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

Л.В. Городняя

ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Часть 1
Сравнение парадигм программирования

Препринт
172

Новосибирск 2014

Препринт посвящен актуальной проблеме системной информатики, заключающейся в исследовании и разработке методов анализа, сравнения и формального определения парадигм программирования. Актуальность направления обусловлена резким ростом числа компьютерных языков нового поколения, ориентированных на применение и развитие современных информационных технологий. Демонстрация методов на материале конкретных парадигм и языков программирования планируется в отдельных препринтах. Содержание препринта представляет интерес для системных программистов, студентов и аспирантов, специализирующихся в области системного и теоретического программирования, и для всех тех, кто интересуется проблемами современной информатики, программирования и информационных технологий.

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

L.V. Gorodnyaya

PROGRAMING PARADIGMS

Part 1
Comparison of Programming Paradigms

Preprint
172

Novosibirsk 2014

The work describes research and specification of basic paradigms of programming. The author analyzes and compares special features of programming languages of different levels, from Assembler to multi-paradigm programming languages. A functional models to comparative description of implementation semantics of basic paradigms is proposed. The author proposes a scheme of describing and defining paradigm features of a programming languages. The approach is illustrated with fragments of programming languages of different levels, which belong to machine-oriented, system, imperative, object-oriented, and productive programming.

ВВЕДЕНИЕ

Препринт знакомит с разнообразием проявления парадигм программирования (ПП) и подходами к их поддержке в языках и системах программирования (ЯСП). В центре внимания исторически значимые и концептуальные языки программирования, в которых видны ключевые идеи и практические следствия их реализации. Стили и языки программирования, характерные для рассмотренных парадигм, отражают эволюцию технологий программирования (ТП), используемых при решении задач системной и прикладной информатики от средств машинного программирования на стыке с аппаратурой до языков сверхвысокого уровня и систем высокопроизводительного программирования, включая средства поддержки жизненного цикла программ (ЖЦП). Сформулированы рекомендации по выбору ПП в зависимости от степени изученности задач и особенностей ЖЦП.

Исследования в области информатики в настоящее время претерпевают изменение системы понятий, используемой в практических информационных технологиях (ИТ). В этом процессе расширяется пространство решений сложных задач, модернизируются методы развития информационных систем (ИС) на основе компьютерных сетей и многопроцессорных комплексов. Альтернативные подходы к обработке информации, накопленные и сложившиеся при создании и применении языков и систем программирования, принято называть парадигмами программирования. В настоящее время практики по существу различают более двух десятков ПП. Многие языки программирования относят к пяти-восьми парадигмам. Изучение и чёткая классификация уже сложившихся и новых ПП призваны помочь обоснованному выбору и созданию компьютерных языков при формировании программных проектов и совершенствовании ИТ.

Парадигмами программирования в форме языков и систем программирования представлено знание о потенциале ИТ. В момент создания язык программирования (ЯП) отражает некий прогноз коллектива специалистов относительно области приложения ИТ. Практика разработки и применения систем программирования (СП) конкретизирует и уточняет такое знание в форме отлаженных программ с комплектами данных для них и прецедентами успешного их применения. Такой успех можно рассматривать как результат удачного выбора ПП, задающей концептуальную схему постановки проблем и методов их решения с удобным инструментом «грамотного» описания фактов, событий, явлений и процессов, выделения частных и

общих понятий на уровне лексикона программирования. Развитие ПП отражает практичность языковых понятий и реализационных структур, используемых при создании сложных программных систем. Интересно отметить, что рейтинг популярности языков программирования отличается от рейтинга ЯП, использованных в успешно завершенных проектах.

Именно парадигмам программирования Роберт Флойд посвятил свою Тьюринговскую лекцию, в которой обратил внимание на значимость этого понятия в контексте проблемы обучения программистов [23]. Авторитетный ученый, заложивший основы теорий анализа и верификации программ, автор ряда весьма эффективных методов обработки данных, счёл необходимым подчеркнуть влияние ПП на успех программистских проектов, на особенности изучения разных ПП и на то, как они должны быть поддержаны в языках программирования. Рассматривая пример структурного программирования в качестве методологически доминирующей парадигмы программирования, Р. Флойд отметил, во-первых, нацеленность этой парадигмы на нисходящее проектирование, пошаговое улучшение и сведение задачи к более простым подзадачам, во-вторых, переход от конкретных объектов и функций машинного уровня к более абстрактным объектам и функциям, позволяющим продумывать модули, выделяемые при нисходящем проектировании.

На своем программистском опыте Р. Флойд сделал интересное наблюдение – искусство программирования включает в себя расширение репертуара используемых парадигм. В своих рассуждениях Р. Флойд опирается на публикации Томаса Куна, дискуссии с известными в программировании учеными (Эдсгер Дейкстра, Никлаус Вирт, Дэвид Парнас, Дональд Кнут, Джон Кок, Марвин Минский) и результаты применения весьма разных и достаточно популярных языков программирования (Fortran, Lisp, APL, Algol, Planner, Cobol, Pascal). Его внимание привлекла задача подготовки рекурсивных сопрограмм, удобно формулируемых в терминах недетерминизма, неэффективность которого практично преодолевается макротехникой. Роберт Флойд высоко оценил созданные в MIT многочисленные примеры потенциала программирования на языке Lisp, показывающие путь от простейших списков до универсальных структур данных (СД), способных манипулировать программами как данными.

Роберт Флойд выразил уверенность, что можно наладить ясное обучение ряду таких семантических методов для всех уровней программного проекта, чтобы у обученных студентов хватало головы на решение полного спектра проблем, начинающихся от упрощённой учебной постановки задачи до непрерывно расширяющегося класса практических задач.

Настоящий препринт посвящен представлению семантических методов, используемых при описании наиболее известных ПП. Следует отметить, что за тридцать лет, прошедших со времени Тьюринговой лекции Роберта Флойда, число различных языков и систем программирования возросло с нескольких сотен до десятков тысяч. При этом число парадигм не столь велико. В разных источниках называют от двадцати до сорока парадигм, нередко включая в их перечень отдельные техники и методы.

В последнее десятилетие собран значительный материал по истории информатики с акцентом на историю идей и проектов в области исследовательского системного программирования. Сравнение исторически значимых ПП с современными предложениями в области языков программирования и ИТ разработки ИС помогает оценке реализованных в них решений, обоснованию проектов программных средств, созданию новых компьютерных языков. За полувековую историю развития программирования создано несколько десятков тысяч компьютерных языков, включая языки, разные по назначению: языки спецификаций, языки моделирования, языки разметки, языки представления знаний и т.д. Для сравнения столь разноплановых языков и систем необходимо выделить принципы, по которым это делать – и это непростая проблема. Изучение и чёткая систематизация уже сложившихся ПП призваны помогать обоснованному выбору подходов к обеспечению производительности, надежности и эффективности сложных ИС, конструируемых с использованием разносортных ИТ и сервисов, применяемых в разных условиях.

Определенные трудности в исследование ПП привносит неоднозначность классификации средств программирования. Многие языки содержат черты различных ПП, а принадлежность языка к популярной ПП может быть декларирована без достаточных оснований. Поэтому ПП сравниваются не только по СД и допустимым средствам их обработки, но, кроме того, по критериям и границам их успешного применения, по требованиям к степени изученности решаемых задач, а также по уровню абстрагирования используемых понятий. Парадигма низкоуровневого программирования может быть характеризована возможностью реализации эффективных решений ценой использования общего доступа к слабо защищенным СД. Языкам высокого уровня свойственно использование иерархии СД, компоненты которой защищены от бесконтрольного взаимодействия независимо создаваемых фрагментов программы. Средства сверхвысокого уровня (языки спецификаций, языки параллельного программирования, системы представления знаний и т.д.) нацелены на полноту пространства реализа-

ционных решений, факторизуемого по отдельным направлениям проблем, связанных с разработкой и применением долгоживущих программ [3].

Современное состояние практичных языков и систем программирования характеризуется развитием компонентных технологий, подобно объектно-ориентированному подходу обеспечивающих классификацию фрагментов на уровне понятий пользователя и их отображение на уровень целевых архитектур. При таком подходе не получают полного выражения системные решения промежуточного уровня, что является целью новых работ по семантике языков программирования, а также исследований аспектно-ориентированного программирования. Резкое возрастание мощности информационных сетей повышает роль точного знания различных характеристик информационных процессов при разработке информационных комплексов, актуальность оценки их потенциала.

В области парадигматизации языков программирования активно работают многие известные зарубежные ученые. Огромный вклад в эту область исследований внесли такие корифеи информатики, как Джон МакКарти (John McCarthy), Джон Бакус (John Backus), Петер Наур (Peter Naur), Джекоб Шварц (Jacob T. Schwartz), Эдсгер Дейкстра (Edsger Dijkstra), Никлаус Вирт (Niklaus Wirth), Роберт Флойд (Robert Floyd) и др [2, 5, 23, 24, 27, 28]. Среди отечественных ученых следует упомянуть Андрея Петровича Ершова и Святослава Сергеевича Лаврова [7,9].

Изложение начинается с общего представления о разнообразии задач программирования и структуре пространства решений, принимаемых при программировании. Затем следует краткий экскурс в методы определения языков программирования, включая вопросы построения систем программирования как основных средств поддержки ПП. Далее рассматриваются вопросы прагматической классификации ПП на основе анализа особенностей их операционной семантики, представленной в форме семантических систем, реализационной прагматики, дающей примеры типовых механизмов, и эксплуатационной прагматики, показывающей практику их применения. В заключении предпринята попытка резюмировать особенности развития парадигм программирования, описать динамику их кристаллизации и наметить технику парадигматической характеристики языков и систем программирования. Приведена сводка терминов, толкование которых может вызывать разночтения.

Собственно примеры сравнительного описания ПП будут представлены в отдельном препринте, посвященном основным парадигмам программирования, включая результаты аналитической характеристики парадигм программирования разного уровня, таких как машинно-ориентированное

программирование, кодирование программ, оперирование процессами, макро-обработка программ и данных, программирование на языках высокого уровня (функциональное, производственное, системное, логическое, объектно-ориентированное) и мульти-парадигматических языках. Парадигма параллельного программирования и язык начального обучения параллельному программирования будут представлены в двух дополнительных препринтах.

1. ПРОЯВЛЕНИЕ ПАРАДИГМ ПРОГРАММИРОВАНИЯ

В середине прошлого (XX) века термин «программирование» не подразумевал связи с компьютером. Можно было увидеть название книги «Программирование для ЭВМ». Теперь по умолчанию термин «программирование» означает организацию процессов на компьютерах и компьютерных сетях.

Программирование как наука существенно отличается от математики и физики с точки зрения оценки результатов исследования. Уровень результатов, полученных физиками и математиками, обычно оценивают специалисты близкой или более высокой квалификации. В оценке результатов программирования большую роль играет оценка пользователя, не претендующего на программистские познания. Поэтому, в отличие от обычных наук, специалисты в области программирования частично выполняют функцию переводчика своих профессиональных терминов в понятия пользователя.

Программирование обладает своим специфичным методом установления достоверности результатов – это компьютерный эксперимент. Если в математике достоверность сводится к доказательным построениям, понятным лишь специалистам, а в физике – к воспроизводимому лабораторному эксперименту, требующему специального, мало кому доступного оснащения, то компьютерный эксперимент может быть доступен широкой публике.

1.1. Многоликое программирование

Еще одна особенность программирования обусловлена его зависимостью от быстро развивающейся элементной и инструментальной базы. Конкретные знания модных новинок устаревают, изобретённые методы впитываются языками и системами программирования или библиотеками стандартных подпрограмм, поэтому для быстрого обновления знаний и навыков нужен классический фундамент, прямое назначение которого не

вполне очевидно пользователям и новичкам. По этой причине программистские знания – это сочетание классики и моды.

Программирование использует в качестве понятийной базы математический аппарат (теория множеств, теория чисел, алгебра, логика, теория алгоритмов и рекурсивных функций, теория графов и др.), но это не означает прямого заимствования методов исследования и непосредственного использования результатов. И методы, и результаты математики программирование адаптирует к компьютерным средствам и реальному времени.

Критерии качества программ весьма разнообразны. Их выбор и упорядочение по существу зависит от класса задач и условий применения программ:

- результативность;
- надежность;
- устойчивость;
- автоматизируемость;
- эффективное использование ресурсов (время, память, устройства, информация, люди);
- удобство разработки и применения;
- наглядность текста программы;
- наблюдаемость процесса работы программы;
- диагностика происходящего.

Упорядочение критериев нередко претерпевает изменения по мере развития области применения программы, роста квалификацией пользователей, модернизации оборудования, информационных технологий и программотехники. Вытекающее из этого непрерывное развитие пространства, в котором решается задача, вводит дополнительные требования к стилю программирования информационных систем:

- гибкость;
- модифицируемость;
- улучшаемость.

Программирование как наука, искусство и технология исследует и творчески развивает процессы создания и применения программ, определяет средства и методы конструирования программ, разнообразие которых складывается в практике и экспериментах и фиксируется в форме языков программирования (ЯП). Сложности классификации быстро расширяющегося множества ЯП приводят к выделению понятия «парадигмы программирования», число которых меняется не столь стремительно. Это ставит задачу оп-

ределения принадлежности ЯП конкретной ПП или поддержки ПП определённым ЯП. Для решения этой задачи ПП характеризуется взаимодействием основных семантических систем, таких как вычисление, обработка структур данных, хранение данных и управление обработкой данных. ЯП, поддерживающий некоторую ПП, в значительной мере наследует характеристику ПП при его реализации в системе программирования (СП), а именно на уровне абстрагирования операционной семантики от возможностей доступной аппаратуры. При таком подходе выделятся три категории парадигм:

- низкоуровневое программирование;
- программирование на языках высокого уровня;
- подготовка программ на базе языков сверхвысокого уровня.

Низкоуровневое программирование связано со структурами данных, обусловленными архитектурой и оборудованием. При хранении данных и программ используется глобальная память и автоматная модель управления обработкой данных.

Программирование на языках высокого уровня приспособлено к заданию структур данных, отражающих природу решаемых задач. Используется иерархия областей видимости структур данных и процедур их обработки, подчиненная структурно-логической модели управления, допускающей сходимость процесса отладки программ.

Подготовка программ на базе языков сверхвысокого уровня нацелено на представление регулярных, эффективно реализуемых структур данных, при обработке которых возможны преобразования представления данных и программ, использование подобий и доказательных построений, гарантирующих высокую производительность вычислений и надежность процесса разработки программ, включая подготовку программ для многопроцессорных конфигураций.

Есть основания полагать, что отмеченная Р. Флойдом потребность в расширении репертуара парадигм при программировании связана с динамикой представления знаний, сводимой к чередованию фаз восходящего и нисходящего проектирования решений развивающихся постановок задач.

Рассматривая программы и программные системы как формы представления знаний, трудно удержаться от попытки анализа динамики представления знаний на основе аналогии с развитием программ и программных систем. Движущими силами этого развития является возникающая в разных видах деятельности неудовлетворённость и вызванная ею потребность в уточнении представления знаний и установления новой информации, которая раньше не попадала в поле зрения, или не было готовности ее понять.

Динамика представления знаний сводится к переходу от одного представления к другому.

