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

Л.В. Городняя

ОПЫТ ПАРАДИГМАЛЬНОЙ ДЕКОМПОЗИЦИИ ОПРЕДЕЛЕНИЯ ЯЗЫКА ПРОГРАММИРОВАНИЯ НА ПРИМЕРЕ ЯЗЫКА CLISP

Препринт

Новосибирск 2025

Рецензент – к. физ.-мат. наук, Д. С. Мигинский

Препринт посвящен экспериментам по декомпозиции описаний языков программирования на основе парадигмального анализа их эксплуатационной и реализационной прагматики. Показан ряд решений для создания методики сравнения парадигм и языков программирования, отражающей выразительную силу языков, продуктивность реализации систем программирования и приспособленность лаконичных описаний к обоснованию практичных, объективных критериев декомпозиции программ. В качестве основного подхода выбрана семантическая декомпозиция определений языков с выделением семантических систем при анализе поддержанных языком парадигм программирования. Такой выбор позволяет выделять автономно развиваемые типовые компоненты программ и диалекты языков программирования, которые могут быть приспособлены к конструированию различных информационных систем. Многие работы по методам разработки программных систем зависят от практичности подходов к декомпозиции программ и программных систем, отлаживаемых с помощью систем программирования. Методы проиллюстрированы на материале мультипарадигмального языка Clisp.

Решение этой проблемы полезно при изучении методов программирования и исследовании истории языков программирования для сравнения парадигм программирования, потенциала используемых схем и моделей, оценки уровня новизны создаваемых языков программирования, а также при выборе критериев декомпозиции и улучшения программ в условиях скрытого размывания, оттеснения или эрозии знаний в форме обновления интерфейсов (feature creep, update fatigue, knowledge dilution). Результаты парадигмальной и семантической декомпозиции могут быть основой при подготовке лаконичных учебных материалов по новым языкам программирования, описания которых слишком объёмны.

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

L.V. Gorodnyaya

A CASE STUDY OF PARADIGMATIC DECOMPOSITION OF A PROGRAMMING LANGUAGE DEFINITION: THE EXAMPLE OF CLISP

Preprint

Novosibirsk 2025

Reviewer Dr. D S.Miginsky

The preprint is devoted to the choice of means of using the methods of syntactically oriented design and functional programming in solving the problems of text processing of definitions of programming languages. The problem of improving the developed programming systems and creating new programming languages aimed at the effective solution of modern problems of developing reliable and convenient information systems, including the organization of parallel computing, is considered.

The research was supported by the Russian Foundation for Basic Research (RFBR) of the Russian Academy of Sciences № 18-07-01048-a.

Введение

Парадигмальная декомпозиция на основе семантического анализа прагматики сложных описаний языков программирования (ЯП) рассматривается как путь преодоления их образовательной сложности. Основные соображения представлены в статьях [3-11]. В первом разделе препринта приведены предварительные соображения о термине «парадигма программирования». Далее во втором разделе описаны мотивы выбора парадигм программирования в качестве ведущего критерия при декомпозиции программ и обоснован выбор мультипарадигмльного языка Clisp для первых экспериментов. В третьем — приведены основные термины, используемые при описании семантики языка Clisp и представлении результатов его парадигмальной декомпозиции. В четвёртом приведены монопарадигмальные понятийные матрицы, полученные как результат декомпозиции комплекта функций Clisp по отдельным парадигмам, приведены результаты сравнения выделенных элементов по слоям реализационной прагматики и по категориям семантических систем, показана визуальная карта языка Clisp. В заключении дан прогноз возможности применения предложенных средств для представления результатов анализа и сравнения языков программирования..

1. Предварительные положения

Во многих источниках, посвящённых пониманию парадигм программирования (ПП), ставят остающиеся без ответа вопросы:

Почему возникло достаточно много парадигм программирования?

Почему одна парадигма и один язык программирования, при всей их полноте, не решают все проблемы?

Почему не каждая парадигма и не каждый язык программирования получают признание программистского сообщества?

Возможные ответы:

1) Новые проблемы по трудоёмкости решения в разы превышают трудоёмкость решения знакомых проблем [9]. Не исключено, что именно эта причина определяет принципиальную разницу в опережающем прогрессе создания новой элементной базы в сравнении с нарастающим отставанием развития программного

обеспечения. Разработчики элементной базы работают в стабильном, более полуста лет знакомом, классе задач повышения эффективности аппаратуры. Развитие программного обеспечения происходит в непрерывно расширяющимся пространством классов задач, связанных совершенствованием элементной базы, с освоением новых сфер приложения ИТ и обучением новых категорий пользователей.

2) Формально, с математической точки зрения, любые ПП и ЯП работают на одном и том же пространстве рекурсивно-вычислимых функций и в этом смысле теоретически позволяют решать все разрешимые проблемы автоматизации процессов[1]. Реальным препятствием является трудоёмкость решения новых проблем людьми. Люди редко решают ранее не известные им актуальные проблемы, требующие способности изобретать или создавать новые подходы и средства, в приемлимое для «структуры момента» время, чаще найденные решения новых проблем несколько запаздывают для получения выгоды.

Популярное определение термина «парадигма программирования» рассматривает его как фундаментальный стиль подготовки компьютерных программ, похожий на чертеж, предоставляющий базовые концепции, примеры, образцы и шаблоны, которые принимают и реализуют разработчики языков программирования, чтобы их применяли программисты. Точнее, это способ мышления, порождающий расширяющийся ряд конкретных парадигм, возникающих как реакция программирования на проявление новых трудно решаемых проблем.

3) Реально понятие «чертёж» живёт в профессиональном сообществе, обладающем знаниями и умениями понимать и представлять общеизвестную символику чертежа, по которой грамотный специалист может произвести нужное «изделие» – товар, применяемый в дальнейшем без особых изменений. Подобная символика представления ПП ещё не сформирована [6]. Кроме того, является ли язык программирования изделием?

Распространено мнение, что программисты обычно

1 Была, но стала недоступной ссылка https://www.janeve.me/software-programming/understanding-programming-paradigms

«обладают» знаниями, инструментами и методами реализации решений с использованием исключительно заранее известных, привычных им парадигм программирования, причём обычно всего одной, в плену которой он «может думать», что заметно противоречит понятию «чертёж», допускающему изучение и определение по мере необходимости.

4) Существуют мультипарадигмальные ЯП и комбинированные, производные, вспомогательный, дополнительные и другие категории ПП. Верно лишь то, что в каждый отдельный момент удобно учитывать одну ПП, что делает используемую парадигму подобной планарным проекциям в инженерии. Языки подвержены развитию по мере роста их популярности, что затрудняет их трактовку как изделий, в отличие от программных модулей.

Методисты обычно рекомендуют сосредоточить больше внимания на ***концепциях*** *программирования и различных парадигмах программирования, что помогает понимать несколько языков программирования и то,* ***чем они отличаются*** *друг от друга.*

5) Конечно, важно выбрать ключевые концепции, по которым реально можно систематизировать ПП и ЯП. Обычно такое внимание сосредоточено на противопоставлении отдельных особенностей изобразительных средств программирования, разнообразие которых слегка превосходит число ПП и на порядки меньше мощности множества ЯП. Это не позволяет их рассматривать в качестве ключевых потому, что любое средство может поддерживать разные ПП, а ЯП можно оснастить библиотеками, поддерживающими любые ПП на уровне СП.

Дидактически удобно логику порождения парадигм программирования рассматривать как череду этапов перехода от концепций ПП к конкретным ЯП, поддерживающим ПП. Можно утверждать, что ЯП развивается через серию таких этапов (обзор концепций; выбор целевого подмножества, спецификация модели программ, язык программирования). Появились прецеденты определения ЯП как комплекта диалектов.

6) Логика восприятия может опираться на не достаточно

достоверные сведения [12]. Реально в практике программирования и применения ИТ осознаются новые трудно решаемые проблемы, продумывание которых в рамках привычных ПП слишком неудобно, а реализация их решений слишком трудоёмка. Вдруг кому-то удаётся изобрести подходящие шаблоны менее трудоёмких конструкций, среди единомышленников формируется речевая практика, приводящая к пониманию моделей программирования новых типовых решений [1, 2, 7, 13, 16, 17, 19]. В результате повышается степень изученности проблем и по этой причине снижается в разы трудоёмкость их решения, повышается продуктивность программирования [8. 14, 15]. Это позволяет получить признание, кристаллизовать профессиональную терминологию и лишь на фоне этого формализуются концепции, позднее получающие «звание» ПП.

Принято говорить, что истоком концепции программирования может быть блестящая ***идея, которую кто-то придумал,*** *редко признают, что всплыла хорошо спланированная и реализованная функция на ЯП, который* ***уже был давно доступен****, например: объекты, наследование, аспекты, функции, ленивое вычисление, области видимости переменных, много-поточность и параллелизм, динамическое программирование и т. д. – всё это было в некоторых языках и технологиях программирования, но не обладало статусом парадигмы.*

7) Именно это и имело место при появлении ООП в качестве практичной ПП [18]. «Придумывание» чаще всего сводится к комбинации ранее известных идей. Новые идеи сами приходят лишь в редкие головы [2, 7, 10, 11, 13, 16, 17].Тем не менее, иногда удаётся неожиданные идеи моделировать и показывать на универсальных языках функционального или логического программирования, сама универсальность которых обычно подвергается критике за «низкую» эффективность экспериментальных, демонстрационных программ. Роль такой работы в общем процессе снижения трудоёмкости и повышения продуктивности программирования обычно не заметна.

Комплект концепций ***выбирается разработчиками*** *ЯП, которые определяют* ***модель программирования****. Затем они приступают к реализации этих функций, в результате чего получается новый язык программирования, являющийся расширенным*

подмножеством исходного языка, обогащённым конкретными реализационными решениями и особенностями используемых архитектур. Когда конкретный способ мышления или модель программирования созревает и все больше и больше разработчиков ЯП начинают ее применять, а программисты на её основе создают работоспособные программы, она становится ***парадигмой программирования****.*

8) Кроме «разработчиков языка» существуют программисты, дающие оценку практичности созданного языка в виде отлаженных программ, только после такой формы признания можно утверждать, что появилась ПП, обычно на это уходит 5-15 лет [4, 10, 20]. Большинство ЯП мультипарадигмальны. Монопарадигмальные языки встречаются редко, их приходится искать или создавать специально для исследовательских целей. Clisp – пример мультипарадигмального ЯП, поддерживающего 6 наиболее популярных, фундаментальных ПП (функциональная, декларативная, рефлексивная, императивная, объектно-ориентированная и мета-программирование) [22].

В числе популярных и ключевых концепций, активно используемых во время программирования, обычно противопоставляют активные и ленивые вычисления, процедуры и функции, внешние и внутренние состояния памяти, статический и динамический анализ программ, последовательные и параллельные процессы их выполнения. Фундаментальные ПП имеют производственную и стратегическую направленность, связанную с исследовательскими и образовательными проблемами.

9) Исследование истории вычислительной техники и программирования показывает, что элементная база, языки, системы и парадигмы программирования – всё это механизмы реализации нужных автоматизируемых средств, они могут поддерживать любую ПП, хотя существуют некие традиции, обычно на практике преодолеваемые техникой библиотечных модулей, поддерживающих любой нужный механизм для любого ЯП [20, 21].

Первые эксперименты по парадигмально семантическому анализу выполнены на материале языков PureLisp и Pascal, результаты представлены в форме понятийных матриц и визуальных диаграмм [3].

Студенты ФИТ и ММФ НГУ выполнили такую работу примерно для 30 языков программирования, преимущественно новых, появившихся в XXI-ом веке (Assembler, C, C++, C#, Clojure, Elixir, Erlang, F#, FORTH, Haskell, Java, JavaScript, Kotlin, LISP, Mojo, Python, R, Rust, Scala и другие). Магистранты С. Е. Демидов и М. Д. Кириченко разработали ряд визуальных и информационных форматов для выполнения экспертами работ по сравнению языков программирования и предложили подходы для автоматизации таких работ.

Выполнена парадигмальная декомпозиция описания языка C, поддерживающего 4 парадигмы (императивная, структурная, скалярная и процедурная), в расчёте на выполнение аналогичной работы для других языков этого семейства [23].

В данном препринте описана методика парадигмальной декомпозиции на примере мультипарадигмаьного языка Clisp, поддерживающего 6 парадигм (функциональная, декларативная, рефлексивная, императивно-процедурная, мета-программирование и ООП) [22], половина из них предназначены для восходящей методики программирования (функциональная, мета-программирование, декларативная), другие — для нисходящей (рефлексивная, императивно-процедурная, ООП), что позволяет процесс программирования на таком языке выполнять как чередование шагов, использующих разные ПП и методики в рамках общей обстановки полного жизненного цикла программ.

Долгоживущие программы, такие как системы программирования (СП), требуют объективных критериев декомпозиции программ, отражающих возможность автономного развития выделенных компонентов СП с целью профилактики повторного программирования и накопления опыта реализации диалектов при развитии реализуемого ЯП.

2. Обоснование выбора парадигм программирования

Существует принципиальная разница в механизмах формирования понятий, фразеологии и речи в математике и программировании, связанная с особенностями признания результатов [4]. Если результат в математике должен получить признание у других математиков, то это даёт основания для формирования чёткой терминологии, общепринятого лексикона и культуры математической речи. Изложение математического результата традиционно начинается со строгого введения терминов, причём, термином может быть объявлено любое слово, независимо от его смысла в естественном

языке: группа, кольцо, поле и т. д.

