсальная базовая функция доступна в любой момент, но без ассоциативного списка	Универсальная базовая функция доступна, но предостерегают, что она затрудняет отладку	Универсальная базовая функция\ объявлена основной
compile	Компиляция функций по мере необходимости, AST доступно в любое время в виде списка исходного кода	Принудительная компиляция программы при чтении с потерей исходного кода	Компиляция функций по мере необходимости, но возможна и полная исходный код AST сохраняется,	Принудительная компиляция программы при её чтении без потери исходного кода — AST через бэктрэкинг.	Представление AST доступно в любое время. Автоматическая компиляция на лету с JVM и JIT.
Присваивания и адреса	На периферии внимания rplaca и rplacd	Базовая функция set!	На периферии внимания	Базовая функция	Используются структуры данных с управляемыми

					состояниями
Print, Read	Функции взаимодействия с устройствами вырабатывают результаты, удобные для комбинаторики выражений	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.

.Вариации структур данных и значений сводятся к границам между лексиконом, синтаксисом, семантикой и прагматикой языка, между периодом компиляции и выполнения программы, к реализации значения «ложь» и особенностям понятия «атом». Системные прагматические решения по обработке структур данных становятся доступными сначала на уровне семантики с помощью функций доступа, затем переходят на уровень конкретного синтаксиса, получают своё представление, что не нарушает их семантическую эквивалентность, т. к. AST диалектов имеет списочное представление. Семантика вычислений в диалектах по разному взаимодействует с семантикой изменения состояний памяти, как правило для именования функций.

Принципы и понятия уточняются в зависимости от продуктивности и критериев качества программирования в разных областях приложения. Для Scheme это сохранение привычки к присваиваниям, для Common Lisp — ценность разнообразия структур данных, для Racket — создание специализированных диалектов без нагромождения библиотек, для Clojure — освоение многопроцессорных комплексов и распределённых информационных систем.

Более подробное изложение результатов сравнения отражено в препринте [17].

Заключение

Вот и завершён обзор результатов сравнения наиболее популярных диалектов языка Lisp. Основная причина проведения такого сравнения связана с разработкой методики оценки продуктивности ЯиСП и программируемых решений в отличие от измерения эффективности и производительности программ. Полученные результаты образуют основу для определения номенклатуры семантических систем языков ФП для методики оценки продуктивности ЯП.

Другая причина — отклик на появление публикаций [18], содержащих утверждение, что Lisp умер, исчерпав свои возможности, и уступил новым более мощным ЯП.

При измерении мощности ЯП по характеристике пространства доступных процессов вычислений самыми мощными оказываются языки

ассемблера. Языки более высокого уровня, включая языки управления заданиями в операционных системах, теряют часть такого пространства ради удобства представления процессов обработки данных и управления ими.

Сравнение диалектов Scheme, Common Lisp, Racket и Clojure, последний из них появился в 2007 году, показывают, что в них сохранены основные возможности языка Lisp, реализационные структуры данных, базовые принципы и понятия с небольшими вариациями на злобу дня. Диалекты Scheme, Common Lisp, Racket и Clojure обладают реализацией на JVM, что говорит о их равномощности, они предоставляют одно и то же пространство процессов вычислений. Авторы этих диалектов чётко называют свои диалекты вариантами языка Lisp. Появление» названий «Racket» и «Clojure» не отменяет роль так названных диалектов в успешной адаптации языка Lisp к новым возможностям аппаратуры и новым поколениям программистов. Объявление языка Lisp умершим несколько предвзято.

Показанные в статье результаты сравнения диалектов языка Lisp позволяют заметить медленное смягчение программистского скептицизма по отношению к принятым в языке Lisp решениям по мере прогресса элементной базы и развития ИТ, Семейство Lisp теперь является одним из старейших и наиболее влиятельных семейств ЯП, его мощность в разных источниках, начиная с Википедий, оценивается от 500 до тысяч ЯП и диалектов. Кроме того, следует учесть языки ФП, наследующие основные идеи языка Lisp, они постоянно разрабатываются, улучшаются и появляются новые, их число также оценивается в сотни или тысячи.

Следующий эксперимент по сравнению ЯиСП предполагается выполнить на материале языков ФП, таких как Erlang, Sisal, F# и Haskell, рассматриваемых как наследники языка Lisp. Далее интересно выполнить сравнение представителей других долгоживущих семейств ЯП, в первую очередь это Fortran и C. Возможно удастся получить ответ на часто задаваемый недоумённый вопрос «Почему, в отличие от языков Fortran, C и Java, язык Lisp не обрастает стандартными библиотеками?». Предварительная гипотеза — гибкость языка Lisp позволяет вместо нагромождения библиотек оперативно создавать диалекты, более тонко, точно и продуктивно решающие новые задачи.

Литература

1. Городняя Л.В. О представлении результатов анализа языков и систем программирования. Научный сервис в сети Интернет: труды XX Всероссийской научной конференции (17-22 сентября 2018 г., г. Новороссийск). — М.: ИПМ им. М. В. Келдыша, 2018.
2. McCarthy J. McCarthy J., Abrahams P. W., Edwards D. J. et al. LISP 1.5 Programming Manual. The MIT Press, Cambridge, 1963. 106 p.