Успешность деятельности ограничена «пропускной способностью» поля зрения. Это ограничение систематически преодолевается посредством обобщения, приводящего к представлениям более высокого порядка – представлениям более мощным, более организованным, например, к процедурам, функциям, фреймам, шаблонам, макросам. Последовательность шагов обобщения можно называть индуктивным развитием представления знаний. В методике программирования индуктивное развитие соответствует восходящим методам, «снизу вверх». Как правило, индуктивное развитие имеет некоторые пределы. Такие пределы при возрастании меры информативности используемых средств рассматриваются Д. Скоттом. Интересен случай, когда пределом является теория, достаточная для порождения всей достоверной информации, установленной на данный момент времени. При разработке программ на конкретном ЯП роль такого предела играет система программирования.

В результате индуктивного развития представления знаний наблюдается тенденция к возрастанию доли средств декларативного характера (таких как описания, отношения, формирователи, типы данных, фреймы, семантические сети, иерархии понятий, аксиоматические системы) в сравнении с долей средств процедурного характера (таких как действия, операции, операторы, процедуры, интерпретаторы, задания). Эта тенденция обуславливает рост популярности дедуктивных методов и может рассматриваться как стимул к переходу от индуктивного развития к дедуктивному. Дедуктивный вывод осуществляет переход от потенциальных знаний к актуальным. Традиционно для этих целей в системах искусственного интеллекта используется метод резолюций, системы продукций и другие средства. Чередование стадий индуктивного и дедуктивного развития можно рассматривать как обоснование выбора метода программирования в зависимости от динамики развития знаний о решаемой задаче (зрелость, степень изученности).

Применение развиваемых таким образом представлений может потребовать возврата к менее структурированным средствам (например, для упрощения обратной связи с областью, породившей решаемые задачи или для более тонкой детализации реализационных решений). Такой переход является конкретизацией представления знаний. В методике программирования конкретизация соответствует нисходящим методам «сверху вниз». (Вопреки лингвистической ассоциации нет причин нисходящие методы считать

обратными восходящим. Обобщение психологически не симметрично конкретизации. Top_Down – Bottom_Up.)

Независимо осуществляемое развитие приводит к задаче установления эквивалентности между различными системами представления знаний. При решении этой задачи возникают предпосылки для выравнивания потенциала систем (вводятся недостающие понятия, выполняются аналогичные построения, реализуются подобные инструменты). Таким образом, выделено четыре типа переходов: индуктивное и дедуктивное развитие, конкретизация и выравнивание. Эта классификация сопоставима с классификацией трансформаций программ в теории смешанных вычислений, предложенной А. П. Ершовым [7].

По степени изученности существенно различаются следующие категории постановок задач, влияющие на стиль мышления и выбор методов решения задач:

- новые,
- исследовательские,
- практичные,
- эффективные.

Для новых постановок задач характерно отсутствие доступного прецедента решения задачи, новизна используемых средств или недостаток опыта исполнителей. Исследовательские постановки задач обычно усложнены требованиями уникальности и универсальности. Практичные постановки задач нацелены на актуальность и удобство применения. Эффективные постановки задач включают в себя исследование возможностей используемых средств.

Другое направление различий обусловлено разнообразием требований к результату программирования – мере организованности созданной программы и рангу работоспособности реализованных решений:

1. Макетный образец – концептуальный минимум.
2. Экспериментальный полигон – потенциальный максимум.
3. Практичная версия – производственный компромисс.
4. Эффективная реализация – предельный баланс.

Макетный образец работоспособен при предъявлении автором небольшого набора подходящих данных. Для экспериментального полигона требуется приспособленность к обработке почти любых данных, какие могут быть заданы в качестве входных. Практичная версия может быть ограничена обработкой данных, реально встречающихся в сфере её приложения.

Для эффективной реализации нужны специально подобранные данные, показывающие её превосходство над менее искусно выполненными решениями задачи.

Независимо от класса решаемых задач парадигмы программирования могут отличаться уровнем абстрагирования от возможностей аппаратуры, степенью изученности постановок задач, мерой организованности и рангом работоспособности программ решения задач. Дальнейшая классификация ПП проявляется в технологиях программирования, схемах жизненного цикла программ и лексиконе программирования.

1.2. Технологии программирования

Понятие «технология» подразумевает существование целевого производственного процесса, в рамках которого за определенное время на ограниченных ресурсах при известной квалификации персонала по конкретным техническим процедурам гарантированно может быть получен запланированной результат.

Не считая учебных, любительских и экспериментальных программ, мир программирования породил широкий спектр весьма разных технологий для различных условий применения программ, таких как обслуживание непрерывно функционирующих комплексов, обеспечение информационной безопасности, оперативное прогнозирование природных катастроф и многое другое. Независимо от этого разнообразия существует чисто программистская линия технологического повышения производительности труда в программировании. Основные вехи этой линии связаны с кристаллизацией определённых ПП, обусловленных созданием конкретных ЯП, изобретением новых принципов реализацией СП, появлением новых технологий программирования (ТП).

1. Создание языка Фортран сопровождалось появлением технологии раздельной компиляции.
2. Язык Лисп дал жизнь технологии символьных вычислений и функционального программирования.
3. Разработка языка Си как средства реализации операционных систем привела к технологии машинно-зависимого переноса программ.
4. Практика применения учебного языка Паскаль позволила сформулировать принципы структурного программирования, выполняющего роль методологически доминирующей парадигмы учебного программирования.

5. Язык Prolog связан с парадигмой логического программирования и реализацией средств логического вывода на базе частичного определения решений, достаточных для практики.
6. Идея организации процессов в языке Simula 67 и реализация работы с объектами в языке SmallTalk 80 дали импульс ООП с технологией декомпозиции программ по иерархии классов, допускающей расширение, не искажающее ранее отлаженные определения.
7. Разработка средств авторского форматирования текстов для публикаций TeX/LaTeX породила попытку технологии грамотного программирования (успешную, но не принятую программистами) и публикационно-ориентированную ветвь функционального программирования на языке Haskell.

За исключением языка Lisp, в большинстве этих подходов программа рассматривается как статический объект, тогда как в реальности она развивается и может частично видоизменяться в процессе разработки.

Стоит вспомнить ещё многоязыковую транслирующую систему БЕТА, в которой были сформулированы идеи компиляции с внутренним языков, на уровне которого выполнялись оптимизирующие преобразования программ [8].

Ростовская школа программирования предложила весьма перспективную технологию вертикального слоения – сосредоточенного представления рассредоточенных действий [20].

Московский ВЦ АН вёл эксперименты по технологии программирования в терминах «точек роста» [9].

Особо стоит отметить UML как методику представления полного комплекта разноплановых точек зрения на разрабатываемую программу, что соответствует технологии совмещения разных ПП, взаимодополняющих друг друга, обеспечивающих наиболее удобное рассмотрение отдельных аспектов решения задачи в целом.

Ряд таких идей теперь успешно развивается в рамках .Net-технологии [19].

Существуют специальные технологии для особо сложных условий выполнения разработки, связанных с повышенными требованиями к защите информации (бортовые системы и станции слежения) или к дозированию улучшений для программ на непрерывно работающих станциях, где для редактирования работающих программ выделяются отдельные такты, за которые можно вносить лишь изменения, не нарушающие работоспособность [14].

Ждут своего часа резервы верификации, обретающие практичность на современной технике [16].

Дж. Вейнберг в своей фундаментальной книге «Психология программирования» отмечал принципиальное различие между профессиональным и любительским программированием, заметное лишь профессионалам, т.к. оно проявляется не в процессе программирования и не в качестве написанных программ, а в отношении к завершению работы. Любитель может прекратить программировать, лишь стоит исчерпаться его интересу к решению задачи. Профессионал после условно полной отладки программы несет ответственность за результаты её эксплуатации [29].

А. П. Ершов подчеркивал разницу в программировании «для себя» и «на заказ», в котором, кроме собственно программы, выполняется разработка эскизных прототипов и выполняется предварительная оценка осуществимости разработки.

Типичные просчёты при не вполне профессиональной разработке:

- неадекватность функционирования;
- недочеты во взаимодействии с оборудованием;
- неожиданные отказы в обслуживании;
- замедленное или ускоренное реагирование в сравнении с ожидаемым временем паузы;
- неполнота информации при визуализации;
- утрата актуальности данных;
- нарушение достоверности документации.

Трудоёмкость преодоления таких просчётов подвигла в 1970-е годы Ф. Брукса на замечательное, почти художественное, описание радостей и горестей программирования. Было много полемики «за и против» восходящих и нисходящих подходов к программированию. Надежды на то, что все проблемы отпадут по мере технического прогресса, оказались иллюзорными. В начале 2000-х годов, при повторном издании своей книги, Ф. Брукс констатировал, что больные проблемы технологии программирования так и не нашли своего решения.

Э. Дейкстра много и убеждённо пропагандировал идею неустранимой сложности программирования [5].

Тем не менее, в конце 1990-х годов стали набирать признание две перспективные технологии, представляющие собой разумный компромисс для разработки программ, обладающих новизной или исследовательским компонентом – экстремальное программирование (XP) и функционально-ориентированное проектирование (FDD) [18].

Технология экстремального программирования сложилась в сфере производства игровых программ. Она решает ряд больных проблем, отмеченных Бруксом, например

- приоритет подготовке тестов как самой удобной документации;
- парный эффект освобождает от «тушения огня керосином»;
- рефакторинг смягчает проблему второй системы, т.к. ничто не делается раз и навсегда;
- коллективное владение кодом исключает эффекты искажения оценки достигнутой работоспособности, т.к. идёт непрерывная интеграция обновляемых версий.

В целом, экстремальное программирование показывает обнадёживающие результаты на постановках задач с частыми обновлениями и соответствует восходящей методике программирования.

Технология функционально-ориентированного проектирования соответствует нисходящей методики программирования и учитывает ряд человеческих факторов, что позволяет процессу разработки достичь устойчивости. В центре внимания находится функциональная декомпозиция постановки задачи на части, осуществимость которых не вызывает сомнения, и доведение их до очевидной готовности и предельной универсальности. Такая технология пригодна для решения сложных задач с заметным исследовательским компонентом [18].

Следует отметить, что многие авторы современных технологий выражают претензии к авторитетным рекомендациям по стилю и технике программирования, якобы сделавшим практику программирования беспомощной. Здесь желательно помнить о молодости программирования и высоком темпе развития информационных технологий, осознание возможностей которых просто не успевает созревать.

Технология экстремального программирования и парадигма логического программирования по степени изученности задач соответствуют рангу «концептуальный минимум», допускают заметную недоопределенность или избыточную новизну, требующих пошагового освоения неизвестного заранее фактического материала. Технология экстремального программирования сопряжена с интенсивной разработкой практичных версий ранга «производственный компромисс».

Технология функционально-ориентированного проектирования и парадигма функционального программирования находятся в явном родстве с задачами ранга «потенциальный максимум», нацеленными на экспериментальное исследование полного пространства возможностей.

Технологии UML и парадигма ООП нацелены на быструю разработку программ для класса задач ранга «производственный компромисс», обеспечивающий достаточное удовлетворение реальных требований без невостребованных реализационных сложностей.

Технологии .Net и парадигма параллельного программирования заняты поиском средств и методов для решения задач, достигающих предельного баланса технических возможностей доступного оборудования. Это требует точного знания особенностей функционирования аппаратуры, алгоритмов конструирования эффективного кода программ, системных подходов к анализу, оптимизации и структурному представлению программ, стереотипов организации пользовательских интерфейсов и методов надежной обработки данных в разнородной памяти на базе многопроцессорных конфигураций [19].

Трудоёмкость параллельного программирования порождает необходимость разработки методов верифицирующей компиляции и оптимизации программных компонентов и средств масштабируемой макрогенерации кода и автоматизируемых трансформаций программ с удостоверением сохранения свойств при их комплексации из ранее отлаженных компонентов [16].

Программа, не устаревая физически, подвержена сложным эффектам, связанным с обнаружением и исправлением ошибок и моральным устареванием не только реализованных решений, но и исходной постановки задачи.

Любое производство содержит процессы поиска и устранения недочётов. В программировании это отладка и тестирование [21].

Независимо от ПП и ТП в процессе разработки программ примерно 45% трудозатрат падает на долю автономного и комплексного тестирования и отладки программ. Столь высокая нагрузка на тестирование в практическом программировании требует ясности в понимании его целей, заметно отличающихся от естественно лингвистических представлений:

Неверно, что тестирование демонстрирует отсутствие ошибок в программе! Тестирование – это процесс, показывающий наличие ошибки в программе.

Неверно, что задача тестирования – доказать корректность программы! Теоретически показано, что конечный набор данных принципиально недостаточен для полной проверки сколько-нибудь нетривиальной программы.

Неверно, что тестирование показывает соответствие программы ее назначению! Назначение программы обычно соприкасается с человеческими

факторами, не поддающимися ни формализации, ни проверке тестами. Отдельные несоответствия ожиданиям показать можно.

Тестирование – это организация такого применения программы, в котором обнаруживается наличие дефекта – ошибки или несоответствия ожиданиям конкретных групп пользователей.

Такое определение влечёт психологические трудности, противоречащие конструктивной природе программирования, нацеленного на регулярное создание «правильных» текстов. При интуитивном выборе тестов автором программы он подсознательно выбирает тесты и сценарии применения программы, на которых программа показывает себя вполне благополучно. Именно эта причина приводит к необходимости внешнего тестирования до признания программы готовым продуктом.

Тестирование начинается с гипотезы, что в программе скорее всего имеются незамеченные при разработке дефекты или недочёты, и необходимо подобрать тестовые данные, при которых применение программы некоторые из них обнаружит, сделает очевидными. При этом различаются собственно тесты – данные, которые программа должна бы обрабатывать, но, возможно, не сделает этого, и так называемые «нетесты» – данные, которые программа не должна бы обрабатывать, но, возможно, допускает их. Таким образом, постановка задачи для тестирования расширяется. Кроме определения того, что должна делать программа, следует определить, что она делать не должна.

С математической точки зрения это означает, что для определения множества недостаточно задать его элементы, надо ещё определить и всё, что множеству принадлежать не может.

Различают два подхода к тестированию – «черный ящик» и «белый ящик». Разница между ними заключается в отношении к информации об устройстве тестируемой программы.

При выборе тестового материала для чёрного ящика не используется информация о структуре программы, средствах её подготовки и отладки. Для этого есть основания – такая информация может вводить в заблуждение, или её надо дополнительно проверять. Белый ящик предполагает использование сведений о логике функционирования программы, структуре её исходного текста и методах конструирования объектного кода. Это помогает выбору результативного комплекта тестовых данных и оценке трудоёмкости предстоящих работ по тестированию. Таким образом, на практике эти подходы дополняют друг друга.

Обычно подготовка тестовых данных подчинена ряду принципов и гипотез, нацеленных на экономию труда и надёжность результатов тестирования:

1. Для выбираемых входных данных сразу подбираются ожидаемые выходные данные.
2. Тестирование проводят люди, непричастные к разработке программы.
3. Все результаты тестирования тщательно изучаются.
4. Все подготовленные тесты хранятся вечно.
5. Тест, не обнаруживающий ошибку, не имеет смысла как тест, но может быть полезен как демонстрационный материал.
6. Вероятность обнаружения новых ошибок сравнима с числом обнаруженных ошибок.
7. Тестирование – это творческий процесс, т.к. возможности допустить ошибку в программе разнообразнее, чем её правильные построения.
8. Удачный тест выявляет новую, незамеченную ранее ошибку.

Методы черного ящика:

- Эквивалентное разбиение данных на множества.
- Анализ граничных значений.
- Восстановление функциональных диаграмм программы.
- Изобретение гипотез об ошибках.