Результат в программировании не требует такой строгости в лексиконе, культура программистского удостоверения результатов апеллирует не к другим программистам, а к компьютерному эксперименту. Достаточно показать, что такой эксперимент можно выполнить, причём его результаты могут быть понятны и не специалистам. Роль лексикона, определений, теорем и доказательств и признания их достоинств коллегами выполняет демонстрация результатов компьютерного эксперимента, Восприятие результатов мало зависит от речевой практики. Оценка их значимости требует сопоставимого опыта выполнения сравнимых по сложности работ.

Кроме того, математические термины — это мир идей, программистские — это мир данных, имеющий смысл как компьютерные реалии — коды, сигналы, адреса и т.п.. Разница между математическими идеями и компьютерными данными напоминает разницу между нотной записью и звуками музыки. Данные бывают целостными, неделимыми, атомарными или составными, содержащими элементы и компоненты, допускающие их выделение и обработку по частям.

Понятие «парадигма программирования» (ПП) не имеет строгого определения, поэтому возникает вопрос о принадлежности новых подходов в программировании и ИТ к множеству ПП и об упорядочении такого множества. **Парадигма программирования** представляет собой расширяющийся ряд отдельных парадигм, поддерживающих продуктивность программирования решений конкретных классов задач. Отдельные ПП проявляются как образ мышления, связанный с компромиссом между особенностями решаемых задач конкретного класса, методами их решения в форме программ, принятыми в ПП критериями качества программ и приоритетами принятия решений в процессе программирования. Такая особенность ПП позволяет понимать выбор парадигмы как процесс принятия, представления и отладки решений при постановке разных задач, поэтому естественно систематизацию ПП выполнить по сопоставлению с приоритетами и варьированием схем постановки задач и методов их решения. В этом плане **функциональное программирование** (ФП) нацелено на расширение границ вычислимости методом символьных вычислений. Типовая задача этого класса — исследование предметной области с недостаточной степенью изученности для производственного программирования и доведение её до алгоритма или прототипа решения.

Наиболее ясная систематизация ПП в настоящее время позволяет выделять базовые, фундаментальные и основные ПП, дополненные комбинированными, производными, вспомогательными, и системообразующими или перспективно-стратегическими. Следует отметить, что академик Андрей Петрович Ершов основное внимание уделял именно стратегическим ПП, включающим фундаментальные, образовательные и технологичные [24]. Множество основных ПП можно разделить на базовые, инструментально расширяющие и неограниченные в зависимости от наполнения семантических систем организации вычислений, работы с памятью, управления вычислениями и конструирования сложных данных.

Описания современных языков программирования (ЯП) обычно содержат список из 5-10 языков-предшественников и ряд ПП, поддержанных языком [20, 21]. Такая характеристика, как правило, не показывает, какие свойства ПП и черты предшественника фактически унаследованы определяемым ЯП и какие особенности являются действительно новыми. Для реальной оценки уровня новизны и практичности ЯП может быть полезной коллекция постановок задач с примерами их решения на разных языках в рамках различных парадигм. Определённое количество примеров обычно присутствует в описаниях ЯП и учебных пособиях. Чаще всего это программа печати приветствия, что позволяет быстро начать эксперименты с системой программирования (СП), но, учитывая, что большинство современных ЯП поддерживают механизмы ввода-вывода на уровне библиотечных вызовов, такие примеры слабо характеризуют язык.

Джон МакКарти провозгласил, что любой информации для обработки на компьютере можно дать **символьное представление**, которое может быть как атомарным, так и составным [17]. Святослав Сергеевич Лавров в начале 1970-ых годов в ответ на предложение Андрея Петрович Ершова термин «Информатика» понимать как перевод «Computer Science» написал шуточную страничку, утверждая, что содержательно более точным был бы термин «**Датаматика**» [11]. Похожие соображения выражали и другие учёные[2]. Действительно, при переносе терминов из математики в программирование ради простоты говорения и удобочитаемости происходит нечто вроде стенографического упрощения, в котором теряется связь понятий с данными. Особенно явно такая потеря происходит на понятиях «значение» и «функция». Если в математике значением может быть

2 Эту идею ещё высказывали Г. С. Цейтин и Эдсгер Дейкстра (*Edsger Wybe Dijkstra)*

элемент любого, возможно бесконечного множества произвольной природы, функция — это линия в пространстве над таким множеством, а точки функции могут быть заданы таблицей или любым умозрительно понятным механизмом или алгоритмом, то программирование имеет дело лишь с данными, **представляющими** часть такого множества, а роль функций выполняют данные, **представляющие** технику доступа к данным, **представляющим** результат функции. Естественно, при быстрой речи по умолчанию термин «представление» оптимизируется, а заодно и исчезает из понимания.

В данном препринте рассматривается эксперимент по парадигмальной декомпозиции комплекта функций мультипарадигмального языка Clisp [22]. Такой комплект можно рассматривать как определение реализации языка, удобное для анализа и декомпозиции. Комплект систематизирован по типам данных, каждый из которых используется почти во всех шести парадигмах. Это означает, что при изучении каждого типа данных следует иметь в виду особенности всех шести парадигм, а большинство людей предпочитает в каждый момент работать в рамках одной парадигмы.

Общая харатеристика ряда парадигм приведена в таблицах 7-13. Результат методики декомпозиции мультипарадигмального языка Clisp показан в форме шести монопарадигмальных понятийных матриц (см. таблицы 14-19). При выполнении декомпозиции возникают проблемы с необходимостью добиваться их ортогональности, что иногда требует волевых решений, которые могут быть уточнены. Ещё полезна дополнительная работа по выравниванию наполнения строк и столбцов матрицы для удобства сравнительного анализа ЯП (таблицы 21-29).

Чисто в учебных целях можно один раз дать расшифровку стенограммо-подобной программистской речи, но реальная практика живого языка такого не допускает.

3. Основные термины

Следует отметить, что жаргон современного практического программирования использует понятие «язык программирования» как «входной язык СП – расширенное подмножество ЯП типовой СП, функционирующей на базе определённой конфигурации оборудования» (ЯиСП). Разница заключается в том, что СП обычно сопровождает реализацию ЯП расширяемым комплектом библиотечных модулей. В результате происходит сглаживание

видимых на практике различий между языками и системами программирования. Кроме того, прямые измерения продуктивности программирования и производительности программ почти не отражают зависимости результата от принятых программистом решений и выбора конструкций ЯП. Хотя программируемые решения представляются в терминах ЯП, его влияние растворяется в весьма сложном комплексе, наследующем производительность СП и оборудования. Таким образом, существует проблема создания методики, позволяющей выявлять такие зависимости совмещением прямых измерений с результатами экспертных оценок особенностей ЯиСП, возможно отличающихся от оценок ЯП [3, 9, 11].

Есть основания при прогнозировании продуктивности программирования учитывать **степень изученности** решаемых задач, **уровень квалификации** и способностей разработчиков программы решения задачи и **понятийную сложность** реализуемых и используемых программируемых и программных средств. Вопрос оценки сложности систем программирования был подробно исследован Т. С. Васючковой, убедительно показавшей, что сравнение СП требует более чем 200 параметров. Представление результатов оценки понятийной сложности, позволяющее структурировать пространство таких параметров, рассмотрено в статье [3].

По **степени изученности** существенно различаются следующие категории постановок задач, влияющие на выбор методов решения задач и продуктивность их программирования [6]:

- новые;
- исследовательские;
- практичные;
- точные.

Для **новых** постановок задач характерно отсутствие доступного прецедента решения задачи, новизна используемых средств или недостаток опыта исполнителей. **Исследовательск**ие постановки задач обычно усложнены требованиями уникальности и универсальности. **Практичные** постановки задач нацелены на актуальность и удобство применения. **Точные** постановки задач включают в себя испытание предельных возможностей используемых средств, связанных с мерой организованности созданной программы и рангом работоспособности реализованных решений. Разброс трудоёмкости в зависимости от степени новизны постановки задачи обычно составляет примерно 1 к

8. Недоучёт такого разброса обычно приводит к систематическим ошибкам прогноза предстоящей продуктивности программирования.

Макетный образец решения новой задачи работоспособен при предъявлении автором небольшого набора подходящих данных. Для **экспериментального полигона** отладки решений исследовательских задач требуется приспособленность к обработке почти любых данных, которые могут быть заданы в качестве входных. **Практичная версия** может быть ограничена обработкой данных, реально встречающихся в сфере её приложения и отлаженными сценариями применения, задаваемыми как шаблоны проектирования. Для **точной реализации** нужны специально подобранные данные, показывающие её превосходство над менее точно и эффективно выполненными решениями задачи, зато она обычно может использовать готовые алгоритмы и прототипы.

3.1. Семантические системы

В качестве отправной точки для представления решений по декомпозиции программ можно принять классическое понятие «алгебраическая система» [12].

Алгебраическая система — это <A, F>, где:

A — основное множество значений,

F — конечный набор операций над множеством A.

Рядом с таким понятием существует неявное, всем известное, правило применения операций к значениям, согласно которому сначала вычисляются значения, потом к ним применяются операции. Примерно так можно формализовать библиотечные модули в СП и классы объектов в ООП до тех пор, пока при реализации ЯП не возникает необходимость использовать разные модели вычислений по одним и тем же формулам. С. С. Лавров предложил понятие «семантическая система», расширяющее понятие «алгебраическая система» заданием явного правила применения функций к значениям [11].

Семантическая система — это <**V, F, R**>, где:

V — основное множество значений,

F — конечный набор функций, возможно принадлежащих основному множеству,

R — правило применения функций к значениям, возможно входящее в набор функций.

Близкое понятие под названием «монады» введено в язык Haskell. Большинство ЯП поддерживает разные варианты утончения понятия «семантическая система», например, где:

F — конечный набор функций, **не принадлежащих** основному множеству,

R — правило применения функций к значениям, **не входящее** в набор функций.

Соответствующее обозначение такого вида семантических систем можно представить как <**V; F; R**>. Вид <V; F; R> характерен для ЯП, обладающих чётким разделением данных и программ, а также для языков управления базами данных. Разницу в строгости границ между составляющими семантической системы можно выражать таким образом:

< V; F; R > – строгое разделение значений и функций (Pascal);
< V, F; R > – разделены функции и правила (C++);
< V; F, R > – разделены значения и функции (Forth);
< V, F, R > – функции, значения и правила не разделены (Lisp).

Форматы < V; F; R > и < V, F; R >, характерные для парадигм императивного программирования (ИП) и ООП, приспособлены к передаче профессионального опыта в форме библиотек процедур, встраиваемых в стандартный каркас применения операций к данным. Форматы < V; F, R > и < V, F, R >, присущие функциональному программированию (ФП), поддерживают передачу опыта в форме диалектов и пакетов, являющихся областями видимости символов со своими правилами их интерпретации. Можно сказать, что эти форматы семантических систем обладают разными пространствами свободы программирования.

3.2. Архитектурные модели

Наиболее известные компьютерные архитектуры обычно представляют как конструкцию из вычислительного устройства (E), памяти (M), устройства управления процессами (C) и средств коммуникации между устройствами и их элементами (S). Такая конструкция допускает детализациею на уровне более тонких аппаратных решений и расширение на уровне периферийных устройств, что даёт первое приближение для выбора основных видов семантических систем в языках программирования [3, 6]. Рассматривая

любые семантические системы, важно отметить разницу в характере выполнения функций таких систем. Так для любого множества значений V реализационно различимы виды функций для методов вычислений FE: (V* → V+), средств доступа к памяти FM: (T : N → V), особенностей управления вычислениями FC: (F → {0, 1})* и обратимой комплексации и структурирования данных FS: (A → K) U (A ← K) = (A ↔ K).

Следует обратить внимание, что такие виды семантических систем обладают кумулятивным эффектом в порядке «VEMCS». Если V – произвольное множество значений, то FE: (V* → V+) — обычные вычисления, заданные формулами над значениями из этого множества, а формула может представлять само-определимое константное значение из V. Для реализации средств доступа к памяти FM: (T : N → V) — характерно выделение понятия "адрес" (N) и подразумевается существование специальной таблицы T, по которой определено соответствие адресов заданным значениям, при задании которых могут использоваться формулы вычислений. Особенности управления вычислениями FC: (F → {0,1})* используют разметку запрограммированных действий для выделения выполняемых, обычно используя понятие адресуемой памяти. Обратимая комплексация или структурирование данных в современных архитектурах FS: (A ↔ K) требует определения границы между атомарными (A) и сложными, конструируемыми (K) объектами с возможностью как наращивания сложности, так и упрощения любых построений с учётом разных условий.

Это приводит к представлению о кумулятивной шкале семантических систем на основе классификации видов функций.

Таблица 1

Классы семантических систем

№ столбца	**V**	**E**	**M**	**C**	**S**
Категории функций (F V)	**V**	**FE: V* → V+**	**FM: (T : N → V)** V = N U V	**FC: (F → {0, 1})*** [3] V = {0,1} U V	**FS: (A → K) U (A ← K)** V = A U K

3.3. Понятийные матрицы

Понятийная сложность средств программирования часто замаскирована удобными интерфейсами. Представление результатов анализа ЯП можно рассматривать как основу для измерения

3 Обозначение уточнено А.В.Климовым

понятийной сложности [9]. Для этого достаточно изучение описаний языков программирования (ЯП) сопровождать парадигмально-семантической декомпозицией, результаты которой представляются понятийными матрицами. Из таких матриц можно вывести матрицу численных характеристик понятийной сложности ЯП. Позициям такой матрицы можно сопоставить элементы обучающего ряда примеров, иллюстрирующих смысл понятий ЯП. Для мультипарадигмальных ЯП можно представить несколько понятийных матриц по числу поддержанных парадигм. Наполнение понятийных матриц разными экспертами могут отличаться. Ряд примеров может отражать методику обучения. Такие понятийные матрицы достаточны для представления результатов анализа ЯП и грубой оценки внешней сложности их применения при прикладном программировании, например, по числу различимых конструкций в каждой клетке понятийной матрицы. Более тонкая оценка внутренней сложности реализации ЯиСП потребует комплекса таких матриц, показывающих сложность внутренних построений, таких как тексты, внутренние форматы, структуры, коды.