3. Kent R. Dybvig The Scheme Programming Language — https://www.scheme.com/tspl4/
4. Graham P. ANSI Common Lisp. //Prentice Hall, 1996. — 432 p.
5. The Racket Reference — https://docs.racket-lang.org/reference/
6. "Clojure Programming". OReilly.com. Retrieved 2013-04-30. https://cdn.oreillystatic.com/oreilly/booksamplers/9781449394707_sampler.pdf
7. Отт А. Введение в Clojure — https://alexott.net/ru/clojure/clojure-intro/
8. Differences Clojure with other Lisps — https://clojure.org/reference/lisps/
9. Сошников Д.В. Программирование на F#. — М.: ДМК Пресс, 2011. — 192 с.
10. Душкин Р.В. Функциональное программирование на языке Haskell / М.: ДМК Пресс,, 2008. — 544 с., ил. с. — 1500 экз. — ISBN 5-94074-335-8.
11. Официальный сайт языка Haskell — "О языке" http://haskell.org/aboutHaskell.html
12. 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/
13. Mitchell, R.W., "LISP 2 Specifications Proposal", Stanford Artificial Intelligence Laboratory Memo No. 21, Stanford, Calif., 1964.
14. Landin, P.J. (January 1964). "The Mechanical Evaluation of Expressions". Comput. J. **6** (4): 308—320. doi:10.1093/comjnl/6.4.308
15. Хендерсон П. Функциональное программирование. Применение и реализация = Functional Programming. — М.: Мир, 1983. — 349 с.
16. 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
17. Городняя Л.В. Lisp и его диалекты // Новосибирск, препринт, 2025 https://www.iis.nsk.su/repository/gorod.14408. *(будет доступен в середине августа)*
18. Биллиг В.А. Хроника языков программирования. Прошлое, настоящее, будущее: учебное пособие / В. А. Биллиг ; ИНТУИТ национальный открытый университет. — Москва: ИНТУИТ, 2024. — 286 с. — ISBN 978-5-9556-0204-2 https://intuit.ru/goods_store/books/10019?ysclid=mdqvqlwbyg377265958

References

1. Gorodnyaya L.V. O predstavlenii rezul'tatov analiza yazykov i sistem programmirovaniya. Nauchnyy servis v seti Internet: trudy XX Vserossiyskoy nauchnoy konferentsii (17-22 sentyabrya 2018 g., g. Novorossiysk). — M.: IPM im. M. V. Keldysha, 2018.
2. McCarthy J. McCarthy J., Abrahams P. W., Edwards D. J. et al. LISP 1.5 Programming Manual. The MIT Press, Cambridge, 1963. 106 p.

3. Kent R. Dybvig The Scheme Programming Language — https://www.scheme.com/tspl4/
4. Graham P. ANSI Common Lisp. //Prentice Hall, 1996. — 432 p.
5. The Racket Reference — https://docs.racket-lang.org/reference/
6. "Clojure Programming". OReilly.com. Retrieved 2013-04-30. https://cdn.oreillystatic.com/oreilly/booksamplers/9781449394707_sampler.pdf
7. Ott A. Vvedeniye v Clojure — https://alexott.net/ru/clojure/clojure-intro/
8. Differences Clojure with other Lisps — https://clojure.org/reference/lisps/
9. Soshnikov D. V. Programmirovane na F#. — M.: DMK Press, 2011. — 192 p.
10. Dushkin R. V. Funktsional'noye programmirovaniye na yazyke Haskell / Gl. red. D. A. Movchan;. — M.: DMK Press,, 2008. — 544 p — ISBN 5-94074-335-8.
11. Ofitsial'nyy sayt yazyka Haskell — "O yazyke" http://haskell.org/aboutHaskell.html
12. 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/
13. Mitchell, R.W., "LISP 2 Specifications Proposal", Stanford Artificial Intelligence Laboratory Memo No. 21, Stanford, Calif., 1964.
14. Landin, P. J. (January 1964). "The Mechanical Evaluation of Expressions". Comput. J. **6** (4): 308—320. doi:10.1093/comjnl/6.4.308
15. Khenderson P. Funktsional'noye programmirovaniye. Primeneniye i realizatsiya = Functional Programming. — M.: Mir, 1983. — 349 p.
16. 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
17. Gorodnyaya L.V. Lisp i yego dialekty // Novosibirsk, preprint, 2025 https://www.iis.nsk.su/repository/gorod.14408.
18. Billig V.A. Khronika yazykov programmirovaniya. Proshloye, nastoyashcheye, budushcheye : uchebnoye posobiye / V. A. Billig ; INTUIT natsional'nyy otkrytyy universitet. — Moskva : INTUIT, 2024. — 286 p. — ISBN 978-5-9556-0204-2 https://intuit.ru/goods_store/books/10019?ysclid=mdqvqlwbyg377265958