Методы белого ящика (учёт критериев):

- Поток управления.
- Поток данных.
- Границы применимости библиотечных модулей.
- Техника реализации конструкций используемого языка программирования.

При автоматизации тестирования решают следующие проблемы:

- Создание и накопление хорошего набора тестов.
- Оценка набора на полноту по ряду критериев.
- Исполнение программ на тестах, оценка результатов и их хранение.
- Символьное исполнение программ.

- Исполнение программ в альтернативных условиях, отличных от условий разработки.

Тонкости выбора тестового материала детально описаны в книге С.К. Черноножкина [21].

Процесс тестирования тесно связан с методами локализации ошибок и принципами их устранения. Это именно независимые процессы, требующие весьма различных дарований от исполнителя, похожих на талант детектива и пластического хирурга. Проблема устранения ошибок осложнена тем, что даже авторская правка программы – взрывоопасный процесс: могут возникнуть так называемые наведенные ошибки, что приводит к возрастанию объема работ по отладке.

На практике придерживаются ряда эвристических принципов исправления ошибок, относящихся к своего рода логике здравого смысла:

- Если обнаружена ошибка, значит, она не последняя.
- Там, где что-то пришлось исправить, вероятна наведённая ошибка.
- Чем меньше находится новых ошибок, тем труднее их найти и исправить.
- Прежде чем исправлять ошибку в работающей программе, надо взвесить цену исправления.

Существует методика контроля готовности программы путём посадки в неё ошибок, процент устранения которых рассматривают как показатель отлаженности программы.

Различается отношение к источникам ошибок и мерам их профилактики в разных ПП. Если императивно-процедурное программирование (ИПП) апеллирует к спецификации типов данных (ТД), обеспечивающих возможность статического контроля при компиляции программ, то функциональное программирование (ФП) предпочитает полный динамический контроль любых условий, гарантирующих корректность вычислений.

С математической точки зрения процесс тестирования можно представить как обнаружение точек графика реализуемой функции, через которые эта функция не должна бы проходить. Локализация ошибки – это установление нужной точки. Исправление ошибки – это преобразование определения функции к виду, дающему её уточнённый график.

В целом процесс разработки программ можно представить как последовательность шагов по уточнению постановки задачи, методов её решения, текста программы решения и набора данных, представляющих тесты и «нетесты». Если последовательность достигает состояния, в котором уточ-

няемые сущности соответствуют друг другу, то утверждается, что завершена отладка программы.

Подходы к отладке не менее требовательны к творчеству, чем методы тестирования, но объём необходимых затрат на отладку может быть сокращён выбором стиля конструирования программ из проверенных шаблонов, рамками надёжных стандартов, использованием удобно реализуемых моделей. Здесь многое даёт парадигма функционального программирования.

Сложность процесса отладки программы можно оценивать по мощности множества данных, на которых необходимо выполнить прогон программы. Так, например, множество данных для отладочного прогона программы, содержащей цикл, должно содержать не менее трёх наборов:

- без захода в тело цикла;
- с однократным заходом в тело цикла;
- с многократным прохождением тела цикла.

Крупные производители программного обеспечения при подготовке программного продукта используют специально организованные многопроцессорные комплексы, поддерживающие испытания программ перед формированием комплекта поставки.

Существуют технологии программирования в стиле непрерывной разработки интегрированной версии программы, согласно которым осуществляется прогон программы на всех тестовых наборах при каждом изменении программы (XP, FDD).

Достаточно очевидные трудности в развитии средств тестирования и отладки программ имеют и чисто исторические причины в области создания языков и систем программирования. На уровне аппаратуры, системы команд и языков управления заданиями в операционных системах, а также во многих языках функционального программирования, имеются средства представления программ отладки и тестирования. При переходе от языков низкого уровня к наиболее популярным императивным языкам высокого уровня такие средства не нашли места в системе понятий входных языков – получилась неполная парадигма. Особенно явно это сказалось на решениях относительно работы с внешней памятью. Если Фортран поддерживает обмен на уровне любых допустимых массивов данных и механизм раздельной компиляции, Лисп обеспечивает, кроме того, обмен на уровне программных компонент и обстановок исполнения программы, то, начиная с Алгола и Паскаля, обмен данными практически сводится к уровню скаляров. Не исключено, что эта проблема найдёт практичное решение по мере

развития новых мульти-парадигматических языков программирования, таких как F# и C#.

Таким образом, существует ряд ПП, появление которых обусловлено развитием ТП, представленных в форме языков и систем программирования, поддерживающих решение задач определённой степени изученности, меры организованности, уровня абстрагирования и ранга работоспособности программ при слабом учёте проблем тестирования и отладки, а также поддержки процесса разработки (полного жизненного цикла) программ.

1.3. Жизненный цикл программ

Важнейшая задача технологии программирования – обеспечение сходимости процесса отладки программы, т.е. достижение взаимного соответствия постановки задачи, методов её решения, текста программы решения и набора данных, представляющих тесты и «нетесты». Итоговая постановка задачи, решение которой реализовано в программе, может оказаться как обобщением, так и сужением исходной задачи, что можно назвать более общим термином – «уточнение». Обнаружение закономерностей в процессе разработки программ привело к понятию «жизненный цикл программы» (ЖЦП) [11].

Изменение ПП в процессе создания программ связано со структурой расширяемого пространства, в котором принимаются решения при практическом программировании.

При решении самой простой задачи можно выделить такие этапы, как подготовка к решению, собственно реализация решений и оценка полученных результатов. В прикладном программировании, использующем результаты докомпьютерного программирования, изначально выделяли этапы разработки и эксплуатации программы, совместно разделенные на фазы:

Разработка:

1) определение требований;
2) спецификация требований;
3) проектирование;
4) реализация программы (программирование и отладка).

Эксплуатация:

5) тестирование;
6) сопровождение.

Подразумевалось, что фазы упорядочены по времени, и добротное достижение цели каждой фазы гарантирует получение конечного результата в виде программы решения поставленной задачи. Некоторое время можно было видеть подтверждения этой гипотезы – на классе задач вычислительного характера, алгоритмы для которых были разработаны и детально исследованы задолго до появления компьютеров. На этом классе задач исторически сложилось определенное разделение труда на разработчиков алгоритмов и операторов счетных машин, выполняющих собственно вычисления на счетных машинах. Появление компьютеров лишь заменило операторов на программистов, реализующих алгоритм.

Так сложилась парадигма прикладного программирования. Класс заранее достаточно изученных для программирования задач быстро исчерпался, и пришли первые разочарования, потребовавшие неоднократного пересмотра принципов организации труда в программировании. Реальность заставила мириться с итеративностью фаз, любая из которых могла потребовать повторного прохождения, что получило название «принцип водопада». Кроме того, со временем обратили внимание на этап подготовки постановки задачи, опережающий этап разработки программ для решения новых задач. Много позднее возник этап «идентификации потребностей» – принятия решения о том, нужно ли вообще разрабатывать программу.

Подготовка:

0) постановка задачи.

Похожая схема работает при учебном программировании:

1) предложено условие задачи;
2) известны требования к оформлению программ;
3) задан изучаемый алгоритм ее решения;
4) требуется реализовать алгоритм в виде программы;
5) выполнить отладку программы;
6) продемонстрировать готовность программы.

В учебной практике первые три фазы отточены на предыдущих поколениях студентов, основные решения представлены в учебниках, и методистами выбраны критерии оценки полученных программ, гарантирующие успех обучения. Итеративность последних трёх фаз обычно ограничивается диагностикой компилятора и замечаниями/придирками преподавателя.

В производственном программировании сложилось понятие жизненного цикла программы (ЖЦП), структура которого требует не только четкого

ответа на вопросы «Что? Как? Кто?», но и уяснения приоритетов между ними.

Что? Что должно получиться в результате решения задачи? Каковы исходные данные, обработку которых будет выполнять программа? В каком виде будет представлен результат обработки данных?

Как? Как будет устроена программа обработки? Каков алгоритм обработки данных? Из каких действий выстраивается функционирование программы?

Кто? Кто сумеет выполнить разработку? Какой должна быть его квалификация? Сможет ли он довести программирование до нужного уровня работоспособности программы на имеющемся оборудовании в заданный срок?

При разнообразии ответов на эти и многие другие сопутствующие вопросы возникли аргументы за выделение в ЖЦП возможных дополнительных фаз, существенных для зрелого долгоживущего производства:

Факультативно:

7) определение очевидно реализуемых компонент;
8) авторское и внешнее тестирование;
9) сдача-приемка программы;
10) опытная эксплуатация;
11) передача на внедрение;
12) модернизация;
13) эволюция требований к программе.

Потребность в разнообразии парадигм часто маскируется распределением работ по разным ролям участников, что приводит к узкой специализации исполнителей.

Приходится учитывать, что на этапе подготовки задачи к программированию многие решения принимаются людьми, далекими от практического программирования, но они располагают знаниями относительно реальных требований к решению задачи. При этом именно профессиональный программист отвечает за обоснованный выбор технологии программирования, используемый инструментарий и оптимизацию необходимых ресурсов. Путь к решениям такого рода даёт выбор схемы предстоящего ЖЦП и ПП.

Появление фазы «эволюция», безусловно, констатирует, что изготовление сколько-нибудь нетривиальной программы – процесс итерационный. Он подразумевает переход к идентификации обновленных потребностей. Независимо от этого возникают повторы отдельных этапов, без которых

завершение разработки не может состояться по непредвиденным ранее причинам. Всё это взламывает прогнозы по трудозатратам и оценку времени готовности программы и приводит к модели непрерывного программирования.

Для любого итеративного процесса ключевой вопрос – гарантии выхода из цикла. Возникло понимание того, что, как правило, необходимость непредсказуемых повторов в ЖЦП связана с упущениями в принятии решений на ранних стадиях, что более тщательная и упорная работа, аккуратный контроль правильности решений оправдаются снижением общей трудоёмкости всего ЖЦП.

По мере расширения классов решаемых задач и областей приложения их решений сложилась каскадная схема ЖЦП, в которой задаются критерии завершения фаз, позволяющие управлять качеством разработки. Если классическая итерационно последовательная схема ЖЦП по принципу «водопад» допускает повторы и неформально нацеливает на минимизацию их числа в терминах достижения цели, после чего предполагается получение готовой программы, то каскадная схема задаёт четкие условия, гарантирующие исключение необходимости в повторении отдельных фаз ЖЦП.

Итак, каскадная схема представляет собой ряд циклически повторяемых фаз, завершение которых удостоверяется заранее выбранным условием:

Таблица 1

Условия завершения фазы ЖЦП

№ фазы	*Название фазы*	*Условие завершения фазы*
0	постановка задачи	достаточны средства управления качеством разрабатываемой программы;
1	определение требований	подготовлен документ, представляющий обзор желаемых результатов работы программы;
2	спецификация требований	выбраны средства для подтверждения результатов программы;
3	проектирование	определены методы верификации правильности проекта;
4	реализация программы	проведено тестирование компонент программы на полном комплекте выбранных средств;
5	тестирование	выполнена интеграция/сборка программы из ранее готовых и разработанных компонентов;

6	сопровождение	проведена аттестация программы, показавшая фактические границы её работоспособности;
7	развитие	проверено соответствие функционирования программы пользовательским требованиям к решению задачи.

Одновременно возникают идеи доказательного программирования.

Другой круг проблем связан с вопросами равномерной загрузки участников разработки, дозированием интенсивности работ, распределением ролей, наращиванием сценариев применения программы, контролем целостности специализированных версий, непредсказуемым появлением новых источников требований и отслеживанием достоверности документации, а также спецификой начальной и завершающей итерации.

Конкретно, начальная стадия отдает приоритет глубине анализа постановки задачи, скорости реализации минимального скелета программы (чтоб «задышала»), наглядности и убедительности при демонстрации обсуждаемых деталей реализации, вынесению в отдельные модули большинства подзадач, которые можно перенести на следующие итерации. Завершающая стадия требует службы распространения вносимых изменений во все версии и независимые копии программы.

Следует учитывать, что расширяющийся класс задач для программирования все активнее посягает на области приложения, в которых ещё не сложилась программируемая алгоритмика и методы решения задач находятся на уровне предварительного исследования. Попытка решить такие проблемы привела к табличной схеме Р. Гантера, позволяющей представлять совмещение фаз по времени выполнения разработки подобно диаграммам Г. Гантера. Для этого процесс разработки структурирован на фазы и технологические функции, образующие двумерное пространство с разметкой контрольных точек и возможных фазовых возвратов.

Фазы:

1) исследование;
2) анализ осуществимости;
3) конструирование возможных решений;
4) программирование;
5) оценка свойств программы;
6) использование программы.

Функции:

1) планирование;
2) разработка;
3) обслуживание;
4) документирование;
5) испытание;
6) поддержка;
7) сопровождение.

При создании такой таблицы рекомендовано учитывать следующие явления:

- программы подвержены итеративному развитию;
- существуют стабильные сценарии применения программы;
- неизбежно наращивание комплекса программируемых и выполняемых функций;
- ничто в разработке на делается однократно – раз и на всегда;
- не исключено размножение числа фаз.

В целом, таблица Гантера помогает выстроить сравнительно точный прогноз трудоёмкости программы, но при условии учёта следующих попутных работ, часто воспринимаемых как накладные расходы:

1. Определение плана итерирования фаз разработки.
2. Анализ полноты комплекта сценариев развития постановки задачи.
3. Моделирование пользовательского интерфейса.
4. Экспериментальная реализация новых компонентов.
5. Опережающий выбор тестовых наборов для новых компонентов.
6. Оперативная оценка успешности хода работ.
7. Пополнение базовой обстановки проекта.

Кроме того, начальная стадия любой фазы дополняется разбиением решаемых подзадач на первоочередные и перспективные и выбором критериев оценки результатов. Завершающая стадия некоторых фаз включает в себя создание комплекта поставки программного продукта, службу его сопровождения и процесс выделения повторно используемых компонентов. Анализ таблицы распараллеливания итерируемых фаз сопровождается разметкой рациональности/недопустимости совмещения технологических функций. Идея доказательности в программировании уступает критериям надежного программирования.

По-прежнему считается, что удачный выбор схемы ЖЦП достаточен для успеха разработки. Это означает, что существует принципиальная возможность создания верифицирующих средств, для которой частичные определения или спецификации могут выполнять роль моделей, допускающих верификацию по мере уточнения и конкретизации таких моделей на ранних стадиях разработки программы.

Проявление зависимости трудоёмкости программирования от степени изученности/новизны решаемых задач наряду с высокой технологичностью программирования, допускающей перевод любых изученных задач в ранг готовых библиотечных модулей, привело к понятию «полный жизненный цикл программы» (ПЖЦП). Отличие от ЖЦП заключается в представлении схем, отражающих стадии созревания постановки задачи и допускающих развитие пространства реализуемых решений по мере продвижения от первых демонстрационных версий, прототипов, макетных образцов программы к готовому программному продукту, пригодному к полноценному функционированию без участия программиста.

С физической точки зрения программа не имеет износа. Неожиданности в ее функционировании могут быть связаны как с незамеченными при разработке недочётами, так и с появлением несоответствия принятых решений новым возможностям оборудования, изменению квалификации пользователей, развитию методов программирования. Таким образом, программы подвержены моральному устареванию, влекущему продолжение разработки программ с ранее завершенным ЖЦП, что приводит к уточнению перечня этапов разработки программы.