Для более точной оценки ЯиСП и анализа измеримых характеристик можно использовать комплекты сопоставимых фрагментов программ на оцениваемых ЯП в форме, приспособленной для прямого эксперимента с СП, поддерживающих изучаемые ЯП. Примеры накапливаются как упорядоченный по сложности их усвоения обучающий ряд. Каждый пример иллюстрирует особенности использования отдельного понятия. Для иллюстрации некоторых понятий может требоваться несколько примеров. Последовательность элементов ряда соответствует порядку изучения понятий, подкреплённому компьютерной практикой. Каждый элемент содержит представление результата прогона фрагмента на СП и пояснение схемы применения таких фрагментов в подходящих задачах. Число примеров, необходимых для понимания практики применения понятия в программах, можно рассматривать как оценку сложности изучения понятия. В зависимости от целей обучения над обучающим рядом можно строить маршруты или учебные траектории, отражающие особенности решаемых задач или сферы применения их решений.

Использование шкалы сопоставимых постановок задач и методов их решения, включая прагматику СП, даёт критерии для разработки подходов к оптимизации программ и систем программирования, уточнения методов измерения трудоёмкости программирования и производительности программ. Для каждого ЯП определены списки поддержанных в нём парадигм, предшественников,

сфера влияния. Такие характеристики можно выводить из списков общих семантических систем, оценивая существенную разницу в традиционной прагматике функционально эквивалентных систем.

3.4. Диалекты или подъязыки

Одним и тем же наборам функций могут соответствовать разные правила **R**, определяющие методы вычислений, влияющие на результативность выполнения программ.

B=RE — обычные арифметические вычисления, отображающие произвольный ряд **значений** аргументов в не менее чем один результат,

X=RM — символьные вычисления, подставляющие **представления** аргументов без их предварительного вычисления,

D=RC — частичные или смешанные вычисления, лавирующие между вычислением и подстановкой в зависимости от разных условий,

P=RS — обобщённые и параллельные вычисления, оперирующие организацией процессов как множеством потоков над комплексами из разных устройств.

Представление таких различий можно выразить дополнив горизонтальную шкалу видов функций вертикальной кумулятивной шкалой моделей вычислений или методов применения функций к значениям (R), что будет выглядеть как матрица семантических систем — понятийная матрица. Различия классов семантических систем на уровне правил применения функций к значениям в ЯВУ выражены таблицами 2-5. Первая строка этой матрицы представляет уровень базовой семантики ЯП, существенные различия видов функций семантических систем которого можно представить Таблицей 2. Вторая строка Таблицы 3 представляет уровень макрорасширений, что вместе с первой строкой позволяет характеризовать средства языков низкого уровня.

Таблица 2

Семантическая декомпозиция минимального ядра ЯП

№ строки	*№ столбца*	***V***	**E**	**M**	**C**	**S**
	Виды функций	*V*	*FE: V* → V+*	*FM: (T : A → V)*	*FC: {F → {0,1}}*	*FS: A ↔ K*
B	**RE: Ядро**	Значение	Операции	Память	Управление	Вектор

Ядро — семантический базис. Полное определение ЯП можно получать как консервативное расширение ядра. Обычно ядро

приспособлено и к неконсервативному расширению пополнением набора библиотечных функций, реализуемых на уровне аппаратуры или операционной системы. Это позволяет в реализации СП для любого ЯП использовать разные парадигмы программирования, необходимые для поддержки полного жизненного цикла программ, чтобы достигать результата независимо от исходных возможностей ЯП.

Значение — минимальное представление объектов из области приложения языка, обычно это само-определимые константы.

Операции — минимальный комплект функций для обработки значений.

Память — введение адресов для лаконичного и уникального представления значений (указатели, идентификаторы, переменные, метки).

Управление — разметка выполнимости композиции элементов программы из (операций, функций, действий и т.п.) специально выбранными значениями, например, {0|1} или {True | False} или {Nil | T}.

Вектор — обратимое конструирование одноуровневых комплектов, рассматриваемых как целостность, из которых можно восстанавливать исходные элементы. На уровне ядра достаточно одной структуры — вектора, списка, очереди или т. п.

Таблица 3

Семантическая декомпозиция макрорасширения ядра ЯП

№ строки	*№ столбца*	***V***	**E**	**M**	**C**	**S**
	Категории функций	*V*	*FE: V* → V+*	*FM: (T : A → V)*	*FC: {F → {0,1}}*	*FS: A ↔ K*
B	RE: Ядро	Значение	Операции	Память	Управление	Вектор
X	**RM: Макро**	Данное	Функции	Задание	Блоки	Стек

Макро — пополнение ядра средствами обработки представлений, используемых с целью укрупнения любых конструкций, что позволяет выполнять консервативное расширение ЯП. Кроме того, оно способствует лаконизму текстов программ. Макротехника позволяет наследовать отлаженость фрагментов программ. Бывает важным исключать дубли частей текста, кода и структур данных. Простейший механизм макрогенерации обычно присутствует в СП как препроцессор. Также бывает устроена техника кодогенерации и обработки шаблонов при компиляции программ.

Реализация укрупнений может быть функционально эквивалентна вызову подпрограмм. Взаимозаменяемость макроподстановки и вызова подпрограмм нередко используется при оптимизации программ.

Данное — хранимое значение или выражение, допускающее уникальность экземпляра, доступного многократно по адресу, возможно, на внешнем устройстве.

Функции — укрупнение операций с возможной параметризацией операндов. Реализационная прагматика может отличаться техникой передачи параметров через стек или специальное поле аргументов или неявно. Последнее позволяет и работу с памятью формально рассматривать как функцию с неявным аргументом, выполняющим роль функции T, задающей соответствия адресов и значений.

Задание — хранимое именованное выражение с возможностью многократного выполнения.

Блоки — хранимое выражение или код программы, представляющий составные действия, ветвления, циклы, вызовы функций, обычно с локализацией переменных, приводящей к понятию «иерархия».

Стек — схема организации данных, с определённой дисциплиной доступа для поддержки иерархии, возможно с защитой независимых блоков.

Повышение уровня ЯП обеспечивается не только особым вниманием к средствам укрупнения данных на базе понятия «иерархия» и оперирование блоками программы. Дальнейшее наращивание объёмов разрабатываемых программ отчасти достигается автоматизацией диагностики и контроля некоторых условий корректности применения операций и функций к их операндам в определённых границах. Становятся важными понятия «предикат» и «тип переменных», удобно проверяемые при компиляции, что представлено третьей строкой Таблицы 4. По мере расширения границ программа может стать достаточно универсальной, способной к разумному поведению на любых входных данных, что выражено в Таблице 5, в которой добавлена слева колонка с обозначениями строк полученной понятийной матрицы.

Таблица 4

Семантическая декомпозиция диагностического дополнения ядра ЯП

№ строки	№ столбца	V	E	M	C	S
	Категории функций	V	FE: V* → V+	FM: (T : A → V)	FC: {F → {0,1}}	FS: A ↔ K
B	RE: Ядро	Значение	Операции	Память	Управление	Вектор
X	RM: Макро	Данное	Функции	Задание	Блоки	Стек
D	**RC: Границы**	Исключения	Предикаты	Типы данных	Логика	Варианты

Границы — методы проверки вычислимости функций и выполнимости заданий. Цель представления границ и диагностики неожиданностей — повышение продуктивности отладки программ упрощением поиска ошибок. При отсутствии ошибок проверка воспринимается как накладные расходы. Встречаются механизмы прерываний и установки ловушек на непредусмотренные ситуации и программирования обработки исключений с возможностью продолжения вычислений.

Исключения — выбор специальных значений для разметки неожиданных ситуаций. В некоторых ЯП вводят значения типа “Error”. При переходе к СП происходит добавление текстовых шаблонов для формирования диагностических сообщений об исключительных ситуациях.

Предикаты — специальные функции, позволяющие определять типы значений или сравнивать значения независимо от расположения данных в памяти. Роль предиката может выполнять любая функция при подходящих договорённостях и схеме реагирования на её результаты.

Типы переменных — связывание типа значения с переменной, хранящей нетипизированный код значения в памяти.

Логика — проверка соответствия типа данных или значений операциям обработки значений или доступа к памяти, возможно с учётом условий вычислимости.

Варианты — схема организации данных без определённой дисциплины доступа для организации перебора равноправных элементов или блоков. Полезно при отладке программ как механизм удостоверения принципиальной выполнимости вычислений при частичной постановке задачи.

Таблица 5

Семантическая декомпозиция практичного обобщения ядра ЯП

№ строк	№ столбцов	V	E	M	C	S
	Категории функций	V	FE: V* → V+	FM: (T : A → V)	FC: {F → {0,1}}	FS: A ↔ K
B	RE: Ядро	Значение	Операции	Память	Управление	Вектор

X	RM: Макро	Данное	Функции	Задание	Блоки	Стек
D	RC: Границы	Исключе-ния	Предикаты	Типы данных	Логика	Варианты
P	**RS: Среда**	Неопреде-лённость	Мульти-операции	Внешний мир	Отображения	Ввод-вывод

Среда — дополнительные средства обеспечения отладки и применения программ, поддерживающие возможность разумного продолжения вычислений при любых исходных данных и аварийных ситуациях во внешнем мире.

Неопределённость — вводятся специальные дополнительные значения и ловушки (_|_, Error, Future). Такое расширение множества значений позволяет учитывать в текстах программ некоторые отдельные особенности процесса разработки, отладки и схемы жизненного цикла программ.

Мультиоперации — допускается произвольное число операндов операций, аргументов и результатов функций. Возможно просачивание определений на однородные структуры данных, позволяющее определения над элементарными данными автоматически распространять на более сложные данные.

Внешний мир — механизм неявного расширения области действия операций и функций на периферийные устройства, рассматриваемые как обобщение памяти.

Отображение — возможность регулярного применения функции к серии данных благодаря использованию представлений функций или указателей в качестве аргументов функций более высокого порядка.

Ввод-вывод — средства приёма данных с внешних устройств и размещения данных на внешних носителях данных, включая средства доступа к устройствам с уровня программы. Стандартно имеется в виду приём данных с клавиатуры и изображение данных на экране. Обычно подразумевается аксиоматика, требующая совместимости форматов ввода-вывода: для всякого вводимого данного существует эквивалентное ему выводимое данное и обратно — если данное может быть выведено, то его можно ввести без потерь.

Таким образом, разным видам функций относительно схемы применения функций к аргументам на уровне ЯП при реализации СП соответствуют определённые позиции специальной понятийной матрицы, полученной как двумерная кумулятивная шкала. Кумулятивный эффект по второму измерению получается совмещением результатов ячейки [B,S] с [X,V], [X,S] с [D,V], [D,S] с [P,V]. Более подробные пояснения даны в таблице 6.

Таблица 6

Кумулятивные эффекты в понятийной матрице при переходе на очередную позицию (наследование средств предшествующих позиций).

№ позиции	*Пояснение*
BV 	Само-определяемые скаляры, их смысл не требует интерпретации, текстовое представление понятно без комментариев.
BE 	Операции над скалярами, они должны вырабатывать скаляры. При выходе за границы V могут его пополнять или вырабатывать сигнал неуспеха. Множество значений может быть пополнено представлениями формул.
BM 	Обработка памяти над таблицей адресов с соответствующими им значениями. Адреса и таблица могут быть значениями или не рассматриваться как значения и использоваться неявно как сущности другой природы. Появляется именование значений и формул, что расширяет класс допустимых формул, обеспечивает многократное использование хранимых результатов и приводит к понятию «данное».
BC 	Безусловное и условное (без «else») управление процессом вычислений, приоритеты в формулах вычисления и переход к очередному действию или по метке.
BS 	Конструирование последовательностей по принципу соседства и перебора слева направо дополняются возможностью возвратов или выбора любого элемента ради обратимости. Как правило это вектора или строки.
XV 	Хранимые данные могут иметь имена, что позволяет решать проблемы укрупнения используемых единиц, воспринимаемых равноправно со скалярами и операциями. Возникают имена логических значений для представления условий выбора хода процесса.
XE 	Композиции операций рассматриваются как безымянные функции, равноправные базовым операциям, они могут

	обрабатывать и вырабатывать не только скаляры, но любые определённые данные, используя упаковку ряда значений в последовательность.
XM 	Именование функций упрощает их многократное использование в формулах, включая представление рекурсии.
XC 	Композиции операций и функций можно рассматривать как одно действие, что приводит к понятию «блок», выделенный скобками, влияющими на порядок вычислений и задающими области видимости имён — иерархия. Появляются системные процедуры, что может приводить к неконсервативным расширениям.
XS 	Структуры однородных данных и процессов сопровождаются дисциплиной доступа к элементам — FIFO, FILO, взаимоисключение, одновременность или др.
DV 	Множество значений пополняется специальными представлениями сигналов «успех-провал» процесса независимо от существования логических значений, а также текстами диагностических сообщений.
DE 	Появляется понятие «предикат» и специальная функция ERROR для выбора обработчиков диагностических ситуаций и продолжения недоопределённых вычислений.
DM 	Появляется типизация значений и данных, а также сигнатур операций и функций, используемая для профилактики неудачных вычислений.
DC	Вводятся специальные схемы управления вычислениями на основе проверки условий соответствия данных и действий для их обработки.
DS 	Появляются структуры с равноправным доступом к разнородным элементам, возможно с их разметкой и указанием кратности вхождения.
PV 	Вводятся мультизначения и специальные значения для представления разного рода неопределённостей, раскрытие или игнорирование которых может быть полезно для продолжения вычислений.
PE	Появляются операции над произвольным числом параметров,

	их распространение на любые структуры данных, проекции формул, отложенные вычисления, формулы доопределения и продолжения вычислений.
PM 	Понятие присваивания распространяется на обмен данными с переферийными устройствами. Возникает идентификация устройств, абстрактные и конкретные имена, паспорта взаимодействий, сигналы готовности действий, время ожидания отклика от устройства, копии и многое другое, отражающее специфику разного оборудования.
PC 	Возникают отображения, средства ввода-вывода, сетевое управление, потоки, эстафета обслуживания, итерирование, условия срабатывания, актив-пассив, администрирование, сервер, ОС.
PS 	одновременность, синхронизация, взаимоисключения, сигналы и сообщения, пакеты, настройки, сервисы, конфигурации.