Этапы:

0) появление задачи (обнаружение потребности);
1) определение требований (зачем это надо);
2) спецификация будущего продукта (что должен получить покупатель);
3) проект (что должен сделать программист);
4) реализация (как именно устроена программ);
5) тестирование (испытания сторонними силами);
6) сопровождение (служба ответов на вопросы и сбор рекламаций);
7) модернизация (потребность в развитии отдельных функций без радикальных изменений).

Возникает дополнительная детализация этапов и фаз разработки с целью оптимизации трудоёмкости неизбежно повторяемых видов работ и учёта разнообразия ролей участников проекта.

1. **Определение требований** (Решается вопрос, стоит ли затевать разработку или можно обойтись чем-то имеющимся.)
   - описание области применения программы (внешний контекст);
   - регламентация функций программы;
   - определение ограничений практичности программы.

2. **Спецификация будущего продукта**
   - лексикон проекта;
   - эскиз поведения разрабатываемой программы;
   - определение измеримых характеристик, пригодных для обоснования успеха;
   - описание условий эксплуатации, оборудования, квалификации персонала;
   - примеры обрабатываемых данных;
   - представление предполагаемых сценариев работы в документации.

3. **Проект**
   - системный анализ;
   - уровень организованности данных;
   - декомпозиция задачи до очевидно реализуемых компонент;
   - модули и компоненты, не требующие отладки;
   - используемые средства;
   - график реализации компонент и сборки работающей системы;
   - технология документирования всех процессов;
   - отслеживания готовности частичных результатов.

4. **Реализация**
   - распределение работ;
   - программирование/кодирование алгоритмов.
   - автономная и комплексная отладка;
   - подготовка к испытаниям;
   - опытная эксплуатация;
   - сдача-приёмка;
   - устранение замечаний.

5. **Тестирование**
   - внешние испытания;
   - экспертные оценки;
   - передача на эксплуатацию;
   - оформление продукта.

6. **Сопровождение**
   - создание базы данных «Ошибки и рекламации»;
   - поиск причин несоответствия программы ожиданиям пользователя (документация, программа, входные данные);
   - выбор позиций внесения исправлений (программа, инструкция по применению).

7. **Модернизация** (Заметить, что пришло время запустить новую версию)
   - улучшение характеристик программы;
   - адаптация к изменению условий применения;
   - учет развития потребностей.

Схемы ПЖЦП выделяют стадии жизни программы, учитывающие разные степени изученности решаемых задач, что позволяет объективно оценивать число необходимых повторов в производстве программ.

Разработка больших систем сопряжена с решением дополнительных проблем, таких как

- большое число итераций при реализации новых компонентов;
- повторное проектирование после реализации части системы;
- обнаружение неточности спецификаций;
- уточнение постановки задачи;
- изменение условий эксплуатации системы;
- расширение границ применения.

Минимизацию возвратов обеспечивают созданием специальных промышленных технологий программирования, включающих в себя, кроме ООП, методику наращивания надежности проектов, например, экстремальное программирование или функционально-ориентированное проектирование [18], сводящие программу к россыпи достаточно мелких, быстро отлаживаемых подзадач. При этом привлекаются два чисто технологических этапа, противоречиво наследующих преимущества каскадной схемы ЖЦП и управления качеством по таблицам Гантера:

- Форсированное принятие решений – как можно больше сразу.

- Управление качеством и профилактика необоснованных преждевременных решений

Каскадная схема

1. Критерии перехода на новую фазу.
2. Обзор достигнутых результатов.
3. Удостоверение соответствия спецификации.
4. Подтверждение требований заказчиком.
5. Верификация проекта.
6. Аттестация фактической разработки.
7. Экспертиза применения системы пользователем.

Управление качеством

1. Разметка контрольных точек переход к очередной фазе.
2. Организация временных функций для поддержки персонала.
3. Анализ осуществимости (исполнители и ресурсы).
4. Конструктивность решений.
5. Практичность, экономичность, коммерческий эффект.

Контрольные точки, используемые при управлении качеством, содержательно перекликаются с определениями критериев завершения фазы в каскадной схеме.

Потребность в оптимизации трудозатрат на первичное программирование приводит к идее модульного программирования, что влечёт за собой своего рода внутренний заказ на производство типовых модулей с дополнительными требованиями к рангу работоспособности результата, заметно повышающими трудоёмкость программирования:

0. Необходимость разработки модуля признана.
1. Ресурсы выделены.
2. Требования сформулированы.
3. Требования утверждены.
4. Спецификации составлены.
5. Спецификации утверждены.
6. Выполнена рабочая компоновка изделия, использующего модуль.
7. Изделие выдержало внешние испытания.
8. Изготовлен модуль в качестве программного продукта.
9. Продукт передан на распространение в составе общих библиотек.

Для больших систем собственно программирование сводится к реализации модулей, сборке программ из модулей и документированию полученных результатов. Следует отметить, что реализация модулей обладает существенно большей (в 3–9 раз) трудоёмкостью, чем разработка функционально эквивалентных автономных программ. Сборка программ из готовых модулей имеет дополнительную нагрузку на ответственное изучение их устройства и функционирования.

Документирование, при всей его важности, не относится к числу работ, успешно выполняемых программистом на уровне требований пользователя. Программист единственно достоверным документом воспринимает код программы. Всё остальное – резкое огрубление информации, точность которой достигнута большим трудом. Это является основанием для появления технических писателей из числа успешных пользователей. Успешный эксперимент по грамотному программированию признания практиков не получил [30].

Использование больших систем включает в себя ещё ряд работ на этапе сопровождения:

- Организация хранения эталонов всех версий программы, получивших внешнее распространение.
- Внедрение системы в области применения.
- Обучение пользователей рациональным методам работы с системой.
- Настройка системы на особенности рабочих мест пользователей.
- Сопровождение версий системы.
- Модернизация системы до исчерпания потребности в ее применении.
- Автоматизация внесения исправлений.

Коллективность разработки влечет заботу об устойчивости проекта к риску невыполнения работ отдельными исполнителями.

Упорядочение столь пёстрой картины ПЖЦП достигается в рамках так называемой спиральной схемы разработки. Ключевая идея сводится к исходному рассмотрению круга из небольшого (3–5) числа направлений в пространстве решений, например:

1. Планирование.
2. Анализ.
3. Конструирование.
4. Реализация.
5. Оценка.

Для начальной стадии проекта выстраивается схема в таком пространстве от центра круга в предположении, что после фазы 5) идет по циклу фаза 1) со сдвигом от центра на небольшой интервал. На любой из следующих фаз могут возникать дополнительные направления пространства, детализирующие эту фазу, или выделяться вспомогательные точки роста, в которых начинает раскручиваться своя спираль решения отдельной подзадачи.

Такая схема пригодна для демонстрации различий трудоёмкости стадий разработки программы в зависимости от степени изученности решаемой задачи и, следовательно, может давать основания для более реального прогноза времени разработки. Для большей наглядности следует такую спираль представить как растущую вертикально, спроецировать её на плоскость и ширину витка проассоциировать с трудоёмкостью.

Возникают парадигмы аспектно-ориентированного и компонентно-ориентированного программирования, отличающиеся подходами к декомпозиции сложных развивающихся программ.

Известно, что развитие постановок задач по степени изученности не обладает монотонностью изменения трудоёмкости реализации текущей версии программы. А именно, после небольших трудозатрат на разработку макетного образца в ранге концептуального минимума, где главная цель показать достоинства идеи, создание экспериментального полигона в ранге потенциального максимума требует принципиально значительных трудозатрат на полноту исследования и разработки средств и методов решения задачи и определения границ практичности её реализации. Интуитивная оценка трудоёмкости реализации практичной версии в ранге производственного компромисса существенно меньше, что обычно провоцирует экономию на предварительных исследованиях в надежде угадать границы практичности, но обычно приводит к необходимости многократного версифицирования решения задачи. Нередкое в пионерскую эпоху достижение предельного баланса при построении эффективных версий теперь является редкостью и сохраняет актуальность преимущественно в области применения суперкомпьютеров.

Общая, достаточно пёстрая, картина средств и методов поддержки ПЖЦП приводит к выводу, что ещё не сформировалась единая информационная модель процесса решения задач методом разработки программ. Такое положение дел приводит к избыточному разнообразию моделей, используемых на разных фазах ЖЦП, что заметно повышает трудоёмкость разработки и затрудняет взаимопонимание при передаче частичных результатов исполнителям очередной фазы. Попытки интеграции в общий корпус независимо созданного инструментария, типа CASE-технологий, не

привели к заметному снижению трудоёмкости разработки и оказались достаточно сложны для практики массового применения.

Следует обратить внимание, что за понятием ПЖЦП стоит понятие «жизненный цикл задачи, решаемой методом разработки программы». Можно сказать, что каждая стадия, этап, фаза заключается в том, что происходит уточнение информации о постановке решаемой задачи в конкретных условиях применения её решения. Проблема проверки того, в какой мере достигнутое уточнение продвигает к программе, представляющей решение задачи, пока не исследована. Существуют лишь средства статического анализа и верификации программ по отношению к заданной спецификации поведения программы.

Некоторую дань пониманию этой проблемы отчасти показывает тенденция создания мульти-парадигматических языков программирования. Не исключено, что веское слово в этой области принадлежит методике верификации на моделях [16].

Итеративные процессы, и ПЖЦП тут не исключение, могут быть конечными, стабильными или расходящимися. Фазы характеризуются различием в требованиях к квалификации и ролям участников. Каждая фаза порождает документ, представляющий результирующее уточнение. Теоретически возможно рассматривать такой документ как представление модели. В принципе ему можно дать вид упрощённой онтологии, приспособленной к лингвистическому анализу на ограниченном профессиональном жаргоне типа деловой прозы или лексикона программирования.

Таким образом, в практике программирования встречается целый ряд схем ЖЦП, отличающихся по стилю мышления при постановке задач и подходам к методам решения задачи. Кроме того, фазы и этапы разработки программ поддерживаются различным инструментарием, что также влияет на успех применения ПП. Понимание этого в UML привело к концепции «видения», дополняющей традиционные механизмы ЯСП. Постижение закономерностей ПЖЦП породило ряд ПП, нацеленных на снижение трудоёмкости разных стадий, этапов и фаз процесса разработки программ. Выбор ПП не только влияет на трудоёмкость, но и в заметной степени определяет работоспособность и живучесть разрабатываемой программы.

1.4. Лексикон программирования

Порождаемые на разных стадиях, фазах и этапах ЖЦП результаты и принимаемые решения лишь отчасти представляются средствами ЯСП, поддерживающей определённую ПП. Остальное описывается естествен-

ным языком или образными диаграммами и может быть оформлено как отдельный документ или размещено в тексте программы на правах комментария. Поэтому поддержка ПП средствами ЯСП в практике дополняется достаточно тонкой лингвистической и конструкторской работой, важность которой, увы, не нашла достойного отражения в курсах обучения программированию.

Тем не менее, многие преподаватели программирования считают важным при обучении программированию начинать с записи алгоритмов на родном языке и лишь после этого переходить сначала к диаграммам, а потом к ЯП.

А. П. Ершов неоднократно возвращался к теме создания лексикона программирования и отдельно – к феномену деловой прозы [7].

На вопрос о проверке способностей к программировании. Э. Дейкстра отвечал, что следует проверять умение ориентироваться в большом количестве текстов и уметь выражать свои мысли на родном языке [5].

Д. Кнут, автор эпохального многотомника «Искусство программирования», счёл нужным потратить несколько лет на создание средств обработки текста, позволяющих удовлетворить самые взыскательные требования к оформлению текста. Кроме того, им была предпринята, строго говоря, успешная разработка технологии грамотного программирования, не воспринятая практиками [30].

Лингвистические способности и способность к переводу на разные языки дали убедительные прецеденты успешного программирования, но они не повлекли массового подражания.

В реальности каждая ПП привносит определённые тонкости в понимание терминов, жаргона и фразеологии. Таким образом, парадигмы имеют представление в лексиконе программирования.

Следует отметить, что в современной практике программирования спецификация проекта часто представляется как словарь проекта. Важной частью комплектации программного продукта является справочная система, качество которой напрямую связано с владением речью.

Здесь же уместно отметить визуальные шаблоны и стандарты на оформление пользовательских интерфейсов.

Более важно то, что понимание большинства общих программистских понятий в разных ПП имеет некоторые отличия, которые не так легко отследить при представлении и обсуждении принятых решений. Поэтому каждой парадигме соответствует определённое подмножество лексикона программирования, вносящее ясность в трактовку используемых терминов, жаргона и фразеологизмов.

1.5. Развитие парадигм программирования

Знакомое из философии и лингвистики слово «парадигма» имеет в информатике и программировании узкопрофессиональный смысл. Парадигма программирования как исходная концептуальная схема постановки проблем и их решения является инструментом грамматического описания фактов, событий, явлений и процессов, возможно, не существующих одновременно, но интуитивно объединяемых в общее понятие.

Каждая парадигма программирования имеет свой круг приверженцев и класс успешно решаемых задач. В их сфере приняты разные приоритеты при оценке качества программирования, отличаются инструменты и методы обработки данных и, соответственно, – стиль мышления и изобразительные стереотипы. Нелинейность развития понятий, зависимость их обобщения от индивидуального опыта и склада ума, чувствительность к моде и внушению позволяют выбору парадигм в системе профессиональной подготовки информатиков влиять на их восприимчивость к новому.

Наиболее распространённая парадигма прикладного программирования на основе императивного управления и процедурно-операторного стиля построения программ получила популярность более пятидесяти лет назад в сфере узкопрофессиональной деятельности специалистов по организации вычислительных и информационных процессов. Последние десятилетия резко расширили географию информатики, распространив ее на сферу массового общения и досуга. Это меняет критерии оценки информационных систем и предпочтения в выборе средств и методов обработки информации.

Консервирующееся в наши дни доминирование одной архитектурной линии, стандартного интерфейса, типовой технологии программирования и т.д. чревато потерей маневренности при обновлении информационных технологий. Особенно уязвимы в этом отношении люди, привыкшие в учебе прочно усваивать все раз и навсегда. При изучении языков программирования подобные проблемы обходят путем одновременного преподавания различных языков программирования или предварительного изложения основы, задающей грамматическую структуру для обобщения понятий, изменяемость которых трудно улавливается на упрощенных учебных примерах. Именно такую основу дает изучение функционального программирования тем, что оно нацелено на изложение и анализ парадигм, сложившихся в практике программирования в разных областях деятельности с различным уровнем квалификации специалистов, что может быть полезно как концептуальная основа при изучении новых явлений в информатике.

Общие парадигмы программирования, сложившиеся в самом начале эры компьютерного программирования, – парадигмы прикладного, теоретического и функционального программирования в том числе, имеют наиболее устойчивый характер.

Прикладное программирование подчинено проблемной направленности, отражающей компьютеризацию информационных и вычислительных процессов численной обработки, исследованных задолго до появления ЭВМ. Именно здесь быстро проявился явный практический результат. Естественно, в таких областях программирование мало чем отличается от кодирования, для него, как правило, достаточно операторного стиля представления действий. В практике прикладного программирования принято доверять проверенным шаблонам и библиотекам процедур, избегать рискованных экспериментов. Ценится точность и устойчивость научных расчетов. Язык Фортран – ветеран прикладного программирования, постепенно стал несколько уступать в этой области Паскалю-Си, а на суперкомпьютерах – языкам параллельного программирования, таким как Sisal. Теперь к ним присоединяются нововведения в C# и Java.