3.5. Парадигмальные различия

Парадигмальная декомпозиция описания языка программирования выполняется на основе семантического анализа, результаты которого представляются для каждой парадигмы в отдельной понятийной матрице. Столбцы понятийной матрицы соответствуют категориям реализационно различимых семантических систем. Строки матрицы отражают слои раскрутки реализации языка — это подъязыки или диалекты, такие как ядро, расширение, границы вычислений и обобщение их контекста для связи с внешним миром. При выполнении семантического анализа желательно исключить синонимы, чтобы результаты можно было использовать при оценке сложности ЯСП. Синонимами предлагается считать конструкции, добавление-исключение которых из языка не влияет на сложность его реализации. Другая сложность выполнения декомпозиции связана с требованием независимости, альтернативности, ортогональности парадигм. Не всегда удаётся чётко определить парадигмальную принадлежность конструкции, поэтому первичная декомпозиция может быть позднее уточнена. Ещё одна сложность связана с границей между семантикой и прагматикой, некоторые звенья которой могут быть неявными или мигрировать с уровня прагматики на уровень семантики при формировании диалектов. В давние времена

Дж.Маккарти утверждал, что системы программирования долдны быть открытыми, потому, что всё, что понадобилось при реализации компилятора, может понадобиться программисту при применении компилятора. Отчасти это можно выразить на примере семантических систем базовых парадигм и других, поддержанных в языке Clisp (табл. 7-13).

Рассматривая любые семантические системы, важно отметить разницу в характере выполнения функций таких систем в различных контекстах. Так, для любого множества данных D, представляющих значения **V** произвольной природы, реализационно различимы схемы функций для методов вычислений E, средств доступа к памяти M через адреса **N**, особенностей управления вычислениями C и коммуникации или обратимой комплексации и структурирования данных S. Это приводит к представлению об основных категориях семантических систем различно реализуемых схем функций F. Исторически на уровне аппаратуры такие категории систем обладали кумулятивным эффектом в порядке «DEMCS - » (см. Таблица 7) — представление чисел, арифмометр, калькулятор, дифференциальный вычислитель, компьютер, причём, аппаратные подсистемы могут взаимодействовать каждая с каждой.

Таблица 7

Ряд категорий семантических систем аппаратного уровня

Подсистема	*Примечание*
D: данные	Данные из множества D представляют значения из V и шкалу прерываний
E: вычисления	Операции по двум-одному значению производят одно или два значения
M: память	Соответствие между адресами из множества N и хранимыми по этим адресам представлениями из множества D для значений из V допускает разные методы доступа к элементам памяти, включая замену хранимых значений, за исключением адреса 0
C: управление	Сравнение значений с нулём позволяет управлять ходом вычислений наряду с передачами по меткам и обработкой прерываний, не считая перехода по порядку
S: коммуникации	Конструирование сложных данных учитывает возможности команд адресации в памяти

Парадигмы программирования можно отличать по приоритетам выбора решений при определении категорий семантических систем в процессе программирования, отмечая парадигмальные различия общих понятий в каждой модели, зависящие от критериев качества программ. Такая разница может быть показана для классических базовых ПП (См. Таблицы 8-11). Для ИПП данные — это адреса и хранимые коды значений, в ООП появляются хранимые методы и сигнатуры объектов, в ФП вместо адресов в памяти может использоваться связывание с любым значением, а в ЛП — с идентификатором. В ИПП и ООП операции в основном унарные или бинарные, а в ФП и ЛП имеется и произвольная арность. Истинностные значения в ЛП включают специальное значение «ESC», позволяющее отличать нормальные значения предикатов от неуспеха в вычислениях, а ФП может в качестве истины использовать любое значение отличное от пустого списка — «NIL». Структуры данных в ИПП могут не рассматриваться как значения, обрабатываемые базовыми средствами, а в ФП такие структуры обрабатываются без особых ограничений. Можно отметить, что базовые ПП различают представления значений, формул, сигналов и контекстов.

При подготовке императивно-процедурной программы доминирует критерий эффективности, понимаемый как оптимальное соотношение пространственно-временных характеристик программы. Поэтому важнейшими считаются средства работы с памятью, в которой размещаются данные и результаты их обработки (1: M). Управление процессом обработки представляется с помощью конструкций ветвления, использующих операции сравнения и предикаты над данными (2: C). Обработка данных рассматривается как изменение состояний памяти в порядке выполнения вычислений (3: E). При необходимости можно реорганизовывать структуру данных и программы (4: S) (см. Таблица 8).

Таблица 8

Парадигмальная шкала ИПП **(MCES -)**

Приоритет	*Подсистема*	*Примечание*
0	D: данные	Значения ограничены размерами их представлений в регистрах памяти по адресам из N. Шкала прерываний не представлена
1	M: память	Работа с памятью без акцента на особенности нуля, разнообразие методов доступа к памяти и обработки

		прерываний
2	C: управление	Кроме аппаратного сравнения значений с нулём, при управлении ходом вычислений используются приоритеты операций и скобки в выражениях наряду с передачами по меткам, но без обработки прерываний
3	E: вычисления	Операции различаются на унарные и бинарные с одним результатом
4	S: комплексация	Можно конструировать сложные данные и выбирать их элементы, используя возможности команд индексной адресации в памяти

В центре внимания ФП полнота семейства методов организации вычислений, доступных для эксперимента. Поэтому всё начинается с выбора базовых средств для обработки символьных представлений сущностей заданной предметной области (1: E). Конструирование сложных объектов освобождено от обязательности соседства элементов (2: S). Работа с памятью в таком случае может не требовать привязки к физическим адресам, а ограничиться представлением ассоциирующей функции над парами данных любой природы (3: M). Управление процессом вычислений может рассматриваться как функция над фрагментами программы, рассматриваемыми как разновидность значений (4: C) (см. Таблица 9).

Таблица 9

Парадигмальная шкала ФП **(ESMC -)**

Приоритет	*Подсистема*	*Примечание*
0	D: данные	Представления значений не ограничены по размеру и сложности, включая функции
1	E: вычисления	Некоторые операции могут обрабатывать любое число параметров и вырабатывать ряд значений, при необходимости объединяемых в сложное данное
2	S: структуры	Можно конструировать произвольно сложные, равноправные с элементарными, данные, все элементы которых доступны с помощью функций в любом выражении

3	M: память	При обработке сложных данных старые значения не изменяются, а новые значения располагаются в памяти независимо, причём соответствие между ассоциированными данными хранится в памяти, допуская изменение ассоциации
4	C: управление	Любое вычисление можно заблокировать или запустить. Программа может содержать точки ветвления из произвольного числа ветвей, выбираемых сравнением результата с «нулём» (NIL), входящим в множество значений

В случае ЛП важно показать существование хотя бы одного полезного решения задачи. Это приводит к доминированию логики недетерминированного поиска выполнимых решений (1: C). Формально вычисляются варианты возможных решений (2: E) в произвольном порядке. Но в качестве структур используются образцы (3: S), позволяющие управлять выбором вариантов. Именуются фрагменты с фиксированным числом параметров (4: M), от необходимости имён которых техника образцов освобождает. Строго говоря это ещё не программирование, а лишь подготовка к нему.

Ведущий критерий качества программ в ООП – сопоставимость иерархии классов объектов с иерархией понятий в области приложения программ, позволяющая готовые программные решения пополнить новыми для расширения прежней области приложения. Поэтому здесь всё начинается с определения иерархии классов объектов (1: S), размещаемых по фиксированным адресам в памяти (2: M), применяемым как указатели ради достижения некоторого уровня эффективности. Управление процессом обработки данных использует сопоставление классов объектов и допустимых методов обработки объектов, размеченных правами доступа из разных частей программы (3: C). Вычисления происходят лишь при успешном сопоставлении и соответствии прав доступа к объектам (4: E) (см. Таблица 9).

Таблица 10

Парадигмальная шкала ООП **(SMCE)**

Приоритет	*Подсистема*	*Примечание*
0	D: данные	Данные представляют не только значения, но и методы их обработки
1	S: структуры	Классы объектов приспособлены к доопределению и наследованию по иерархии классов

2	M: память	Дозированный доступ к элементам объектов сопровождается механизмами неявных обработчиков ситуаций, адреса могут быть значениями
3	C: управление	Выполнимость методов над объектами обусловлена проверкой их совместимости и прав доступа по иерархии классов
4	E: вычисления	Операции бинарные и унарные, также как любые вычисления, можно перегрузить, добавив обработку возможных прерываний

Подробный анализ семантики ООП, сопровождённый сопоставлением с другими ПП и частичной формализацией основных механизмов реализации ЯП, представлен в работе [6].

Показывая отличия в схемах определения функций для разных категорий семантических систем в зависимости от ПП, связанных с различием критериев качества программ и приоритетов используемых в их реализации средств, следует отметить, что переход от ЯП к СП обычно сопровождается расширением числа поддержанных ПП. При определении языка Haskell это привело к понятию «монада», позволяющему любому ЯП достигать практичности на уровне СП, что обычно и выполнялось с помощью библиотечных модулей.

Кроме сравнительно ясных классических базовых ПП, есть основания выделять основные инструментальные системно **расширяющие** ПП, нацеленные на подготовку и конструирование программ, операционных систем (ОС) и баз данных (БД), поддержку работы с файлами и разнообразными конфигурациями устройств, а также обеспечение обратной связи при выполнении любых программ. Прежде всего это ПП, поддерживающие системное программирование, использующее синтаксически и грамматико ориентированную обработку определений ЯП при конструировании СП (VDM, BNF в инструментах типа Lex/YACC и их аналогах во многих новых ЯП), мета-программирование (Рефал), языки и системы программирования и разработки программных инструментов (Bliss, ЯРМО, C), создание промежуточных форм для поддержки особо сложных работ (внутренний язык системы БЕТА), отладки программ и их тестирования.

Все расширяющие ПП работают со много более сложными элементами, обладающими своей жизнью, допускающими включение во многие системы и конфигурации, в которых возможно изменение их состояния. Представления данных включают в себя, кроме сложных

структур данных, формальных определений и кодов, процессы, устройства, роли участников и комплексы. Методы обработки элементов и их взаимодействия подчинены более жёстким требованиям правильности, что влечёт поддержку улучшения элементов по частям, то есть целенаправленного развития по мере выявления ошибок или необходимости в повышении эффективности. Имеет место разделение труда по уровню квалификации и ответственности.

Так, например, в области разработки СП характерно деление на разработчиков СП и применяющих СП программистов, работающих в рамках базовых ПП и при необходимости подключающих расширяющие ПП в форме побочных эффектов библиотечных модулей. Появление большого количества DSL-языков на базе технологии Clang-LLVM показывает новый уровень сформировануости парадигм системного программирования, стирающий эту границу. Предметно-ориентированные DSL-языки заслуживают отдельного рассмотрения именно как новый уровень программистского языкотворчества. Если при развитии аппаратуры успешный опыт накапливается в форме системы команд, а в обычных ЯП накопление опыта программирования выполняется в форме отлаженных процедур, то конструирование DSL — механизм накопления опыта решения задач применения ИТ в форме языков программирования, что допускает включение такого механизма в инструментально расширяющие ПП.

Таблица 11

Парадигмальная шкала средств разработки СП - Мета **(ECMS** **)**

Приоритет	*Подсистема*	*Примечание*
0	D: данные	В качестве значений выступают определения реализуемого ЯП и система команд целевой машины, на которой предстоит выполнять программы, подготовленные на базе данного ЯП
1	E: вычисления	Синтаксически управляемый автомат с переводом строит переход от ЯП к его абстрактному представлению, более удобному для дальнейшей обработки программ и кодогенерации. Константные вычисления и символьные преобразования могут выполняться в этом процессе и заменяться на результат с целью повышения эффективности программ
2	C:	Проверка текстов на принадлежность ЯП сопровождается

	управление	диагностичесими сообщениями и, возможно, рекомендациями по продолжению разбора текста. Результаты лексического, синтаксического и семантического анализа проверяются на соответствие шаблонам предстоящей кодогенерации. Определение ЯП работает подобно представлению типа данных. Кодогенерация может выполняться автономно. При выполнении созданной программы могут быть дополнительные проверки правильности вычислений.
3	M: память	Заполняется встроенная база данных, формируемая как таблицы, сопоставляющие адреса или идентификаторы отдельным конкретизациям понятий, выделяемых при анализе текста, или их атрибутоам. Возможно привлечение промежуточных представлений для решения сложных технических задач. Строится схема программы, которая может быть достаточной для её выполнения или для кодогенерации
4	S: структуры	Используются графовые и кодовые представления понятий, выделяемых при анализе текстов на принадлежность ЯП и выводе атрибутов, полезных для предстоящей кодогенерации и исполнения программы

Происходившее автономно развитие средств и методов работы с базами данных в настоящее время активно интегрируется в общий контекст ИТ, создаёт реализационную поддержку методам искусственного интеллекта и представления знаний с привлечением экспертов (GPS, RDF, OWL), является основой мощного пространства производственных и бизнес технологий, особенно на базе Интернет-сервисов. Сформировано разделение труда на администраторов, ответственных за хранение данных, и клиентов, интересующихся их содержанием, способных его применять и уточнять. Отдельные методы из этой сферы, изначально накапливающей решения по надёжности обработки данных, перемещаются в пространство решений при создании новых языков программирования.