Теоретическое программирование придерживается публикационной направленности, нацеленной на наглядность и сопоставимость результатов научных экспериментов в области программирования и информатики. Программирование пытается выразить свои формальные модели, показать их значимость и фундаментальность. Эти модели унаследовали основные черты родственных математических понятий и утвердились как алгоритмический подход в информатике. Стремление к доказательности построений и оценка их эффективности, правдоподобия, правильности, корректности и других формализуемых отношений на схемах и текстах программ послужили основой структурированного программирования и других методик достижения надежности процесса разработки программ, например, грамотное программирование. Стандартные подмножества Алгола и Паскаля, послужившие рабочим материалом для теории программирования, сменились более удобными для экспериментирования аппликативными языками, такими как ML, Miranda, Scheme, Haskell и другие.

Функциональное программирование сформировалось как дань математической направленности при исследовании и развитии искусственного интеллекта и освоении новых горизонтов в информатике. Абстрактный подход к представлению информации, лаконичный, универсальный стиль построения функций, ясность обстановки исполнения для разных катего-

рий функций, свобода рекурсивных построений, доверие интуиции математика и исследователя, уклонение от бремени преждевременного решения непринципиальных проблем распределения памяти, отказ от необоснованных ограничений на область действия определений – все это увязано Джоном Маккарти в идее языка Лисп. Продуманность и методическая обоснованность первых реализаций Лиспа позволила быстро накопить опыт решения новых задач, подготовить их для прикладного и теоретического программирования. В настоящее время существуют сотни функциональных языков программирования, ориентированных на разные классы задач и виды технических средств.

Основные парадигмы программирования сложились по мере возрастания сложности решаемых задач. Произошло расслоение средств и методов программирования в зависимости от глубины и общности проработки технических деталей организации процессов компьютерной обработки информации. Выделились разные стили программирования, наиболее зрелые из которых – машинно-ориентированное, системное, логическое, трансформационное, высокопроизводительное/параллельное и объектно-ориентированное программирование.

Машинно-ориентированное программирование характеризуется аппаратным подходом к организации работы компьютера, нацеленным на доступ к любым возможностям оборудования. В центре внимания – конфигурация оборудования, состояние памяти, команды, передачи управления, очередность событий, исключения и неожиданности, время реакции устройств и успешность реагирования. Ассемблер в качестве предпочтительного изобразительного средства на некоторое время уступил языкам Паскаль и Си даже в области микропрограммирования, но усовершенствование пользовательского интерфейса может восстановить его позиции.

Системное программирование долгое время развивалось под прессом сервисных и заказных работ. Свойственный таким работам производственный подход опирается на предпочтение воспроизводимых процессов и стабильных программ, разрабатываемых для многократного использования. Для таких программ оправданна компиляционная схема обработки, статический анализ свойств, автоматизированная оптимизация и контроль. В этой области доминирует императивно-процедурный стиль программирования, являющийся непосредственным обобщением операторного стиля прикладного программирования. Он допускает некоторую стандартизацию и модульное программирование, но обрастает довольно сложными построениями, спецификациями, методами тестирования, средствами инте-

грации программ и т.п. Жесткость требований к эффективности и надежности удовлетворяется разработкой профессионального инструментария, использующего сложные ассоциативно семантические эвристики наряду с методами синтаксически-управляемого конструирования и генерации программ. Бесспорный потенциал такого инструментария на практике ограничен трудоемкостью освоения – возникает квалификационный ценз.

Логическое программирование возникло как упрощение функционального программирования для математиков и лингвистов, решающих задачи символьной обработки. Особенно привлекательна возможность в качестве понятийной основы использовать недетерминизм, освобождающий от преждевременных упорядочений при программировании обработки формул. Продукционный стиль порождения процессов с возвратами обладает достаточной естественностью для лингвистического подхода к уточнению формализованных знаний экспертами, снижает стартовый барьер.

Трансформационное программирование методологически объединило технику оптимизации программ, макрогенерации и частичных вычислений. Центральное понятие в этой области – эквивалентность информации. Она проявляется в определении преобразований программ и процессов, в поиске критериев применимости преобразований, в выборе стратегии их использования. Смешанные вычисления, отложенные действия, «ленивое» программирование, задержанные процессы и т.п. используются как методы повышения эффективности информационной обработки при некоторых дополнительно выявляемых условиях.

Высокопроизводительное программирование нацелено на достижение предельно возможных характеристик при решении особо важных задач. Естественный резерв производительности компьютеров – параллельные процессы. Их организация требует детального учета временных отношений и неимперативного стиля управления действиями. Суперкомпьютеры, поддерживающие высокопроизводительные вычисления, потребовали особой техники системного программирования. Графово-сетевой подход к представлению систем и процессов для параллельных архитектур получил выражение в специализированных языках параллельного программирования и суперкомпиляторах, приспособленных для отображения абстрактной иерархии процессов уровня задач на конкретную пространственную структуру процессоров реального оборудования.

Объектно-ориентированное программирование провозгласило приоритет представления понятийной системы сферы применения разрабаты-

ваемой программы в форме расширяемой иерархии классов объектов, обработка которых формируется с помощью методов, подчинённых типизированной дисциплине доступа к данным.

Направление развития парадигмы программирования отражает изменение круга лиц, заинтересованных в развитии и применении информационных систем. Многие важные для практики программирования понятия, такие как события, исключения и ошибки, потенциал, иерархия и ортогональность построений, экстраполяция и точки роста программ, измерение качества и т.д. не достигли достаточного уровня абстрагирования и формализации. Это позволяет прогнозировать развитие парадигм программирования и формировать курсы для программистов на перспективу предстоящего появления парадигм верификационного, компонентного программирования. Если традиционные средства и методы выделения многократно используемых компонентов подчинялись критерию модульности, понимаемой как оптимальный выбор минимального сопряжения при максимальной функциональности, то современная элементная база допускает оперирование многоконтактными узлами, выполняющими простые операции.

Противопоставление разрабатываемых программ, обрабатываемых данных и управления заданиями уступает представлению об интерфейсах, приспособленных для участия в информационных потоках подобно навигации. Прежние критерии качества: скорость, экономия памяти и надежность обработки информации – все больше заслоняются игровой привлекательностью и широтой доступа к мировым информационным ресурсам. Замкнутые программные комплексы с известными гарантиями качества и надежности форсированно вытесняются открытыми информационными комплектами с непредсказуемым развитием состава, способов хранения и обработки информации.

Исследовательский подход с учебно-игровым стилем экспериментального, обучающего и любительского программирования может дать импульс поисковой изобретательности в совершенствовании технологии программирования, не справившейся с кризисными явлениями на прежней элементной базе.

Эволюционный подход с мобильным стилем уточнения программ достаточно явно просматривается в концепции объектно-ориентированного программирования, постепенно перерастающего в субъектно-ориентированное, аспектно-ориентированное и даже эго-ориентированное программирование. Повторное использование определений и наследование

свойств объектов могут удлинить жизненный цикл отлаживаемых информационных обстановок, повысить надежность их функционирования и простоту применения. Когнитивный подход с интероперабельным стилем визуально-интерфейсной разработки открытых систем и использование новых аудио-видео средств и нестандартных устройств открывают пути активизации восприятия сложной информации и упрощения ее адекватной обработки.

Адаптационный подход с эргономичным стилем индивидуализируемого конструирования персонифицированных информационных систем предоставляет информатикам возможность грамотного программирования, организации и обеспечения технологических процессов реального времени, чувствительных к человеческому фактору.

Эти симптомы обновления парадигмы программирования определяют направление изменений, происходящих в системе базовых понятий, в концепции информации и информатики. Тенденция использования интерпретаторов (точнее неполной компиляции) вместо компиляторов, анонсированная в концепции языка Java в сравнении с Си, и соблазн объектно-ориентированного программирования на фоне общепринятого императивно-процедурного стиля программирования можно рассматривать как неявное движение к функциональному стилю. Моделирующая сила функциональных формул достаточна для полноценного представления разных парадигм, что позволяет на их основе экстраполировать приобретение практических навыков организации информационных процессов на будущее

Важно отметить, что основные ПП по-разному проявляют себя на разных классах задач и выбор ПП влияет на трудоёмкость программирования, успешность технологии программирования и надёжность применения программ.

Информатика прошла путь от профессионального программирования высококвалифицированной элиты технических специалистов и научных работников до свободного времяпрепровождения активной части цивилизованного общества. Освоение информационных систем через понимание с целью компетентных действий и ответственного применения техники сменилось интуитивными навыками хаотичного воздействия на информационную среду со скромной надеждой на везение, без претензий на знание. Обслуживание центров коллективного пользования, профессиональная поддержка целостности информации и подготовки данных почти полностью отступили перед самообслуживанием персональных компьютеров, незави-

симым функционированием сетей и разнородных серверов со взаимодействием различных коммуникаций.

Парадигма программирования является инструментом формирования профессионального поведения, гарантирующего надёжность массово используемых ИТ.

1.6. Эксплуатационная прагматика

Можно констатировать, что строго определённая форма программы и её практически автоматное функционирование подчинены целому ряду трудно формализуемых аспектов, отражающих экспертную оценку целесообразности выбора постановки задачи, средств и методов её решения и успеха в программировании. В этом ряду следует ещё раз отметить некоторые элементы эксплуатационной прагматики:

Критерии качества программ – могут изменяться в процессе программирования, что возможно повлечёт существенный пересмотр многих ранее принятых решений.

Степень изученности решаемых задач – ведущий фактор прогноза трудоёмкости достижения работоспособной программы, пригодной для неавторского применения.

Развитие систем представления знаний представляет собой чередование фаз восходящего и нисходящего уточнения информации, что влияет на выбор технологии программирования.

Фразеология и лексикон программирования представляют собой основу профессиональной коммуникации, обеспечивающей рабочее взаимодействие в коллективном проекте.

Уровень абстрагирования текста программы от аппаратуры показывает трудоёмкость переноса программы на другие системы.

Технологии программирования обеспечивают гарантированное получение результата разработки при условии соответствия технологическим требованиям.

Тестирование и отладка программы требуют специального обустройства, средства которого за редким исключением не представлены в ЯП, но могут содержаться в СП.

Сходимость ЖЦП обеспечивается грамотным выбором соответствия степени изученности решаемой задачи и схемы ЖЦП, допускающей при подходящей технологии выполнение отладки программы, удовлетворяющей заданному критерию качества.

Ранг работоспособности реализованных решений зависит от полноты и универсальности программных компонент, созданных при разработке программы.

Выводы:

1. Выбор ПП влияет на успех ТП и трудоёмкость ПЖЦП, а также на повышение изученности решаемой задачи, включая методы её решения, на организованность, работоспособность и живучесть разрабатываемой программы.

2. ПП проявляется в удобстве уточнения постановки задачи, метода её решения, декомпозиции программы на фрагменты, модули, аспекты или компоненты и представления принятых в проекте решений в терминах лексикона программирования и ЯСП, поддерживающих данную ПП.

3. Классификация ЯСП по поддерживаемым ПП требует анализа их операционной семантики и реализационной прагматики.

2. ПОДДЕРЖКА ПАРАДИГМ ПРОГРАММИРОВАНИЯ

Задача этой главы – показать, каким образом определение ЯП и реализация СП поддерживают разные ПП. Рассмотрим средства и методы, используемые при определении языков программирования и их расширение при реализации систем программирования. Для примеров используется материал учебных концентров ЯП Lisp и Pascal.

Обычно определение ЯП задается раздельно на уровнях лексики, синтаксиса, семантики и прагматики. Реализация СП для определённой семантики ЯП может быть выполнена как интерпретатор или компилятор или и то и другое совместно.

2.1. Лексика

При определении лексики и синтаксиса формальных языков обычно используют формулы Бэкуса-Наура (БНФ) или синтаксические диаграммы[1].

Различия в поддержке парадигм программирования проявляются на всех уровнях реализации ЯСП, начиная с лексики. Так, например, в большинстве ЯП за редким исключением понятие «идентификатор» определяется формулой:

```
идентификатор = БУКВА | идентификатор {БУКВА | ЦИФРА}
```

[1] Можно ознакомиться в [2, 4] и других книгах по программированию.

При переходе к реализации СП часто допускают традиционные ограничения на число литер, участвующих в распознавании идентификатора, что приводит к неразличимости длинных идентификаторов, обладающих совпадающими началами. Имеются ЯП, требующие при реализации обеспечивать участие всех литер в распознавании идентификатора.

В большинстве ЯП целые числа без знака определяются формулой:

целое = ЦИФРА {ЦИФРА}

Как правило, при реализации СП практикуется традиционное ограничение на величину числа, оправдываемое предпочтением эффективных вычислений в пределах машинного слова или четырёх байтов. Но известны ЯП, свободные от такой традиции. СП для них способны обрабатывать целые числа произвольной длины, что повышает результативность программ и обеспечивает организацию счёта в произвольной точностью.

Таблица 2

Примеры лексем языка Pascal

	Примеры	*Пояснение*
1	123456789	Целые числа в диапазоне ±2**32
2	factorial	Идентификаторы, распознаваемые по определённому в реализации числу литер
3	if then else begin end while do var const procedure	Зарезервированные ключевые слова
4	. , ; :	Разделители
5	[ ] ( )	Ограничители
	:= * mod div & + - or = # < > <= >=	Операции

Первое ограничение нацелено на поддержку эффективных вычислений, второе – на скорость распознавания идентификаторов при компиляции; остальные ограничения призваны обеспечить удобочитаемость текста программы.

Таблица 3

Примеры лексем языка Lisp

	Примеры	*Пояснение*
1	12345678901234567890123 45	Целые числа без ограничения на число цифр
2	Атом	Атомы, распознаваемые по всем литерам с возможностью расширения алфавита.

3	NIL	Изображение пустого списка в виде атома.
4	пробел	Разделитель элементов списка
5	( )	Ограничители – начало и конец списка

Отказ от ограничения на длину чисел повышает результативность вычислений. Потери на скорости распознавания атомов оправдываются естественностью в выборе их внешнего вида. Отсутствие зарезервированных слов упрощает экспериментирование с семантикой программ. Пробел в качестве разделителя элементов списка эргономичнее других символов при наборе текста. Унификация скобок в представлении программы списками даёт возможность удобного преобразования её структуры при отладке.

Таким образом, некоторые различия в поддержке разных ПП закладываются уже при выборе лексики ЯП при его реализации в СП.

Встречаются вариации в определении лексем, так, например, понятие «лексема» может выделяться пробелами или другими разделителями или ограничителями (Bliss).

Тип значения идентификатора может определяться первой литерой, категория значения может быть выражена регистром, в алфавит могут быть включены дополнительные литеры «-» или «_», на вхождения которых могут быть особые ограничения (запрет быть первой или последней литерой и т.п.).

Списки, векторы, потоки, множества или кортежи значений или выражений могут быть заключены в разные скобки и разделены разными символами:

```
(a b c)
[a, b, c]
<a, b, c>
{a, b, c}
(a, b, c)
(a; b;c)
(a:b:c)
```

Элементы структур могут быть однородными или произвольными, их число может быть фиксировано или неограниченно, они могут быть помечены ключами и обладать кратностью.

```
{1, 2, 3}
{1, {2}, 3}
{<1 a>, <2 b> , <3 c>}
```

{date: MON, year: 2014}
{pencil: red * 2, list: A4 * 100}

Для представления рамочных конструкций – блоков, ветвлений, циклов, – как правило, используются резервированные ключевые слова, свои в каждом ЯП:

If .. then
If … elseif …
If … goto
If .. fi

Do … od
Do … end
Begin … end
<head … head>

Встречается провоцирующий ошибки синтаксис, например, формат переключателя в языке С требует оператора break для естественного завершения ветви.