Таблица 12

Парадигмальная шкала СУБД — декларативное программирование **(MSEC-** **)**

Приоритет	*Подсистема*	*Примечание*
0	D: данные	Данные представляют объекты реального мира и их характеристики, а также сведения о правах доступа к данным

1	M: память	Хранение содержательных представлений в форме отношений дополняется использованием имён, индексов, функций, позволяющих повышать эффективность доступа к данным. Долговременным хранилищем данных являются файлы. На время обработки данных создаётся их временная копия в оперативной памяти.
2	S: структуры	Отношения между хранимыми данными могут быть представлены как записи, организованные в таблицы, графы, иерархии (деревья), сети
3	E: вычисления	Операции обеспечивают формирование запросов, направленных на доступ к накопленным данным, включая их обработку
4	C: управление	Кроме прав доступа к хранимым данным используются предикаты и логические выражения. Имеются рекомендации по учёту так называемых «нормальных форм», позволяющих повысить эффективность хранения данных и доступа к ним, причём некоторые из них дают одновременный выигрыш по скорости и по объёму памяти.

Работы по операционным системам весьма осторожно переходят к использованию языков высокого уровня, преимущественно скриптовых и интерпретируемых, пригодных для реагирования в динамике реального мира. Возникает смежная линия компонентного программирования (Com/Dcom, Corba, SOAP, .Net), поддерживающая ООП при решении вопросов, требующих повышенной квалификации.

Таблица 13

Парадигмальная шкала ОС — рефлексивное программирование **(CMSE -)**

Приоритет	*Подсистема*	*Примечание*
0	D: данные	Данные представляют задания, процессы, файлы, каталоги, устройства, протоколы и настроечные значения, а также права доступа.
1	C: управление	Система функционирует как взаимодействие приаппаратного ядра и пользовательской оболочки. Имеется контроль прав доступа с учётом приоритетов и успеха-провала в ранее выполненных действиях, включая наличие-отсутствие необходимых устройств и файлов. Выполнение заданий инициируется с уровня командной строки.

2	M: память	Основное хранение данных в файлах сопровождается использованием очередей, приоритетов, шкалы допустимых прерываний и настроечных значений, размещаемых при загрузке системы и допускающих уточнение привилигированным системным администратором
3	S: структуры	Основная структура — иерархия файлов, дополненная очередями, строками, каналами и кодами, включая данные о регистрации пользователей и их правах доступа
4	E: вычисления	Работу операций выполняют доступные из командной строки команды уровня ядра, дополняемые представлениями программируемых заданий уровня оболочки

Расширение спектра новой аппаратуры, пригодной для массового применения без технического сопровождения, привела к задачам формирования программно-аппаратных многопроцессорных комплексов для расширения сфер применения ИТ.

Не менее заметно выделяется группа **неограниченных** коммуникационно интерфейсных ПП, поддерживающих обработку больше-объёмных данных (bigdata, sematic-web, rdf), дистанционную работу в сетях, сервис-ориентированное программирование на базе языков разметки и переписывания (html, XML, PHP), параллельные, векторно-ориентированные для обработки массивов (APL) или поддерживающие теоретико-множественные инсерционное механизмы, включая динамические вставки-замены (SETL) и высокопроизводительные вычисления на суперкомпьютерах (OpenMP, mpC) и мобильных устройствах (см. Таблицы 14-17). Языков программирования, поддерживающих неограниченные ПП, пока создано немного, но возможно их число будет возрастать, пока не получит удобной формы синхронизация процессов над общей памятью. За редким исключением потребность в неограниченных ПП реализуется в форме отдельных специализированных программных инструментов.

Переход к обработке больше-объёмных данных пока развивается на уровне специализированных языков и инструментов, практику применения которых несколько сдерживает отсутствие методов декомпозиции, преобразования и проектирования сложных формирований, обладающих трудно контролируемой динамикой обновления. Большинство решений сводится к выбору удобной визуализации графов.

Всеохватность методов дистанционного доступа резко влияет на качество жизни большинства слоёв населения, что сопряжено с вопросами социальной ответственности. Имеется эффект механизмов типа моды и рекламы.

Постановки задач ПВ учитывают существование областей приложения, в которых недостаточна скорость получения результатов по доступным программам решения конкретных задач. Парадигмы этого направления находятся в стадии формирования из-за сложности перехода к масштабируемым решениям языков сверх высокого уровня, допускающим автоматическую настройку на реальные конфигурации оборудования, а также из-за отсутствия традиции представлять течение времени в математических моделях ЯП.

Высокопроизводительные вычисления (ВПВ) расширяют спектр методов управления вычислениями и критериев эффективности программ благодаря возможности использования процессорных резервов и наследования ранее отлаженных программ, как правило посягая на их неприкосновенность. Это позволяет рассматривать ВПВ как бесспорный полигон для признания потенциала методов верификации, есть основания такие методы рассматривать как важную составляющую неограниченных парадигм.

В последние годы проявляются причины обуславливать конструирование средств верификации программ формализацией используемых ПП, а программистские проекты сопровождать обоснованием выбора не только инструментария, но и ПП, чтобы избегать межпарадигмальных конфликтов, чреватых трудно уловимыми ошибками, связанными с изменением обстановки функционирования программируемых компонентов.

Язык Clisp поддерживает парадигмы функционального, рефлексивно-экспериментального, императивно-процедурного, декларативного, объектно-ориентированного и мета-программирования. Различие между парадигмами зависит от степени изученности решаемых задач и развития методов их решения. Парадигма **функционального** программирования нацелена на удостоверение вычислимости заданных формул, парадигма **рефлексивно-экспериментального** программирования обеспечивает гибкость вычислений и отладку в зависимости от условий применения решений, парадигма **императивно-процедурного** программирования позволяет повышать эффективность вычислений, парадигма **декларативного** программирования допускает создание онтологии реализационных решений, парадигма **объектно-ориентированного**

программирования использует модели области приложения, а парадигма **мета-программирования** предназначена для перехода от решения отдельной задачи к универсальным решениям классов подобных задач.

3.6. Термины языка Clisp

Atom (**атом**) - данное, не делимое средствами языка, представляющее подмножество значений, возможно имеющих смысл в математике или в иных сферах. Каждому атому соответствует уникальный код, используемый для его идентификации.

List (**список**) — данное, представляющее составные занчения из заключённого в круглые скобки перечня выделяемых элементов — произвольного числа, возможно ни одного.

NIL = () - **пустой список** по определению рассматривается одновременно и как атом, и как список.

Symbol **(символ) —** данное, представляющее атом, обладающий определёнными свойствами, системными или программируемыми.

Object (**объект**) - данное, представляющее символ со свойством «класс».

Interpretation (**Интерпретация**). - данное, представляющее реализацию функции или её определения как правила сопоставления входных данных значениям языка.

Tree (**дерево**) — составное данное, представляющее произвольную конструкцию из пар любых данных.

A-list (**ассоциативный список**) — составное данное, представляющее список пар любых данных.

Property (**свойство**) — данное, используемое при обработке символов для различения их категорий, представленных индикаторами.

Indicator (**индикатор**) — данное, представляющее символ, выделяющий свойство - системные (встроенные) индикаторы expr, fexpr, subr, fsubr, value.

Form (**формы**) - данное, представляющее выражение, которое может быть вычислено при подходящих условиях, обычно символ или список.

Function **(Функция) —** данное, представляющее соответствие между аргументом и результатом.

Functional **(Функционал)—** функция, аргументом или результатом которой может быть функция.

Map **(Отображение)** — функция, аргументом которой может быть другая функция, серийно применяемая к элементам структуры данных.

Filter **(Фильтр)** — функция, серийно применяющая другую функцию к элементам структуры данных, не все результаты которой включаются в итоговый результат.

Higher Order Functions **(Функции высших порядков)** — функция, аргументом которой может быть другая функция.

Place (**адрес**) — уникальный адрес для размещения свойств символов, представляющих переменные, включая их значения и определения.

Self (**результат**) адрес данного, представляющего символ для размещения результата применения метода в ООП.

Clozure (**замыкание**) - составное данное, представляющее конструкцию из определения фукнции и списка значений свободных переменных.

Flag (**флаг**) - индикатор без свойства.

Program (**программа**) - данное, представляющее список вычисляемых форм.

Generalized variables (**обобщённая переменная**) — данное, предназначенное для доступа и обработки по определённому адресу.

Evaluation (**вычисления)** — интерпретация, сводящая формы к данным, представляющим их значения.

Transformation (**Преобразования**) - пара форм, эквивалентных в определённом отношении.

Message (**сообщение**) — данные для управления процессом выполнения программ, а также диагностических сообщений и организации диалога: «Нет переменной», «Нет функции» , «Нет переменной», «Нет метки» , «Нет метода», «Не совпадают права доступа», обычно это атомы или строки.

Trace (**слежение**), - системный флаг отладки функций, чтобы видеть их аргументы и результаты.

Verification (**Верификация)** — проверка программ на эквивалентность спецификации, логике или модели.

Data type (**Типы значений)** - встроенные, системные предикаты, выделяющие коды преставления данных в памяти, требующие особой обработки, отличающейся от обработки других типов.

Sequence (**последовательность**) — структуры данных, приспособленные к последовательному доступу.

Psewdo-atom (**псевдо-атом**) — неделимое данное, свойства которого не требуют списка свойств в силу их само-определимости.

Number (**Числа**)— псевдо-атомы: character, strings, ixnums. Ignums. Ratios, characters, flonums complex, COMPILATION OPTIONS — флаги управления.

Structure (**структура**) - составное данное из конечного числа полей произвольного типа, доступ к которым определён.

Package (**пакет**) — сложный объект, предоставляющий в рамках определённого пространства имён доступ к данным, обеспечивающим решение определённого класса задач.

Name spaces (**пространство имён**) - список атомов со списками свойств, обеспечивающих решение определённого класса задач в рамках заданного пакета.

String stream (**потоки&строки**) - данные для последовательного доступа - данное, представляющее обработчики, разновидность последовательностей.

Hash table (х**эш-таблица**) - составное данное, представляющее список уникальных данных, таблица расстановки, используется и при идентификации атомов.

Sequence (**последовательность**) - составное данное, представляющее элементы для последовательного доступа .

Compiler (**компилятор**) – функция перехода от символьного определения отлаженной функции к эквивалентному ей машинному коду.

Compiler compiler (**Компилятор компиляторов)** — функция, способная по данным, представляющим определение компилируемого ЯП, построить для него компилятор.

Super (**мета-компиляция** или **супер-компиляция**) — специальная техника оптимизации алгоритмов, основанная на знании конкретных входных данных алгоритма. Супер-компилятор принимает исходный код алгоритма плюс некоторые данные о входных параметрах и возвращает новый исходный код, который исполняет свою задачу на этих данных быстрее или является лучше исходного алгоритма по каким-то другим показателям.

Projections **(Проекции) —** результат частичного выполнения программы, зависящий от определённого параметра.

Generic (**обобщённое данное**) - составное данное, представляющее шаблоны для подстановки разных типов данных.

File (**файл)** — объект для долговременного хранения произвольных данных.

System (**система**) — универсальная программа, допускающая понимание решаемых задач сводить к одному слову независимо от сложности решения и определения (К.Лоренц).

Partial calculations (**частичные вычисления)** — вычисления с учётом заранее определённых условий или разметки.

Lazy evaluation (**Ленивые вычисления**) — отложенные вычисления, допускающие выполнение по мере необходимости.

4. Парадигмальная декомпозиция языка Clisp

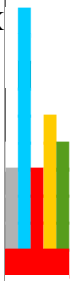
4.1. Функциональное программирование (ESMC -) предназначено для решения задач, обладающих исследовательским компонентом, требующим отладки алгоритма и проверки его работоспособности в эксперименте. Согласно этой парадигме обработка данных начинается с представления атомов как уникальных именованных указателей, для реализации которых используется неявная хэш-таблица. Атомы можно сравнивать на равенство указателей. Программа строится как иерархия выражений, на каждом уровне которой происходит связывание имен и с их определениями как данных или безымянных функций в зависимости от синтаксической позиции. Возможна блокировка выражений и возобновление вычислений. Определения функций создают локальные области видимости связанных переменных с помощью ассоциативных списков, что гарантирует неизменяемость данных. Интерпретатор допускает определения функций с помощью машинного кода, для реализации сопряжения с которым используется неявный вектор аргументов. Такие функции привлечены для создания структур данных, используя атом Nil как представление пустого списка и значения «ложь». Поддержано неявное освобождение памяти — «сборка мусора» (GC).

Таблица 14

Понятийная матрица ФП языка Clisp

№ *категории*	**V**	**E**	**M**	**C**	**S**
диалекта	V	**F: V* → V**	**AL : Atom → V**	**(F → {NIL, T})***	**A ↔ K**
B	Ядро языка				
	() NIL и другие **атомы**	**eq:** V V→{NIL,~Nil}	((λ (x1 x2 ...) expr) a1 a2 …) GC	(eval <expr>) (quote <expr>)	cons: V1 V2 → (V1 . V2) car:

4 Высота цветного столбика символизируют число функций соответствующей категории.

	 11223[5]					(V1 . V2) →V1 cdr: (V1 . V2) →V2
		Консервативное расширение				
X	 39354	Sexpr: (V . V) **список** tree, a-list	atom (Fn a1 a2 ...) ((λ (x1 x2 ...) expr) a1 a2 …) apply evlis evcon **FunCall** append, last	(**acons** &optional &rest &key [&aux &allow-other-keys]] **замыкание** **Defun**	cond (if <t> <e1> <e2>) tailp map **reduce**	**pairlis** (V . . .) ; list, list* **assoc, rassoc** **nth, nthcdr**
		Полный динамический и типовый контроль				
D	 35541	«Нет значения», «Нет функции» subrs (built-in functions) fsubrs (special forms) user defined functions	null (functionp <s>) ERROR Strings functions Predicate functions	let, Let*, flet, labels Свободные переменные **(complement <fun>)** expr, fexpr subr, **fsubr**	lazy BREAK continue **Типы значений**	Строки
		Псевдо-атомы. практичный компромисс				
P		Числа character strings fixnums (integers) bignums (integers) ratios characters **flonums** (floating point numbers) **complex**	+ - * / arithmetic functions Strings, characters, numbers of any type,	Псевдо-функции и свойства: subrs, fsubrs, put, get, remprop closures evaluate	command loop REPL= ((λ (Nil wrk) (eval wrk) Nil '(print (eval (print (read)))) **))**	PRINT READ

5 Цифры символизируют высоту столбика — числовая характеристика.

	94222	**numbers**				

При консервативном расширении поддерживается возможность символьного определения именованных функций и их захамыканий, произвольных ветвлений, включая отображения,свёртки и фильтры, расширяется спектр функций обработки списков. На уровне динамического управления вычислениями используются строки для диагностических сообщений и функция ERROR для продолжения вычислений, поддержаны ленивые вычисления, вводятся разные виды областей видимости и полный контроль типов значений. Практичный компромисс пополняет спектр обрабатываемых данных и элементарных мультиопераций над списками из них, поддерживает программируемые свойства атомов, стандартный цикл организации вычислений и средства ввода-выводе данных как функции.

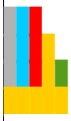
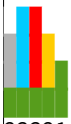
Общая числовая характеристика имеет вид (**`11223=9 39354=24 35541=18 94222=19`**). По ней, не вникая в детали, можно сделать вывод, что парадигма ФП в языке Clisp имеет сравнительно маленькое ядро B и оно заметно расширяется в диалектах X, D, P.

4.2. Мета-программирование (**ECMS -**) поддерживает само-применимость программ, точнее, их обработку как данных, что позволяет этой парадигме выполнять роль встроенного препроцессора, поддерживающего формирование и моделирование разных схем вычислений.

Таблица 15

Понятийная матрица META языка Lisp

№ диалекта	*категории*	**V** V	**E** F: V* → V	**M** AL : Atom → V	**C** (F → {NIL, T})*	**S** A ↔ K
B	11112	Введение и вызов макро-определений				
		list абстракт-ная интер-претация	(Fmacro A B …)	(defmacro <sym> <fargs> <expr>...)	complement	backquote comma-at, comma
X	12322	Преобразование и копирование разных видов списков				
		program преобра-зования	subst sublis	nsubst nsublis copy-tree	nsubst-if nsubst-if-not	copy-list copy-alist
		Унификация обработки строк со средствами ввода-вывода				

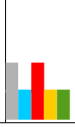
D	 33321	*NIL* string *STREAM*	make-string stringp *частичные вычисления,*	(macrolet (<binding>...) <expr>...) read-from-string read-line	macroexpand macroexpand-1	(princ-to-string <expr>)
P	 23321	Специализация на проблемы компиляции программ				
		string stream суперком-пиляция	string functions stream functions hash-tables	Make-string-input-stream, with-output-to-string get-output-stream-list, get-output-stream-string make-string-output-stream)	(y-or-n-p [<fmt> [<arg>…]]) (yes-or-no-p [<fmt> [<arg>...]])	(pp-def <funct> [<stream>])

Общая числовая характеристика мета-программирования в языке Clisp (**`11112=6 12322=10 33321=12 23321=11`**) при поверхностном взгляде показывает, что его понятийная сложность меньше, чем сложность ФП.

4.3. Декларативная парадигма программирования (**MSEC -** ) выполняет системообразующую роль на основе встроенной триадной БД, допускающей расширение в процессе выполнения программы и пополнение заранее подготовленными пакетами. Это встроенная база данных, приспособленная к хранению и обработке любых понятийных систем, расширяющих исходные понятия ЯП

Таблица 16

Понятийная матрица определения БД языка Pure Lisp

№	*категории*	**V**	**E**	**M**	**C**	**S**
диалекта		V	**F: V* → V**	**AL : Atom → V**	**(F → {NIL, T})***	**A ↔ K**
B	 21211	Неявное данное «поле» = «адрес»				
		Place - адрес indicator	(getf <place> <prop> [<dflt>])	(remf <place> <prop>) (make-symbol <pname>)	(symbol-name <sym>)	property list. (symbol-plist <sym>)
X		Символ как атом, обладающий свойствами				
		symbol Строки	(get <sym> <prop> [<dflt>])	(putprop <sym> val> <prop>) (remprop	(declare [<declaration> ... .])	gensym

	21211			<sym> <prop>)		
D	42321	Категории символов				
		NIL Expr, Subr Trace Value	Atom: V→{NIL,T} apply: F V* AL→V	(mark-as-special <sym> [<flag>]) (defconstant (defvar	(symbol-function <sym>) (symbol-value <sym>)	**apropos**
P	43342	**Пространсва имён**				
		package **string** **symbol** **object**	(use-package <pkgs> [<package>]) (in-package <package>) (do-all-symbols (<var> [<result>]) <expr>...)	(defpackage <package> [<option>…]) (delete-package <package>) (make-package <package> &key :nicknames :use)	(symbol-package <symbol>) (package-valid-p <package>) (find-symbol <string> [<package>]) (find-package <package>)	(package-use-list <package>) (list-all-packages)

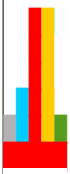
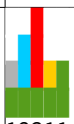
Числовая характеристика декларативной парадигмы в языке Clisp (**21211=11 21211=11 42321=12 43342=16**) показывает, что основная понятийная нагрузка сосредоточена на обеспечении практичности.

4.4. Рефлексивно-экспериментальное программирование пронизано реактивностью — в любой позиции можно построить список, представляющий выражение или определение, и исполнить его. Такое функционирование похоже на работу операционных систем, что позволяет поддерживать взаимодействие с любыми средствами внешнего мира.

Таблица 17

Понятийная матрица рефлексивно-экспериментального программирования на CLisp

№	*Категории*	**V**	**E**	**M**	**C**	**S**
Диалекта		V	**F: V* → V**	**AL : Atom → V**	**(F → {NIL, T})***	**A ↔ K**
B	11122	Конструирование обработчиков				
		Атомы, списки	Function	FunCall **(funcall <fun> <arg>...)**	Null step	List nth, nthcdr

X	 12551	Отслеживание процессов				
		формы expr fexpr subr fsubr	evalhook applyhook	(baktrace [<n>]) trace untrace (debug) (nodebug)	ecase ccase etypecase ctypecase check-type	command loop
D	 12253	Диагностика и контроль				
		bug fixes and extensions	cerror errset	(clean-up) (top-level)	assert <test> break command loop catch throw (unwind-protect <expr> <cexpr>...)	(pp <object> ...) (pp-def <funct> (pp-file <file> [<stream>])
P	 12311	Внешний мир, практичный компромисс				
		compilation options	**structure functions** multiple value functions	file i/o functions compile **system functions**	inspect.lsp commands (all …)	input/output functions

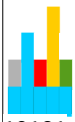
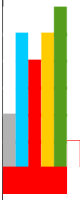
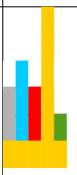
Появляется явный вызов функций, проверка пустоты списков и пошаговое исполнение вычислений и другие возможности наблюдения, расширяются средства обработки списков, дополняются возможности динамического контроля и взаимодействия с внешним миром. Числовая характеристика рефлексивной парадигмы в языке Clisp (**`11122=7 12551=14 12253=13 12311=8`**) показывает равномерное распределение понятийной сложности.

4.5. Императивно процедурная парадигма предназначена для повышения эффективности и производительности программ, алгоритмы которых предварительно отлажены в рамках функциональной парадигмы программирования. Улучшение или оптимизация достигается техникой процедур, глобальных переменных, деструктивных аналогов функций, полной динамической проверкой типов данных и поддержкой структур данных, типичных для многих производственных языков программирования.

Таблица 18

Понятийная матрица ИПП языка CLisp (**MCES -**)

№	*категории*	**V**	**E**	**M**	**C**	**S**
диалекта		V	F: V* → V	AL : Atom → V	(F → {NIL, T})*	A ↔ K
		Функциональная модель императивно процедурной техники				

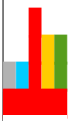
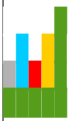
B	 12131	список	(prog …), progv prog1, prog2, progn	(setq ..), set	return, return-from <name> Label (go <sym>)	процедура *список рабочих переменных, список меток*
		Распространение прагматики ИПП: Destructive functions				
X	 25456	generalized variables = sym place	nth elt aref get getf delete, fill, replace nsubst, nsublis, nintersection, nunion, nset-difference, nset-exclusive-or	(defsetf), (setf),psetq, push, pop incf, decf (block..)(tagbody ..)	loop, do , do*, dolis, dotimes	Rplaca, rplacd nconc, nreconc, nbutlast, sort, map-into, mapcan, mapcon, nreverse,
		Введение строгого полного динамического контроля типов данных				
D	 23251	«Нет переменной» «Нет метки»	ERROR, cerror, errset break, continue	gethash delete-if, delete-if-not nsubst-if, nsubst-if-not	and, or when, unless case, typecase unwind-protect check-type	the format function
		Расширение входного языка практичными структурами данных				
P	 56535	object array hash table sequence structure	object functions array functions hash table functions aref hash-table-count maphash	(generic <expr>) make-array vector make-hash-table gethash, remhash	typep every, notevery, some, notany	generic print prin1, princ pprint terpri fresh-line

Числовая характеристика императивного программирования в языке Clisp (**12131=8 25456=22 23251=13 56535=24**). По сумме она сравнима со сложностью ФП.

4.6. ООП (SMCE -  **)** нацелен на улучшение и развитие готовых решений, что в языке Clisp поддержано в форме пакета CLOS. Можно сказать, что ээто пратичный компромисс с доминирующей технологией производственного программирования.

Таблица 19

Понятийная матрица ООП определения языка Pure Lisp

№	категории	V	E	M	C	S
диалекта		V	F: V* → V	AL : Atom → V	(F → {NIL, T})*	A ↔ K
		Виды данных для ООП				
B	 21111	**объект класс**	**self'**	symbol '**self'**	show	'Class' is the **class** of
		Области видимости и управление с помощью сообщений				
X	 11322	**Name spaces**	answer the **evaluator** binds the symbol 'self'	**defclass defmethod definst**	Messages: defined for the super-class. If no method is found, an error occurs.	**superclass** send Class :new
		Привлечение хэш-таблиц				
D	 13331	**message.**	Messages: self - object - constant, *obarray* - the object hash table.	multiple object arrays **(name spaces)** Типы значений	The Object method **:show** (typep <expr> <type>) (type-of <expr>)	(generic <expr>)
		Формирование пакетов и взаимодействий				
P	 12123	system пакеты	OBJECT FUNCTIONS: send send-super	(hash <expr> <n>)	inspect.lsp Structure elements can be: examined and changed.	The file object the OPEN function a FILE-STREAM object

Числовая характеристика ООП в языке Clisp (**21111=6 11322=8 13331=11 12123=9**)=34 показывает, что эта парадигма мало увеличивает понятийную сложность языка.

Представленные результаты семантического анализа парадигм программирования, поддержанных в языке Clisp можно представить сводной таблицей числовых характеристик их понятийной сложности.

Таблица 20

Сводка числовых характеристик парадигм языка Clisp.