Изредка встречаются одинаковые изображения разных понятий. Типичный пример: символ «;» в языке Паскаль понимается как разделитель операторов, а в Си – как завершитель оператора.

Для удобства отладки на безе СП часто практикуют синонимы или псевдонимы для длинных идентификаторов или сложных цепочек доступа.

Вариации влияют на удобочитаемость программ и распознавание принадлежности текста ЯП.

Лексически близкие ЯП могут привлекать удобочитаемостью на первых шагах отладки программ.

Таблица 4

Сравнение ЯП. Пример оператора ветвления

Pascal	*C*
If a = b then x := c else y := c	If a == b then x = c; else y = c;
If a = b then begin x := c; end else begin y := c; end	

Встречаются в СП средства настройки на желаемую лексику, но используют их нечасто. Похожую работу выполняют такие части СП, как препроцессоры, макрорасширения, сборка текста программы из ранее отлаженных модулей.

2.2. Синтаксис

При определении синтаксиса ЯП используются БНФ и синтаксические диаграммы в качестве системы координат при описании смысла текстов программ и её фрагментов. Удобный подход в терминах предшествования не всегда дает результат из-за существования более сложных контекстных взаимосвязей. В таких случаях используется понятие «синтаксическая позиция», позволяющее более точно налагать смысл на текст и при необходимости расширять определение синтаксиса так, что различимым смыслам программы соответствуют разные синтаксические позиции в БНФ, задающей ЯП. Синтаксические позиции в БНФ или синтаксической диаграмме можно задать номерами в их изображении или префиксами, на которых анализирующий автомат зайдёт в тупик.

Синтаксический маршрут позволяет в текста выделять эквивалентные цепочки.

Отсутствие пересечения множеств в регламенте БНФ препятствует лаконичному представлению отдельных проекций, возникает избыточная интегрированность описаний.

Встречаются эквивалентные формы для одной конкретизации понятия как в ориентированных на удобочитаемость языках Кобол, Робик, Haskell [31] или зависимость семантики от синтаксической позиции (Лисп – функция или значение, знаки унарных или бинарных операций во многих ЯП).

Разные наборы понятий ЯП – могут отсутствовать процедуры, указатели, ссылки, типы переменных и др. или различные оттенки общих понятий – роли в процессе – определённость на этапе анализа/компиляции или исполнения.

Проблемы синтаксического анализа хорошо исследованы. Сформулированы условия, называемые нормальными и каноническими формами, соответствие которым гарантирует эффективность алгоритма анализа и возможность автоматизации конструирования распознавателей и обработчиков, включая интерпретаторы и компиляторы. Известны эквивалентные преобразования БНФ, позволяющие, если это возможно, приводить определения ЯП к нормализованным или каноническим формам.

Цель преобразования синтаксических формул при определении анализаторов и компиляторов можно проиллюстрировать на схеме рекурсивного определения понятия «Идентификатор»:

БНФ	*Пояснение*

`Идентификатор ::= БУКВА`
`| Идентификатор БУКВА`
`| Идентификатор ЦИФРА`
	Леворекурсивная формула

Рис. 1. Определение идентификатора

Удобное для эффективного *синтаксического* разбора определение имеет вид

БНФ	*Пояснение*

`Идентификатор ::= БУКВА | БУКВА КонецИд`
 	
`КонецИд ::= БУКВА КонецИд`
`| ЦИФРА КонецИд`
`| ПУСТО`
	Праворекурсивная формула

Рис. 2. Эффективное определение идентификатора

Синтаксическая диаграмма	*Пояснение*
	Эквивалент праворекурсивной БНФ (рис. 2)

Рис. 3. Синтаксическая диаграмма анализатора

Этот пример показывает, что удобные для анализа формулы приведены к виду, когда каждую альтернативу можно выбрать по одному текущему символу.

Новые ЯП обычно избегают синтаксически сложно распознаваемых конструкций и сразу ориентируются на автоматическое построение распознавателей. Это способствует и уменьшению числа ошибок при подготовке программ.

Синтаксис большинства ЯП обеспечивает понимание программ с помощью комплекта ключевых слов, символизирующих конкретные шабло-

ны, управляющие конструированием кода программы и, тем самым, процессом её выполнения.

Таблица 5

Синтаксис бестипового учебного концентра языка Pascal[2]

БНФ	*Пояснение*
ident = letter {letter | digit}	Идентификатор
integer = digit {digit}	Целое
factor = ident | integer | "(" expression ")". expression = factor [("=" | "<" | "+"|"-" | "*" | "DIV") factor	Выражение: = < – Отношения между числами + - * DIV – Операции над числами
statement = [assignment | ProcedureCall | IfStatement | WhileStatement]	Оператор
assignment = ident ":=" expression	Присваивание
StatementSequence = statement {";" statement}	Последовательность исполнения команд
IfStatement = "IF" expression "THEN" StatementSequence ["ELSE" StatementSequence] "END"	Ветвление
WhileStatement = "WHILE" expression "DO" StatementSequence "END"	Цикл
IdentList = ident {"," ident}. FormalParameters = "(" [IdentList {";" = ["VAR"] IdentList }] ")". ProcedureHeading = "PROCEDURE" ident [FormalParameters]. ProcedureBody = declarations ["BEGIN" StatementSequence] "END" ident. ProcedureDeclaration = ProcedureHeading ";" ProcedureBody. declarations = ["CONST" {ident "=" expression ";"}] ["VAR" {IdentList;"}]	Определение функции
ActualParameters = "(" [expression {"," expression}] ")" . ProcedureCall = ident [ActualParameters]	Вызов функции
module = "MODULE" ident ";" declarations ["BEGIN" StatementSequence] "END" ident "."	

[2] Понятие «тип данных» будет привлечено позднее.

Обеспечивается внешняя понятность текста программы при её аналитическом чтении и оценке. При отладке не исключена потребность в блок-схемах для уяснения логики вложенных ветвлений и циклов.

Ряд ЯП представляет программы как непосредственно структуры данных, используемые при представлении и выполнении программы. В таком случае синтаксис много проще. Используется его конкретизация для семантической сопоставимости с другими ЯП.

Таблица 6

Синтаксис учебного концентра языка Lisp

Конкретизация синтаксиса для представления семантики	*Пояснение*
форма ::= переменная | (QUOTE S-выражение) | (COND {(форма форма)} | (функция {аргумент})	Переменная Константа Ветвление Вызов функции
аргумент ::= форма	
переменная ::= идентификатор	
функция ::= название | (LAMBDA список_переменных форма) | (FUNC название функция)	Имена, включая операции над списками Безымянная функция Именование функции
список_переменных ::= ({переменная})	Любое число, возможно ни одного
название ::= идентификатор	
идентификатор ::= атом	
Собственно синтаксис и лексика	
S-выражение ::= атом | (S-выражение . S-выражение) | ({S-выражение})	Атом Консолидация Список
атом ::= БУКВА {БУКВА | ЦИФРА}	

Исключение синтаксического сахара в ЯП обеспечивает внутреннее понимание сущности реализуемого алгоритма при подготовке и отладке программы автором.

Синтаксически подобные ЯП могут быть удобны при подготовке и преобразовании программ при их переносе на другие системы.

Вывод: разница в поддержке ПП проявляется на уровне как определения лексики и синтаксиса ЯП, так и предполагаемого уточнения отдельных понятий при их реализации в СП.

2.3 Семантика

В изучении парадигм программирования центральную роль играет семантика, поэтому следует вспомнить основные положения Венской методики[3], согласно которой языки программирования определяют в терминах универсальной функции и операционной семантики, используя типичные понятия программирования при спецификации систем программирования, что удобно для сравнительного описания парадигм программирования в понятном виде [26].

Благодаря хорошо разработанной концепции абстрактных объектов, позволяющей концентрировать внимание лишь на существенном и игнорировать второстепенные детали, Венский метод годится для описания и машин, и алгоритмов, и структур данных, особенно при обучении основам системного программирования. Согласно концепции абстрактных объектов (абстрактный синтаксис, абстрактная машина, абстрактные процессы) интерпретирующий автомат содержит в качестве компоненты состояния управляющую часть, содержимое которой может изменяться с помощью операций, подобно прочим данным, имеющимся в этом состоянии. При сравнении сложности интерпретаторов для разных языков можно использовать нормализованные предикатные формы, сопоставляя предикаты с выделяемыми понятиями.

Модель автомата с состояниями в виде древовидных структур данных, созданного согласно Венской методике для интерпретации программ, является достаточно простой[4]. Тем не менее, она позволяет описывать основные нетривиальные понятия программирования, включая локализацию определений по иерархии блоков вычислений, вызовы процедур и функций, передачу параметров. Такая модель поддерживает представление понятий программирования, полезное в качестве стартовой площадки для исследования разных парадигм программирования и сравнительного анализа стилей программирования.

Проблема определения ЯП и СП наиболее тщательно проработана в Венской методике определения языков программирования. Эта методика разработана в конце 60-х годов. Основная идея – использование абстракт-

---

[3] Венский метод (ВМ) определения языков программирования был разработан в 1968 году в Венской лаборатории IBM под руководством П. Лукаса на основе идей, восходящих к Дж. Маккарти.

[4] При сравнении императивного и функционального подходов к программированию П. Лэндин (P. J. Landin) предложил абстрактную машину SECD для спецификации машинно-зависимых аспектов семантики Лиспа

ного синтаксиса (АС) и абстрактной машины (АМ) при определении семантики языка программирования. Конкретный синтаксис (КС) языка отображается в абстрактный, а абстрактная машина может быть реализована с помощью конкретной машины (КМ), причем и отображение, и реализация могут иметь небольшой объем и невысокую сложность.

Диаграмма	*Пояснение*
КС ↔ АС → АМ → КМ	Существует отображение конкретного синтаксиса в абстрактный и обратно. Абстрактный синтаксис отображается в абстрактную машину. Абстрактная машина реализуется с помощью конкретной машины.

Рис. 4. Схема декомпозиции определения ЯП по Венской методике

Сущность определения языка концентрируется в виде так называемой универсальной семантической функции языка (УФ : АС → АМ), выполняющей переход от абстрактного синтаксиса к абстрактной машине – трансляцию (интерпретацию и/или компиляцию). УФ и АМ образуют определение функциональной и операционной семантики ЯП. Для производственных ЯП определение УФ достаточно сложно, поэтому часто определение семантики ЯП сводят к определению АМ.

Если КС удаётся нормализовать, то его перевод в АС и обратно (КС ↔ АС) можно построить автоматически. Граница между АС и АМ может быть установлена введением уровня базовых средств (БС), что гарантирует тривиальность перехода БС ↔ АМ, профилактику усложнённости АМ и взаимозаменяемость АМ и БС при реализации СП.

Диаграмма	*Пояснение*
КС ↔ АС АМ → ↓ ↕ КМ БС →	При отображении абстрактного синтаксиса в абстрактную машину выделяется уровень базовых средств ЯП, просто отображаемых на абстрактную машину и обратно. Переход к конкретной машине может быть выполнен и как реализация базовых средств.

Рис. 5. Схема декомпозиции определения ЯП с выделением уровня базовых средств

Выбор БС можно связать с минимальным учебным концентром ЯП, достаточным для ознакомления с идеями ЯП и перехода к самостоятельному овладению СП. Именно так Дж. Маккарти выделил Pure Lisp. В состав

БС включают операции ЯП и средства управления вычислениями, не сводимые к более простым средствам ЯП. Можно ограничит БС одной областью данных, рассчитывая, что операции над другими областями данных несложно включить как команды АМ.

Сложность реализации универсальной функции УФ: АС →БС → АМ преодолевается декомпозицией её на вспомогательные семантические проекции (ВС) – семантический спуск. Вспомогательные семантические проекции {ВС1,…,ВСn} представляют отдельные механизмы ЯП, эффективно сводимые к аппаратным решениям при реализации автомата АМ → КМ.

Диаграмма	*Пояснение*
КС ↔ АС АМ → ↓ ↕ КМ {ВС1,…,ВСn} → БС →	При определении абстрактного синтаксиса выделяются вспомогательные семантические функции, реализация которых возможно с помощью базовых средств. Определение вспомогательных семантик может быть выполнено средствами реализуемого ЯП.

Рис. 6. Схема декомпозиции определения ЯП
с разложением на вспомогательные семантики

Трудоёмкость реализации и жизнеспособность применения СП для определённого ЯП по ВМ зависит от того, удаётся ли декомпозировать УФ на достаточно простые, понятные и удобно реализуемые ВС. Нечто похожее А.Л. Фуксман называл «вертикальным слоением программ», поддерживающим «сосредоточенное представление рассредоточенных действий» [20]. Основные ВС представляют средства работы с памятью, организации вычислений, обработки структур данных и управления вычислениями. Возможно выделение диагностики, типового контроля, средств укрупнения действий и т.д.

При такой архитектуре можно СП рассматривать как три комплекта функций, обеспечивающих анализ программы, ее трансляцию (интерпретацию или компиляцию) и исполнение (вычисление или кодогенерацию). Главная задача анализа – обнаружить в тексте программы основные понятия и выделить, вывести или вычислить по тексту программы значения компонентов структуры, представляющей собой абстрактный синтаксис программы. Эта работа сводится к набору распознавателей и селекторов, названия которых могут быть выбраны в зависимости от смысла понятий, составляющих программу, а реализация варьируется в зависимости от кон-

кретного синтаксиса языка. Тогда при любом конкретном синтаксисе разбор программы выполняется определением, одноимённым анализу ее абстрактного представления, которое и грает роль спецификации. Набор распознавателей частично показывает сложность ЯП, но более точный показатель сложности появляется при определении универсальной функции, возможно требующей вспомогательных понятий, не имеющих явного представления на уровне ЯП, но полезных при описании его реализационных механизмов. Так, например, появилось понятие «замыкание функции» в дополнение к понятиям «определение функции» и «применение функции».

Любое определение анализа выглядит как перебор распознавателей, передающих управление композициям из селекторов, выбирающих существенные компоненты из анализируемой программы и заполняющих поля определенной структуры или значениями, или программами их вычисления. Содержимое полей предназначено для генерации кода программы, эквивалентного исходному тексту программы, а заодно и ее абстрактной структуре. Например, Лисп-форму PROG можно рассматривать как представление абстрактного синтаксиса для подмножества языка Паскаль, содержащего переменные, константы, арифметические операции и сравнения, пустой оператор, присваивание, последовательное исполнение операторов, условный оператор, безусловный переход GOTO и циклы. Тогда необходимо определить набор распознавателей, выявляющих эти понятия, и селекторов, выделяющих их характеристики. Селекторы имеют смысл лишь при истинности соответствующего распознавателя.