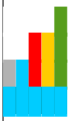
Парадигма	*Характеристика*	*Сложность*	*Примечание*
ФП	**11223 39354 35541 94222**	70	Расширение и границы
Препроцессор	**11112 12322 33321 23321**	39	Лаконичный базис

БД	**21211 21211 42321 43342**	50	Практичность и *границы*
ОС	**11122 12551 12253 12311**	42	Расширение и контроль
:=	**12131 25456 23251 56535**	67	Практичность и расширение
ООП	**21111 11322 13331 12123**	34	Контроль, доступ и внешний мир

4.7. Результаты сравнения парадигм по слоям прагматики

Таблица 21

Матрица сравнения **ЯДЕР (V E M C S)** парадигм языка Clisp

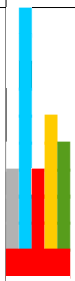
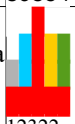
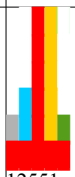
№ категории		**V**	**E**	**M**	**C**	**S**
ПП		V: 21	F: V* → V	AL : Atom → V	(F → {NIL, T})*	A ↔ K
ФП	 11223	NIL и другие атомы	eq: V V→{NIL,~Nil}	((λ (x1 x2 ...) expr) a1 a2 …) GC	(eval <expr>) (quote <expr>)	cons: V1 V2 → (V1 . V2) car: (V1 . V2) →V1 cdr: (V1 . V2) →V2
Мета	 11112	List, () абстракт-ная интер-претация	(Fmacro A B …)	(defmacro <sym> <fargs> <expr>...)	complement	backquote comma-at, comma
БД	 21211	**Place - адрес indicator**	(getf <place> <prop> [<dflt>])	(remf <place> <prop>) (make-symbol <pname>)	(symbol-name <sym>)	property list. (symbol-plist <sym>)
OS	 11122	данное	Function	(funcall <fun> <arg>...)	Null step	List nth, nthcdr
ИП :=	 12131	список 12133	(prog …), progv prog1, prog2, progn	**(setq ..), set**	return, return-from <name> Label (go <sym>)	процедура рабочие переменные метки
ООП		объект класс	**self**	symbol 'self'	show	'Class' is the class of

	21111					

Достаточно беглого взгляда, чтобы оценить число понятий на уровне ядра — от пяти до десяти. Видно какие механизмы потребовали дополнительных понятий. Так, для декларативной парадигмы (БД) понадобился неявный тип данных «поле», для ООП объект **self**, для ИП функции **setq и set**, имитирующие присваивание в локальной области видимости, ограниченной функцией prog.

Таблица 22

Матрица для сравнения средств **расширения (X)** языка Clisp

№	*катего-рии*	**V**	**E**	**M**	**C**	**S**
ПП		V: 21	F: V* → V	AL : Atom → V	(F → {NIL, T})*	A ↔ K
ФП	 39354	Sexpr: (V . V) **список** tree, a-list	atom (Fn a1 a2 ...) ((λ (x1 x2 ...) expr) a1 a2 …) apply evlis evcon **FunCall** append, last	(**acons** &optional &rest &key [&aux &allow-other-keys]] **замыкание** **Defun**	cond (if <t> <e1> <e2>) tailp **map** **reduce**	pairlis (V . . .) ; list, list* assoc, rassoc **nth, nthcdr**
Мета	 12322	program преобра- зования	subst sublis	nsubst nsublis copy-tree	nsubst-if nsubst-if-not	copy-list copy-alist
БД	 21211	symbol **Строки**	(get <sym> <prop> [<dflt>])	(**putprop** <sym> val> <prop>) (**remprop** <sym> <prop>)	(declare [<declaration> ...])	**gensym**
OS	 12551	формы expr fexpr subr fsubr	evalhook applyhook	(baktrace [<n>]) **trace** untrace (debug) (nodebug)	ecase ccase etypecase ctypecase check-type	command loop **REPL**
ИП :=		generalized variables = sym	nth elt aref get getf delete, fill,	(defsetf), (setf),psetq, push, pop	**loop,** **do ,** **do*,**	**Rplaca, rplacd** **nconc,** nreconc, nbutlast,

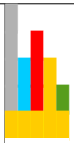
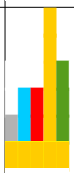
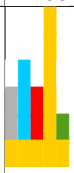
	 25456	place	replace nsubst, nsublis, nintersection, nunion, nset-difference, nset-exclusive-or	incf, decf (block..) (**tagbody** ..)	**dolis, dotimes**	sort, map-into, mapcan, mapcon, nreverse,
ООП	 11322	Name spaces	answer the evaluator binds the symbol 'self'	defclass defmethod definst	Messages: defined for the super-class. If no method is found, an error occurs.	superclass send Class :new

Сразу видно, парадигмы языка Clisp сильно отличаются по мощности комплекта средств расширения языка — от 8 до 24 и что ФП лидирует по разнообразию методов организации вычислений. Появляются отображения, замыкания, глобальные определения функций. Кроме того, генерация уникальных атомов, деструктивные функции и циклы.

Таблица 23

Понятийная матрица сравнения **Граней (D)** парадигм языка Clisp

№ столбца		**V**	**E**	**M**	**C**	**S**
ПП		V	F: V* → V	AL : Atom → V	(F → {NIL, T})*	A ↔ K
ФП	 35541	«Нет значения», «Нет функции» subrs (built-in functions) fsubrs (special forms) user defined functions	null (functionp <s>) ERROR Strings functions Predicate functions	let, Let*, flet, labels Свободные переменные **(complement <fun>)** expr, fexpr subr, **fsubr**	lazy BREAK continue **Типы значений**	Строки
Мета	 33321	*NIL* string **STREAM**	make-string stringp *частичные вычисления*,	(macrolet (<binding>...) <expr>...) read-from-string read-line	macroexpand macroexpand-1	(**princ-to-string** <expr>)

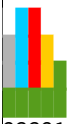
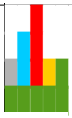
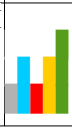
БД	 42321	NIL Expr, Subr Trace Value	**Atom**: V→{NIL,T} apply: F V* AL→V	(mark-as-special <sym> [<flag>]) **defconstant** **defvar**	(symbol-function <sym>) (symbol-value <sym>)	**apropos**
ОС	 12253	bug fixes and extensions	cerror errset	(clean-up) (top-level)	**assert** <test> break command loop catch throw (unwind-protect <expr> <cexpr>...)	(pp <object> ...) (pp-def <funct> (pp-file <file> [<stream>])
ИП :=	 23251	«Нет переменной» «Нет метки»	**ERROR**, cerror, errset break, continue	**gethash** delete-if, delete-if-not nsubst-if, nsubst-if-not	and, or when, unless case, typecase unwind-protect check-type	the format function
ООП	 13331	**message.**	Messages: self - object - constant, *obarray* - the object hash table.	**multiple** object arrays (name spaces) Типы значений	The Object method :show (typep <expr> <type>) (type-of <expr>)	(generic <expr>)

Разнообразие от 11 до 18 механизмов управления вычислениями сопровождается включением специальных видов данных. Введён динамический контроль типов значений, потоки, сообщения, обработка хэш-таблиц, аналог выводу данных в строки, возможность обзор полного состава системы (**apropos**) и другое.

Таблица 24

Понятийная матрица внешнего **МИРА (P)** языка Clisp

№	*столбца*	**V**	**E**	**M**	**C**	**S**
ПП		V: 21	F: V* → V	AL : Atom → V	(F → {NIL, T})*	A ↔ K
ФП		**Числа** character strings fixnums (integers) bignums	+ - * / arithmetic functions Strings, characters, numbers of	Псевдо-функции и свойства: subrs, fsubrs, put, get, remprop	command loop REPL= ((λ (Nil wrk) (eval wrk) Nil '(print (eval (print	**PRINT** **READ**

	 94222	(integers) ratios characters **flonums** (floating point numbers) **complex numbers**	any type,	closures evaluate	(read))))) **))**	
Мета	 23321	string stream суперком-пиляция	string functions stream functions **hash-tables**	Make-string-input-stream, with-output-to-string get-output-stream-list, get-output-stream-string make-string-output-stream)	(y-or-n-p [<fmt> [<arg>…]]) (yes-or-no-p [<fmt> [<arg>...]])	(pp-def <funct> [<stream>])
БД	 43342	**package string symbol object**	(use-package <pkgs> [<package>]) (in-package <package>) (do-all-symbols (<var> [<result>]) <expr>...)	(defpackage <package> [<option>…]) (delete-package <package>) (make-package <package> &key :nicknames :use)	(symbol-package <symbol>) (package-valid-p <package>) (find-symbol <string> [<package>]) (find-package <package>)	(package-use-list <package>) (list-all-packages)
ОС	 12311	**compilation options**	**structure functions** multiple value functions	file i/o functions compile **system functions**	inspect.lsp commands (all …)	input/output functions
ИП :=	 56535	object **array** hash table **sequence structure**	object functions array functions hash table functions aref hash-table-count maphash	(generic <expr>) make-array vector make-hash-table gethash, remhash	**typep** every, notevery, some, notany	generic print prin1, princ pprint terpri fresh-line
ООП		**system** пакеты	OBJECT FUNCTIONS: **send** send-super	(hash <expr> <n>)	inspect.lsp Structure elements can be: examined and	The file object the OPEN function a FILE-STREAM

	12123				changed.	object

Видно, что парадигма ФП на этом уровне обогащена большим комплектом типов данных, таких числа и структуры данных. Появляется компиляция и передача сообщений.

4.8. Результаты анализа по категориям семантических систем

Просматривая полученные шесть монопарадигмальных матриц (таблицы 14-19) можно достаточно быстро сделать определённые выводы относительно сходства и различия семантических систем, образующих определение языка программирования. Сравнение по строкам можно видеть в таблицах 21-24, по столбцам в таблицах 25-29.

Конечно, такие выводы можно получить и анализируя обычную документацию по языку, но это займёт много больше времени. Начнём с обзора типов данных, используемых при организации вычислений. Напоминаем, что программирование работает не со значениями, а исключительно с представлениями данных [11].

Концепция данных в языке Lisp начинается с противопоставления атомарных и составных конструкций — атомы и символьных выражений (S-выражения). Атомы — предельно абстрактное данное, оно не обязано иметь значение, может просто обозначать само себя или обладать разными свойствами. Среди атомов выделяется атом Nil, выполняющий роль ограничителя составных конструкций — это пустой список, а заодно и роль значения «ложь», что удобно для компактной записи ветвлений над списками. Для организации вычислений в понятие атома включаются числа, литеры (character), строки (**string) и ошибочные ситуации** (BUG, ERROR), о которых происходит сообщение (message.), выглядящее как строка. Для парадигмы ООП зарезервирован атом ***self.***

Для уникальной идентификации атомов в системе программирования обычно используется хэш-таблица (hash table), что в дальнейшем позволяет использовать такую структуру (structure) равноправно остальными структурами данными и допускает поддержку реализации векторов для передачи аргументов для машинного кода (Array), а для ООП классов. В этой таблице, хранятся данные об атомах, организованные в так называемые «списки свойств», устроенные с использованием атомов в роли индикаторов (indicator) свойств и флагов, используемых при организации разных режимов вычисления, например, для отладки флаг (Trace). Элементы таблицы имеют фиксированные адреса, что приводит к неявному

использованию понятия «поле» (Place). Атом, обладающий свойствами, называется символ (symbol).

Символьные выражения строятся как консолидация пар данных в бинарные узлы, что выглядит как бинарное дерево (**tree),** самый удобный случай с ограничением вправо атомом Nil называется «список», список из пар называется «ассоциативный список» **(**a-list). Он используется для связи имен с их значениями или эквивалентных данных при оптимизации. Обобщение строк и списков даёт понятие «последовательность» или «поток» (sequence).

Разные виды символьных выражений используются для представления функций (functions), их замыканий (Clozure), вычислимых форм выражений (Eval), «ленивых» или частичных вычислений, ветвлений (Cod) или условных выражения (If) для контроля хода вычислений, программ (Program) и пакетов,(package), результатов компиляции функций (Compile) и опций управления компиляцией (COMPILATION OPTIONS), поддержки связи с внешним миров с помощью файлов (Read, Print) и доступа к операционной системе (system). Таким образом на уровне данных используется более 30-ти понятий.

Таблица 25

Понятийная матрица **ТД** языка Clisp - различимые данные

(value, nil, атомы, числа, bug, character, string – строки, message, list списки, tree, a-list, sequence, symbol, place, indicator, флаг, hash table, structure, array, класс, self, trace, system, file файл, form формы, program, package, functions, compile компилятор, compilation options, closure, lazy evaluation отложенные действия, partial evaluation частичные вычисления)

	FP	Мета	БД	ОС	ИП	ООР
B	NIL и другие атомы	list абстракт-ная интер-претация	**Place - адрес indicator**	Атомы, списки	список	объект класс
X	Sexpr: (V . V) список tree, **a-list**	program преобра-зования	**symbol Строки**	формы **expr fexpr subr fsubr**	generalized variables = sym place	**Name spaces**
D	«Нет значения», «Нет функции» subrs (built-in functions) fsubrs	*NIL* string *STREAM*	NIL Expr, Subr **Trace** Value	**bug** fixes and extensions	«Нет переменной» «Нет метки»	**message.**

	(special forms) **user defined functions**					
P	Числа character strings fixnums (integers) bignums (integers) ratios characters flonums (floating point numbers) complex numbers	string stream **супер-ком-пиляция**	**package** string symbol object	**compilation options**	object array hash table sequence structure	system пакеты

Показательно, что декларативная парадигма на базовом уровне потребовала введения неявного типа данных «поле». Его нет на уровне семантики языка, но появляются функции, оперирующие такими данными. Вводится контроль ошибок, особенно для программируемых функций. Поддерживается компиляция и супер-компиляция.

Таблица 26

Понятийная матрица **ОПЕРАЦИЙ (E)** языка Clisp (**сигнатуры**)

	FP	Мета	БД	ОС	ИП	ООР
B	**eq:** V V→{NIL,~Nil}	(Fmacro A B …)	**(getf <place> <prop> [<dflt>])**	Function	**(prog** …), progv prog1, prog2, progn	**self**
X	**atom** (Fn a1 a2 ...) ((λ (x1 x2 ...) expr) a1 a2 …) apply evlis evcon **FunCall** append, last	subst sublis	**(get <sym> <prop> [<dflt>])**	evalhook applyhook	nth elt aref get getf delete, fill, replace nsubst, nsublis, nintersection, nunion, nset-difference, nset-exclusive-or	answer the **evaluator** binds the symbol 'self'
D	**null** (functionp <s>) ERROR Strings functions Predicate functions	make-string stringp *частичные вычисления,*	**Atom**: V→{NIL,T} apply: F V* AL→V	cerror errset	**ERROR**, cerror, errset break, continue	**Messages**: self - object - constant, *obarray* - the object hash table.

P	+ - * / arithmetic functions Strings, characters, numbers of any type,	string functions stream functions hash-tables	(use-package <pkgs> [<package>]) (in-package <package>) (do-all-symbols (<var> [<result>]) <expr>...)	**structure functions** multiple value functions	object functions array functions hash table functions aref hash-table-count maphash	object functions: send send-super

Функции getf и get оперируют типом данных «поле». Возникает вспомогательная функция FunCall для перехода от представления произвольных функций к их выполнению. Возникли функции для работы со структурами данных, векторами, хэш-таблицами, строками и потоками.

Таблица 27

Понятийная матрица **ПАМЯТИ (М)** языка Clisp **(неявные)**

	FP	Мета	БД	ОС	ИП	ООР
B	((λ (x1 x2 ...) expr) a1 a2 …) **GC**	**(defmacro** <sym> <fargs> <expr>...)	(remf <place> <prop>) (make-symbol <pname>)	FunCall **(funcall <fun> <arg>...)**	**(setq** ..), set	**symbol self**
X	**(acons** &optional &rest &key [&aux &allow-other-keys]] **замыкание Defun**	nsubst nsublis copy-tree	(putprop <sym> val> <prop>) (remprop <sym> <prop>)	(baktrace [<n>]) trace untrace (debug) (nodebug)	(defsetf), (setf),psetq , push, pop incf, decf (block..) (tagbody ..)	**defclass defmethod definst**
D	**let, Let*, flet, labels** Свободные переменные **(complement <fun>)** expr, fexpr subr, **fsubr**	**(macrolet** (<binding>...) <expr>...) **read-from-string** read-line	(mark-as-special <sym> [<flag>]) defconstant defvar	(clean-up) (top-level)	gethash delete-if, delete-if-not nsubst-if, nsubst-if-not	**multiple** object arrays **(name spaces)** Типы значений

P	Псевдо-функции и свойства: subrs, fsubrs, put, get, remprop closures evaluate	Make-string-input-stream, with-output-to-string get-output-stream-list, get-output-stream-string make-string-output-stream)	**(defpackage** <package> [<option> …]) (delete-package <package>) (make-package <package> &key :nicknames :use)	**file i/o** functions **compile system functions**	(generic <expr>) make-array vector make-hash-table gethash, remhash	(hash <expr> <n>)

Появилась функция для вызова сборки мусора, отдельные функции для ООП и компиляции, управления потоками ввода-вывода, создания отдельных элементов разных структур данных и объявления разных видов областей видимости.

Таблица 28

Понятийная матрица **УПРАВЛЕНИЯ (C)** языка Clisp (**расширения**)

	FP	Мета	БД	ОС	ИП	ООР
B	**(eval <expr>) (quote <expr>)**	complement	**(symbol-name** <sym>)	ecase ccase etypecase ctypecase **check-type**	**return,** return-from <name> Label (go <sym>)	**show**
X	**cond** (if <t> <e1> <e2>) tailp map **reduce**	nsubst-if nsubst-if-not	**(declare** [<declaration> ...])	**assert** <test> break command loop catch throw (unwind-protect <expr> <cexpr>...)	**loop, do , do*, dolis, dotimes**	Messages: defined for the super-class. If no method is found, an error occurs.
D	**lazy** break continue **Типы значений**	macroexpand macroexpand-1	(symbol-function <sym>) (symbol-value <sym>)	**inspect.lsp** commands (all …)	and, or when, unless case, typecase unwind-protect **check-type**	The Object method **:show** (typep <expr> <type>) (type-of <expr>)
	command loop	(y-or-n-p	**(symbol-**	ecase	**typep**	inspect.lsp

P	**REPL=** ((λ (Nil wrk) (eval wrk) Nil '(print (eval (print (read))))) **)**	[<fmt> [<arg>…]]) (yes-or-no-p [<fmt> [<arg>...]])	**package** <symbol>) (package-valid-p <package>) (find-symbol <string> [<package >]) (find-package <package>)	ccase etypecase ctypecase check-type	every, notevery, some, notany	Structure elements can be: **examined and changed**

Видно разнообразие ветвлений и разных средств наблюдения за ходом вычислений, динамический контроль типов значений.

Таблица 29

Понятийная матрица **СТРКТУР (** **S)** языка Clisp (**внешний мир)**

	FP	Мета	БД	ОС	ИП	ООР
B	**cons:** V1 V2 → (V1 . V2) car: (V1 . V2) →V1 cdr: (V1 . V2) →V2	backquote comma-at, comma	**property list.** (symbol-plist <sym>)	List	**процедура** *список рабочих перемен ных, список меток*	'Class' is the **class** of
X	pairlis (V . . .) ; list, list* assoc, rassoc **nth, nthcdr**	**copy-list** **copy-alist**	**gensym**	command loop **REPL**	**Rplaca, rplacd** **nconc,** nreconc, nbutlast, sort, map-into, mapcan, mapcon, nreverse,	**Области видимости** и управление с помощью сообщений
D	**Строки**	(princ-to-string <expr>)	**apropos**	(pp <object> ...) (pp-def <funct> (pp-file <file> [<stream>])	the format function	**superclass** send Class :new
P	**PRINT** **READ**	(pp-def <funct> [<stream>])	(package-use-list <package>)	input/output functions	generic print prin1, princ pprint	Привлечение хэш-таблиц

			(list-all-packages)	terpri fresh-line	

Здесь сосредоточено разнообразие средств ввода-вывода, области видимости и пространства имён, обзор состава системы, генерация уникальных атомов, супер-классы, копирование списков.

4.9. Визуальная карта языка Clisp

Таблица 30

Визуальная карта языка Clisp

	ФП	Мета	БД	ОС	ИПП	ООП
Ядро						
Расширение						
Границы						
Контекст						

Такие визуальные диаграммы как в таблице 30 позволяют мгновенно оценить разницу в специфике парадигм, поддержанных мультипарадигмальным языком Clisp, подтверждающую общее мнение о его возможностях программирования на протяжении полного жизненного цикла программ, начиная с **функционального** проектирования, переходящего в **рефлексивный** компьютерный эксперимент, затем **императивную** оптимизацию программы, создание **декларативной** онтологии по реализуемым символам, **мета**-конструирование обобщённых и настраиваемых макро-определений с переходом к практичному комплекту **классов объектов**. Таким образом получается нечто вроде визитной карточки ЯиСП Clisp, позволяющей быстро выполнять грубое сравнение с другими языками.

Заключение.

Изложенные представления можно рассматривать как конкретизацию понятийной сложности по Колмогорову [9]. Близкие работы начаты ещё в середине 1960-ых годов, когда возникла задача создания ЯП, поддерживающих процессы разработки СП при создании новых ЯиСП. В конце 1970-ых в рамках работ по проекту МАРС [10] был выполнен обзор динамики развития компьютерных конфигураций, что обосновывает выбор основных семантических систем уровня аппаратуры, допускающих эффективную реализацию. В те годы был выполнен ряд серьёзных попыток создания языков системного программирования, позволяющих конструировать системы программирования. Наиболее продвинутые из них, такие как Венская методика, не учитывали оценок трудоёмкости и понятийной сложности системного программирования, но они составили базу Си-ориентированным LEX-YACC, идеи которых унаследовали Clang-LLVM [19] и другие инструменты разработки новых ЯП.

Прогресс в области эксплуатационных характеристик оборудования даёт основания для пересмотра и развития подходов к обработке программ системами программирования. Средства и методы синтаксически ориентированного конструирования прошли серьёзные испытания предыдущей эволюцией языков и систем программирования в качестве инструментов снижения трудоёмкости и повышения продуктивности программирования с обеспечением надёжности разрабатываемых программ. Теперь стоит задача исследования их возможностей и резервов в новых эксплуатационных условиях на фоне непрерывного размывания знаний и смены поколений программистов. Прежде всего это видно по расширению

спектра парадигм программирования и росту актуальности ПВ [4, 10, 21]. Возможно решение таких проблем связано с развитием средств и методов представления синтаксиса ЯП и его обогащение методами ФП и другими новыми формами.

Интересно, что описание языка Lisp 1.5 занимало 106 страниц, в которых разместилось и описание реализации, и непростые примеры применения языка. Описание диалекта Scheme занимает 461 страницу[6], диалекта Clojure – 600-800 страниц, причём и то, и другое не предоставляют описания реализации. Приятно, что описание JavaScript уже не так объёмно – всего 314 страниц. Описание стандарта языка C++ превосходит 1200 страниц. Конечно, для первого ознакомления можно прочитать и столь заметный объём. В настоящее время число новых языков программирования стремительно возрастает [20, 21], и их описания редко укладываются в 500 страниц. Это требует специальной разработки методов лаконичного представления особенностей языков программирования для практичной оценки их возможностей и сравнения при выборе подходящего. Предложенную визуализацию при оценке сложности и продуктивности программирования на базе ЯиСП можно дополнить разделением требований к постановкам задач по сферам применения на академические и производственные, а также, по уровню изученности на чёткие, практично развиваемые, усложнённо трудоёмкие и новые.

6 https://www.scheme.com/tspl4/grammar.html#./grammar:s22

Список литературы

1) Болски М. И. Язык программирования Си. Справочник: Пер. с англ. – М.: Радио и связь, 1988. – 96 с.: ил. ISBN 5-256-00171-X
2) Вирт Н. От Модулы к Оберону. // Системная информатика. Вып 1. Проблемы современного программирования. − Новосибирск: Наука. Сиб. отд-ние, 1991. − с. 63-75.
3) 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.
4) Городняя Л.В. Парадигма программирования, Учебн. пос., 1-е изд. в библиотеку вуза. СПб. «Издательство ЛАНЬ» (Учебники для вузов. Специальная литература), 2019, 232 с.
5) Городняя Л.В. Русское программирование в условиях новой поляризации мира и искусственной изоляции России. http://rkka21.ru/interview-l.gorodnaya.htm
6) Городняя Л.В. Систематизация парадигм программирования по приоритетам принятия решений/ Электронные библиотеки. Том 23 № 4 (2020): Тематический выпуск «Научный сервис в сети Интернет». Часть 2. с. 666-696. https://elbib.ru/article/view/616/711
7) Захаров Л.А., Покровский С.Б., Степанов Г.Г., Тен С.В. Многоязыковая транслирующая система. − Новосибирск, 1987. − 151 с.
8) Касьянов В.Н., Евстигнеев В.А. Графы в программировании: обработка, визуализация и применение. - С-Пб.: ЕХП-Петербург, 2003, - 1104 с.
9) Колмогоров А.Н. Три подхода к определению понятия «количество информации». − Проблемы передачи информации, 1965. № 1 (1): с. 3–11.
10) Котов В.Е. МАРС: архитектуры и языки для реализации параллелизма. // Системная информатика. Вып 1. Проблемы современого программирования. − Новосибирск: Наука. Сиб. отд-ние, 1991. − с.174-194.
11) Лавров С.С. Методы задания семантики языков программирования // Программирование, 1978. N 6. С. 3-10.
12) Мальцев А.И. Алгоритмы и рекурсивные функции. − М.: Наука. 1965 г., 392 с.

13) Фуксман А.П. Технические аспекты создания программных систем. – М.: Статистика, 1979. – 180 с.
14) Хендерсон П. Функциональное программирование. – М.: Мир, 1983. – 349 с.
15) Baar T. Verification Support for a State-Transition-DSL Defined with Xtext. Perspectives of System Informatics - 10th International Andrei Ershov Informatics Conference, PSI 2015, in Memory of Helmut Veith, Kazan and Innopolis, Russia, August 24-27, 2015, Revised Selected Papers. Lecture Notes in Computer Science 9609, Springer 2016, p.50-60.
16) Lucas P., Lauer P., Stigleitner H. Method and Notation for the Formal Definition of Programming Languges. IBM Laboratory – Venna, TR 25.087, 1968.
17) McCarthy J. LISP 1.5 Programming Mannual. – The MIT Press., Cambridge, 1963. – 106p.
18) Peter Wegner. 1990. Concepts and paradigms of object-oriented programming. SIGPLAN OOPS Mess. 1, 1 (August 1990), p. 7-87. https://pdfs.semanticscholar. DOI: http://dx.doi.org/10.1145/ .
19) http://clang.llvm.org/get_involved.html материалы по Clang — LLVM.
20) http://progopedia.ru/ — Сайт с описаниями 171 языка и 31 парадигмы.
21) https://www.levenez.com/lang/ — Диаграмма, представляющая хронологию появления и наследования концепций многих ЯП
22) Graham P. ANSI Common Lisp. Prentice Hall, 1996. 432 p.
23) Городняя Л. В. Визуализация результатов анализа языков программирования для их поверхностного сравнения. Вестник НГУ. Серия: Информационные технологии. 2021 Т.19, №2. С. 29–52. DOI: 10.25205/1818-7900-2021-19-2-29-52
24) Городняя Л. Перспективно стратегические парадигмы программирования академика Андрея Петровича Ершова https://www.sorucom.org/articles/materialy-mezhdunarodnoy-konferentsii-sorucom-2020/4540/