Таблица 7

Основные конструкции учебного концентра языка Pascal над целыми числами

`X`	Переменные
`123`	Константы
`C`	
`(A1 + A2)`	Вычисления
`(A1 = A2)`	Отношения
`;`	Пустой оператор
`X := a`	Присваивание
`S1; S2`	Последовательные операторы
`if p then ST else SF`	Ветвление
`while P do S`	Цикл
`var X`	Объявление переменной

`const C = 123`	Объявление именованной константы
`proc Pr (V…) S`	Определение процедуры
`Pr (A…)`	Вызов процедуры

Таблица 8

Основные конструкции учебного концентра языка Lisp над списками[5]

`X`	Переменная
`(quote C)`	Константой может быть любое символьное выражение
`(atom X)` `(eq X Y)`	Проверка символьного выражения на неделимость Совпадение адресов
`(cond (P ST)…(T SF) )`	Ветвление
`(lambda (X …) E…)`	Безымянная функция
`(defun F E)`	Именование функции
`(F A …)`	Вызов функции

Унификация значений и функций позволяет в базовые средства не включать именование функций, рассматривая такой механизм как вспомогательную семантику «Категории объектов», которые появятся в более полном определении ЯП. Формально Defun можно свести к передаче параметра с помощью Lambda.

Таблица 9

Абстрактный синтаксис учебного концентра языка Pascal над целыми числами представлен в форме списков

`X`	`(value X)`
`123`	`(value 123)`
`C`	`(value C)`
`var X`	`(var X)`
`const C = 123`	`(const C 123)`
`(A1 + A2)`	`(sum A1 A2)`
`(A1 = A2)`	`(eq A1 A2)`
`;`	`(empty)` или `(NIL)`
`X := a`	`(set X A)`

[5] Может сам работать как свой АС.

```
S1; S2;                  (step S1 S2)
if p then ST else SF;    (if P ST SF)
while p do S;            (while P S)
proc Pr (V…) S           (let Pr (V …) S)
Pr (A…)                  (Pr (A…))
```

Использование списков в качестве абстрактного синтаксиса позволяет все распознаватели (var, const, sum, eq, empty, assign, step, if, while) свести к анализу головы списка, что снимает вопрос конструирования или разработки анализатора.

Все селекторы (X,C,A1,A2,A,S1,S2,ST,SF,P,S) сводятся к композиции шагов доступа, выполняемых после соответствующего распознавателя. Такие определения, сводящиеся к выбору 2-го, 3-го или 4-го элемента списка, практически не требуют отладки, работают с первого предъявления.

Конструкцию while можно в БС не включать, т.к. она сводима к func. Можно выделить ВС «Представление итераций», которое в полном ЯП будет включать и другие форматы циклов.

ЯП с общим АС семантически эквивалентны, они сравнимы по трудоёмкости отладки программ.

Многие методы верификации программ активно используют структурную и рекурсивную индукцию[6] при доказательстве таких свойств рекурсивно определенных функций, как эквивалентность или правильность относительно модели. Поэтому операционная семантика ЯП, заданная над списочными структурами и поддерживающая все уровни определения языка от текста до кода, включая моделирование средств различных ПП, образует технологичную основу для решения актуальных проблем разработки надежных информационных систем.

2.4. Интерпретация программ

Определение УФ, обеспечивающее корректный переход от абстрактного синтаксиса программы к абстрактному коду АМ, требует доступа к представлению соответствия между именами и их значениями в зависимости от контекста и предшествующих определений. При интерпретации и на этапе компиляции программ такое соответствие представляют таблицей идентификаторов/атомов, в которой хранятся связи вида Имя-Смысл, преобразуемые по принципу стека, естественно отражающего динамику вызова функций. Важно обратить внимание на учет изменения контекста при

[6] Предложена Дж. Маккарти

последовательном выполнении шагов программы, а также на несовпадение порядка в тексте с очередностью выполнения композиций функций. Формально управляющие ходом вычислений операторы могут рассматриваться как функции, преобразующие полное состояние памяти V – контекст исполнения программы.

При задании операционной семантики необходимо отследить корректность обработки порождаемых структур данных, что для функций и операций может быть сформулировано как свойство чистого результата. Согласно этому свойству задана четкая дисциплина манипуляций со стеком, хранящем промежуточные результаты при обработке данных: каждая операция берет из стека в точности все свои аргументы и обязательно располагает в нем свой единственный результат. Обработка процедур может быть сведена к формату обработки функций формированием фиктивного результата.

Различают два стиля задания семантики ЯП – семантика вычисления значений и семантика изменений состояния памяти. Основой определения интерпретатора для семантики вычисления значений является функция EVAL (evaluation), вычисляющая произвольные выражения языка с учетом состояния таблицы атомов TA, устроенной как стек. Для семантики изменений состояний памяти функция EXEC (execution) выполняет ту же работу на базе двух устроенных как векторы таблиц TA и TF, раздельно хранящих значения переменных и определения процедур, соответственно.

Универсальная функция или интерпретация – это функция, которая может вычислять значение любой формы, включая формы, сводимые к вычислению произвольной заданной функции, применяемой к аргументам, представленным в этой же форме, по доступному описанию данной функции. (Конечно, если функция, которой предстоит интерпретироваться, имеет бесконечную рекурсию, интерпретация будет повторяться бесконечно.)

Определим универсальную функцию eval от аргумента expr – выражения, являющегося произвольной вычислимой формой языка Лисп.

Такая универсальная функция должна предусматривать основные виды вычисляемых форм, задающих значения аргументов, а также представления функций, в соответствии со сводом правил языка. При интерпретации выражений учитываются следующие решения, представленные при определении AC:

– атом обычно понимается как переменная. Для него следует найти связанное с ним значение. Например, могут быть переменные вида x, elem, смысл которых зависит от контекста, в котором они вычисляются;

- константы, представленные как аргументы функции QUOTE, можно просто извлечь из списка ее аргументов. Например, значением константы (QUOTE T) является атом T, обычно символизирующий значение «истина»;
- условное выражение требует специального алгоритма для перебора предикатов и выбора нужной ветви;
- остальные формы выражений рассматриваются по общей схеме как список из функции и ее аргументов. Обычно аргументы вычисляются, а затем вычисленные значения передаются функции для интерпретации ее определения. Так обеспечивается возможность писать композиции функций;
- если функция представлена своим названием, то среди названий различаются имена встроенных элементарных функций, такие как CAR, CDR, CONS и т.п., и имена функций, введенных в программе;
- для встроенных функций интерпретация сама «знает», как найти их значение по заданным аргументам, а для введенных в программе функций использует их определение, которое находит подобно переменной по имени в таблице атомов – в контексте[7];
- если функция построена с помощью λ-конструктора, то, прежде чем её применять, понадобится связывать переменные из λ-списка параметров со значениями аргументов;
- если представление функции начинается с DEFUN, то понадобится сохранить имя функции с соответствующим ее определением так, чтобы корректно выполнялись рекурсивные вызовы функции. Определения функций накапливаются в «хвосте» системной переменной TA, то есть работают как глобальные определения.

Таким образом, интерпретация выражений осуществляется как взаимодействие четырех ВС:
- обработка структур данных (cons, car, cdr, atom, eq);
- конструирование функциональных объектов (lambda);
- идентификация объектов (список параметров, set, defun);
- управление логикой вычислений и частичными вычислениями (композиции, quote, cond, eval).

[7] Похоже на резервированные слова, но в реальных СП все функции (более тысячи) равноправно являются встроенными функциями.

Идентификация и управление отчасти привязаны к синтаксическим позициям без специальной лексики («синтаксический сахар»). В большинстве ЯП аналоги первых двух подсистем нацелены на обработку элементарных данных и конструирование составных значений, кроме того, иначе установлены границы между подсистемами.

Обозначения:

[XL . YL] . AL работает как (pairlis XL YL AL) – функция аргументов XL,YL, AL строит список пар-консолидаций соответствующих элементов из списков XL, YL и присоединяет их к списку AL. Полученный список пар, похожий на таблицу с двумя столбцами, называется ассоциативным списком или таблицей атомов. Такой список может использоваться для связывания имен переменных и функций при организации вычислений интерпретатором.

AL [X] работает как (assoc X AL) – функция двух аргументов, X и AL. Если AL – таблица атомов, подобная тому, что формирует функция pairlis, то assoc выбирает из него первую пару, начинающуюся с X. Таким образом, это функция поиска определения или значения в таблице атомов.

X | Y работает как (append X Y) – сцепляет списки в один общий список.

X . Y работает как (cons X Y) – X становится «головой» списка Y.

@F – адрес подпрограммы, выполняющей функцию F.

Универсальная функция EVAL, которую предстоит определить, должна удовлетворять следующему условию: если представленная аргументом форма сводится к вычислению функции, имеющей значение на списке аргументов из этой же формы, то данное значение и является результатом функции EVAL.

Явное определение универсальной функции позволяет достичь четкости механизмов обработки Лисп-программ.

При написании на демонстрационном подмножестве Лиспа определения функции EVAL согласно приведенной выше спецификации, задаётся переход от данного списочного представления выражения E к его значению с учетом заданной таблицы атомов TA, хранящей значения атомов.

Вводим функцию EVAL для обработки форм и обращения к функциям. Эта функция использует таблицу атомов (TA) для хранения связанных имен – значений переменных и определений функций, изначально хранящую значения встроенных констант и адреса подпрограмм @car, @cdr), @cons, @atom и @eq, выполняющих работу встроенных элементарных функций. Это составляет исходный контекст выполнения программы. Значения (NIL . ()) и (T . T) обеспечивает, что атомы NIL и T обозначают сами

себя. ТА обязательно должна содержать глобальное определение встроенной константы «NIL», можно и сразу разместить в ней константу «T», выполняющую роль значения «истина».

Пусть ТА = ((Nil . ()) (T . T) (CAR . @car) (CDR . @cdr) (CONS . @cons) (ATOM . @atom) (EQ . @eq))

Таблица 10

Схема интерпретации Lisp-программ

Шаблон или предикат	*Шаг интерпретации (очередная форма или результат)*	*Примечание*
(eval expr TA)	Если форма expr может быть вычислена при заданном ТА, то её значение и есть значение универсальной функции EVAL	Первый аргумент EVAL – форма expr, второй – контекст или таблица атомов ТА.
TA [var]	TA [var]	Если форма – атом, то этот атом может быть только именем переменной, а значение переменной должно уже находиться в ТА
(QUOTE const)	Const	Если форма начинается с атома «QUOTE», то она представляет собой константу, значение которой выделяется как CADR от нее самой
(COND branchs)	(evcon branchs TA)	Если форма начинается с атома «COND», то форма – условное выражение или ветвление. Вводим вспомогательную функцию EVCON, которая обеспечит вычисление предикатов (пропозициональных термов) по порядку, выбор и вычисление формы, соответствующей первому предикату, принимающему значение «истина»
(fn . args)	(eval (fn . (evilis args TA)) TA)	Все остальные случаи рассматриваются как список из функции с аргументами, которые

		обрабатывается функцией EVILIS, затем представление функции, возглавляющее список вычисленных значений аргументов передаются функции EVAL
		Если функция – атом, то существует две возможности:
(CAR (x . y))	X	Атом может представлять одну из элементарных функций (CAR CDR CONS ATOM EQ). В таком случае соответствующая ветвь вычисляет значение этой функции на заданных аргументах, используя встроенные подпрограммы вычисления функций – @car, @cdr), @cons, @atom и @eq соответственно
(CDR (x . y))	Y	
(CONS x y)	(x . y)	
(ATOM x)	TA [x]	
(EQ x x)	T	
(EQ x y)	NIL	
(TA [fn] . x)	(eval (TA [fn] . x) TA)	В противном случае, этот атом – название ранее заданного определения функции. Определение можно найти в TA, подобно вычислению переменной
((LAMBDA var expr) arg)	(eval expr ([var . args] . TA))	Если функция начинается с LAMBDA, то ее аргументы попарно соединяются со связанными переменными и размещаются в TA, а тело определения (форма из λ-выражения) передается как аргумент функции EVAL для дальнейшей обработки
(DEFUN fn expr)	(TA | (fn . expr))	Если функция начинается с DEFUN, то ее название и определение соединяются в пару, и полученная пара размещается в таблице атомов TA, чтобы имя функции стало определенным при дальнейших вычислениях, работало а рекурсивных вызовах и как глобальный объект

Ветвям сводки синтаксиса соответствуют ветви универсальной функции. Определение универсальной функции является важным шагом, показывающим одновременно и механизмы реализации языков программирования, и технику функционального программирования на любом языке [10].

Для ЯП, не допускающих совместного хранения данных и функций, контекст исполнения программ образуют отдельные таблицы для переменных и подпрограмм – TA и TF, соответственно.

Пусть

TA = ((res . NIL) (TRUE . TRUE) (FALSE . FALSE))[8]

TF = ((sum . @sum) (def . @def) (mul . @mul) (div . @div) (eq . @eq) (lt . @lt) …)

Таблица 11

Схема интерпретации Pascal-программ

Выражение или оператор	*Exec : form TA TF -> TA' TF'*	*Примечание*
(value X)	((res . TA [X]) . TA) TF	Значение переменной или константы выбирается из таблицы TA
(var X)	TA [X] := NIL	Значение объявленной переменной не определено
(const NC C)	TA [NC] := C TF	Значение константы сохраняется в TA
(sum A1 A2)	(res . (exec (TF[sum] (exec A1 TA TF) (exec A2 TA TF) TA TF)) . TA) TF	Зарезервированная переменная res хранит результат вычислений
(eq A A)	((res . TRUE) . TA) TF	Аргументы совпадают
(eq A1 A2)	((res . FALSE) . TA) TF	Аргументы различны
(empty) или (NIL)	TA TF	Пустой оператор ничего не меняет
(set X A)	((X. A) . TA)) TF	(TA [X] := A)
(step S1 S2)	(exec S2 (exec S1 TA TF) TA TF) TF	Сначала интерпретируется шаг S1, затем S2 учитывает произведенный им на TA побочный эффект
(if TRUE ST SF)	(exec ST TA TF) TF	Выбор ветви в зависимости от предиката
(if FALSE ST SF)	(exec SF TA TF) TF	

[8] Зарезервированная переменная res хранит результат вычислений.

(while P S)	(WHILEV (exec P TA TF) P S) TF	Сведение цикла к рекурсии
(WHILEV FALSE S)	TA TF	
(WHILEV TRUE P S)	(exec (while P S) (exec S TA TF) TF)TF	
(proc Fn V S)	TA (TF | (Fn V S))	Объявление определения процедуры
(TF[Fn V S] A)	(exec S [V . A].TA TF) TF	Вызов процедуры

Основные различия:

- exec не поддерживает безымянные функции.
- eval не поддерживает присваивания переменным
- exec поддерживает раздельное хранение значений и определений функций/процедур
- eval допускает конструирование определений функций в процессе вычислений.

2.5 Абстрактная машина

Особенности процесса компиляции достаточно сложны даже для простых языков, поэтому спецификация результата компиляции часто задается в терминах языково-ориентированных абстрактных машин (АМ) [1]. Такой подход полезен для решения ряда технологических проблем разработки программных систем (мобильность, надежность, независимость от архитектур и т.п.). Абстрактная машина может рассматриваться как развитие и обобщение языка ассемблера, обусловленное возможностью использовать более сложные структуры данных в качестве регистров, или как часть интерпретатора, выполняющая вызовы подпрограмм (монитор). Система команд АМ представляет собой реализацию БС, дополненную рядом системных действий по передаче параметров и защите областей действия, подразумеваемых ЯП, но не имеющих чёткого синтаксического представления.

Абстрактная машина удобна для определения машинно-зависимых аспектов семантики ЯП. Такой подход позволяет не загромождать определение ЯП, добиться его прозрачности, но главное, такое определение может быть машинно-независимым и переносимым.

АМ различает обычно следующие категории команд:

- засылка значений из памяти в стек;
- вычисления над безымянными операндами в стеке при обработке выражений;
- пересылка значений из стека в память с учетом локализации;
- организация ветвлений и циклов;

– организация вызовов процедур и функций с сохранением/восстановлением контекста.

Могут быть и другие категории команд.

Абстрактная машина SECD (AML)

При сравнении императивного и функционального подходов к программированию, П. Лэндин (P.J. Landin) предложил специальную абстрактную машину SECD, удобную для спецификации машинно-зависимых аспектов семантики Лиспа. Подробное описание этой машины можно найти в книге Хендерсона по функциональному программированию [21].

Рассмотрим функциональный подход к низкоуровневой детализации программ при их реализации на уровне ассемблера, использованный при реализации языка Lisp. Понятие абстрактной машины введено для определения операционной семантики языка функционального программирования по Венской методике, а именно для отображения абстрактного синтаксиса языка на язык абстрактной машины. Кроме того оно позволяет проанализировать процедуру включения средств уровня ассемблера в высокоуровневую обстановку языка Lisp, опробованных при раскрутке Lisp-системы. Для языка Lisp такое определение написано на Lisp-е, как и определение интерпретатора[9].

Машина SECD работает над четырьмя регистрами: стек для промежуточных результатов, контекст для размещения именованных значений, управляющая вычислениями программа, резервная память (Stack, Environment, Control list, Dump). Регистры приспособлены для хранения выражений в форме атомов или списков. Состояние машины полностью определяется содержимым этих регистров. Поэтому функционирование машины можно описать достаточно точно в терминах изменения содержимого регистров при выполнении команд, что выражается следующим образом:

s e c d → s' e' c' d' – переход от старого состояния к новому.

Встраиваемые в ядро интерпретатора операции должны соответствовать стандартным правилам доступа к параметрам и размещения выработанного результата. Таким же правилам должен подчиняться и компили-

[9] Лисп-компилятор имеет уникальную историю, с которой можно ознакомиться в главе по ФП.

руемый код. Это позволяет формально считать равноправными встроенные и программируемые функции. Компилятор по исходному тексту программы строит код программы, эквивалентный тексту.

Для характеристики встроенных команд интерпретатора и результата компиляции программ базового Лиспа понадобятся следующие команды:

LD – ввод данного из контекста в стек;
LDC – ввод константы из программы в стек;
LDF – ввод определения функции в стек;
AP – применение функции, определение которой уже в стеке;
RTN – возврат из определения функции к вызвавшей ее программе;
RAP – применение рекурсивной функции
DUM – резервирование памяти для хранения аргументов рекурсивной функции.
SEL – ветвление в зависимости от активного (верхнего) значения стека;
JOIN – переход к общей точке после ветвления;
CAR – первый элемент из активного значения стека;
CDR – без первого элемента активное значение стека;
CONS – формирование узла по двум верхним значениям стека;
ATOM – неделимость (атомарность) верхнего элемента стека;
EQ – равенство двух верхних значений стека;
STOP – останов.

Стек устроен традиционно по схеме «первый пришел, последний ушел». Размер стека не ограничен. Каждая команда абстрактной машины «знает» число используемых при ее работе элементов стека, которые она удаляет из стека и вместо них размещает выработанный результат. Исполняются команды по очереди, начиная с первой в регистре управляющей программы. Машина прекращает работу при выполнении команды «останов», которая формально характеризуется отсутствием изменений в состоянии машины:

$$s\ e\ (\mathrm{STOP})\ d \rightarrow s\ e\ (\mathrm{STOP})\ d$$

Следуя Хендерсону, для четкого отделения обрабатываемых элементов от остальной части списка будем использовать следующие обозначения:

- (x . l) – это значит, что первый элемент списка – x, а остальные находятся в списке l.
- (x y . l) – первый элемент списка – x, второй элемент списка – y, остальные находятся в списке l и т.д.

Теперь мы можем методично описать эффекты всех перечисленных выше команд.

s e(LDC q . c)d → (q . s) e c d
(a b . s)e(CONS . c)d → ((a . b) . s) e c d
((a . b) . s)e(CAR . c) → (a . s) e c d
((a . b) . s)e(CDR . c) → (b . s) e c d

Для предиката оговариваем, в каких случаях вырабатывается значение «истина».

((a . b) . s)e(ATOM . c)d → (NIL . s) e c d
(A . s)e(ATOM . c)d → (T . s) e c d

Для неатомарных значений – NIL, , т.е. ложь, истина «T»- для атомов,

(a a . s)e(EQ . c)d → (T . s) e c d
(a b . s)e(EQ . c)d → (NIL . s) e c d

Истина «T» – для совпадающих указателей, для несовпадающих – NIL, т.е. ложь.

Для доступа к значениям, расположенным в контексте, можно определить специальную функцию N-th, выделяющую из списка элемент с заданным номером N в предположении, что длина списка превосходит заданный адрес.

s e (LD n . c) d → (x . s) e c d

где x – это значение (N-th n e)

При реализации ветвлений управляющая программа соответствует следующему шаблону:

(... SEL (... JOIN) (... JOIN) ...)

Ветви размещены в подсписках с завершителем JOIN, после которых следует общая часть вычислений. Для большей надежности на время выполнения ветви общая часть сохраняется в дампе – резервной памяти, а по завершении ветви – восстанавливается в регистре управляющей программы.

(NIL . s) e (SEL c1 c0 . c) d → s e c0 (c . d)
(T . s) e (SEL c1 c0 . c) d → s e c1 (c . d)

s e (JOIN) (c . d) → s e c d

Организация вызова процедур требует защиты контекста от локальных изменений, происходящих при интерпретации тела определения. Для этого при вводе определения функции в стек создается специальная структура, содержащая код определения функции и копию текущего состояния контекста. До этого в стеке должны быть размещены фактические параметры процедуры. Завершается процедура командой RTN, восстанавливающей регистры и размещающей в стеке результат процедуры.

s e (LDF f . c) d → ((f . e) . s) e c d
((f . ef) vf . s) e (AP . c) d → NIL (vf . ef) f (s e c . d)
(x) e (RTN) (s e c . d) → (x . s) e c d

где f – тело определения, ef – контекст в момент вызова функции, vf – фактические параметры для вызова функции, x – результат функции.

Рекурсивные вызовы функций требуют резервирования памяти для уникальных ссылок на разные поколения фактических аргументов, что достигается путем размещения фиктивного объекта «Ω», который функция rplaca, в порядке исключения, замещает на реальные данные.

S e (DUM . c) d → s (Ω . e) c d
((f . ef) vf . s) e (RAP . c) d → NIL (rplaca vf ef) f (s e c . d)

Для SECD реализационное замыкание включает в себя структуроразрушающие функции rplaca и rplacd, размещающие свой результат непосредственно в памяти второго аргумента. Это требует соответствующих ветвей в определениях синтаксиса и интерпретатора, что можно рассматривать как шаг раскрутки СП.

Таблица 12

Реализационное дополнение AML

	Новая конструкция	*Значение*	*Примечание*
Семантика	(RPLACA x (y . z))	(x . z)	Добавление новых операций над данными
	(RPLACD x (y . z))	(y . x)	

Таким же образом система команд AM может быть расширена простым дополнением правил. Так, можно её механизм распространить на другие типы данных, например, на целые числа:

(a b . s)e(SUM . c)d → ((a + b) . s) e c d
(a b . s)e(DIF . c)d → ((a - b) . s) e c d
(a b . s)e(MUL . c)d → ((a * b) . s) e c d
(a b . s)e(DIV . c)d → ((a / b) . s) e c d
(a . s)e(ADD1 . c) → ((a + 1) . s) e c d
(a . s)e(MIN1 . c) → ((a - 1) . s) e c d
(a . s)e(NUMBER . c) → (t . s) e c d

где t – истинностное значение NIL или T.

Соответствующее расширение ЯП может быть выполнено на уровне лексики, синтаксиса и семантики следующим образом:

Таблица 13

Расширение AML средствами обработки чисел.

	Новая конструкция	*Значение*	*Примечание*
Лексика	Number = digit {digit}	целое	Новая область данных
Синтаксис	атом = Number		Встраивание новой области в систему понятий ЯП
Семантика	(SUM x y)	(x + y)	Добавление операций над новыми данными
	(DIF x y)	(x - y)	
	(MUL x y)	(x * y)	
	(DIV x y)	(x / y)	
	(ADD1 x y)	(x + 1)	
	(MIN1 x y)	(x - 1)	
	(NUMBER x)	NIL или T	Предикат, выделяющий числа

Абстрактная машина SECM (AMP)

Аналогичная AMP для языков Pascal и Oberon содержит следующие команды:

LD – ввод данного из контекста в стек промежуточных значений;
LDC – ввод константы из программы в стек;
LDP – ввод определения процедуры;
LDW – ввод числа из памяти в стек;
STW – сохранение значения в глобальной памяти;

STE – сохранение значения в локальном контексте;
RET – возврат из процедуры к вызвавшей ее программе;
IF – ветвление в зависимости от активного (верхнего) значения стека;
EQ – равенство двух верхних значений стека;
LT – верхнее значение в стеке меньше, чем второе;
INC – увеличение числа на 1;
DEC – уменьшение числа на 1;
SUB – вычитание из верхнего элемента стека;
ADD – прибавление к верхнему элементу стека;
MUL – произведение двух чисел
DIV – частное от деления верхнего элемента стека на второй элемент;
BR – безусловная передача управления по абсолютному адресу;
AP – применение процедуры к аргументам, расположенным в стеке;
STOP – останов.

Машина SECM, как и SECD, работает над четырьмя регистрами: стек для промежуточных значений, контекст для размещения аргументов, локальных значений переменных и регистра возврата, управляющая вычислениями программа, память (Stack, Environment, Control program, Memory). Регистры приспособлены для хранения данных в форме векторов или списков, не пересекающихся по фактическим адресам. Состояние машины полностью определяется содержимым этих регистров. Поэтому функционирование машины можно описать достаточно точно в терминах изменения содержимого регистров при выполнении команд, что выражается следующим образом:

s e c m → s' e' c' m' – переход от старого состояния к новому.

Теперь мы можем методично описать эффекты всех перечисленных выше команд.

s e (LDC q . c)m → (q . s) e c m
(a . s) e (INC . c)m → (a+1 . s) e c m
(a . s) e (DEC . c)m → (a-1 . s) e c m
(a b . s) e (ADD . c)m → (a+b . s) e c m
(a b . s) e (SUB . c)m → (a-b . s) e c m
(a b . s) e (MUL . c)m → (a*b . s) e c m
(a b . s) e (DIV . c)m → (a/b . s) e c m
(a a . s) e (EQ . c)m → (TRUE . s) e c m
(a b . s) e (EQ . c)m → (FALSE . s) e c m

(a b . s) e (LT . c)m → (t . s) e c m где t имеет значение «TRUE» или «FALSE»

(TRUE . s) e (IF ct cf . c) m → s e (ct . c) m

(FALSE . s) e (IF ct cf . c) m → s e (cf . c) m

Используем дополнительные обозначения:

[x] – содержимое памяти по адресу x

e[n] – содержимое n-го элемента контекста

A(Pr) – число аргументов процедуры Pr

L(Pr) – число локальных переменных процедуры Pr

@c – адрес позиции «c» в программе

_ – Произвольное значение (_ подчерк)

s e(LDW x . c) m → ([x] . s) e c m

s e (LD n . c) m → (e[n] . s) e c m

(a . s) e (STW x . c) m → s e c (m[x]=a)

(a . s) e (STE x . c) m → s (e[x]=a) c m

s e (LDP @Pr. c) m → (@f L(Pr) A(Pr) . s) e c m

(@Pr KL N a1 ... aN . s) e (AP . c) m → s ((KL+N) _ ... _ a1 ... aN @c . e) Pr m

s (K e1 ... eK p . e) (RET . c) m → ( ) e p m

s e(BR @Pr . c) m → s e Pr m

Разница между SECD и SECM в основном сводится к операциям над данными. Кроме того, SECM имеет дополнительные команды по непосредственной работе с памятью для реализации присваиваний и передаче управления для реализации итераторов.

Обе абстрактные машины, как SECD, так и SECM, содержат, кроме образа базовых средств ЯП, дополнительные команды, обеспечивающие пересылки данных между регистрами и реализационные средства, поддерживающие эффективность реализации ЯП (rplaca, BR, метки и др.). В результате АМ способна выполнять вычисления несколько более широкого класса, чем задано БС данного ЯП. Это приводит к идее определения реализационного замыкания ЯП в соответствии с фактическими возможностями АМ. Такое замыкание выполняет роль ядра ЯП.

Предстоящие шаги инкрементального, т.е. пошагового, процесса раскрутки СП могут быть связаны с расширением типов обрабатываемых данных, подключением удобных форматов управления вычислениями, спе-

циализацией особых категорий функций, созданием программного инструментария и т.д. до полного или практически достаточного покрытия ЯП его реализацией в СП.

Дальнейшее развитие СП заключается в пошаговом расширении спектра обрабатываемых данных и присоединении ВС до полного покрытия ЯП его реализацией в СП. Каждая новая область данных подключают к АМ как комплект подпрограмм, реализующий предикат для выяснения принадлежности данного новой области и операции обработки данных. ВС, как правило, могут быть определены в терминах самого ЯП с помощью средств его ядра.

Интеграция вспомогательной семантики новых компонент ЯП в ранее реализованную версию СП осуществляется комплексным включением ряда согласованных определений на всех уровнях определения ЯП:

- дописывание БНФ;
- добавление ветвей в определение интерпретатора;
- дополнение АМ новыми командами.

В случае SECM кроме помимо образа БС фактически реализована работа с векторами и указателями (адреса в памяти). Поэтому реализационное замыкание учебного концентра языка Pascal естественно будет включать в себя обработку структур данных. Соответствующие, фактически поддержанные на уровне АМ, правила языка:

Данные = array of

АМ: x[i] := y
X := y [i]
*x := @y доступ по указателю к адресу

Допустимость таких операций существенно зависит от распределения памяти и независимости её частей, что приводит к необходимости контроля границ памяти при индексировании векторов и доступе по указателю. Информация для такого контроля во многих ЯП представляется в форме предписания типа данных (ТД) переменным. Таким образом, реализационное замыкание ЯП над SECM включает в себя представление ТД для всех переменных, включая параметры процедур.

Таблица 14

Встраивание нового понятия в определение синтаксиса ЯП

Новая конструкция	*Пояснение*	*Пример*
selector ={"[" expression "]"}.	Индексирование доступа к элементу вектора	[ind]
factor = ident selector	Доступ к элементу вектора	X [ind]
assignment = ident selector ":=" expression	Присваивание элементу вектора	X [ind] := 6
ArrayType = "ARRAY" expression "OF" type	Вектор как тип данных	ARRAY 10 OF integer
type = ident | ArrayType	Типы данных: целое или вектор	integer
FPSection = ["VAR"] IdentList ":" type	Раздел параметров процедуры, возможно изменяемых	VAR x,y : integer
FormalParameters = "(" [FPSection {";" FPSection}] ")"	Список формальных параметров процедуры	(a,b: integer ; VAR x,y : integer)
declarations = ["CONST" {ident "=" expression ";"}] ["TYPE" {ident "=" type ";"}] ["VAR" {IdentList ":" type ";"}]	Объявления констант, типов данных и переменных	CONST id = 2+3; TYPE arr10 = ARRAY 10 OF integer; VAR a1,a2,a3 : intger; xx,yy : arr10 ;

Дальнейшее расширение ЯП м.б. сведено к подключению ТД и присоединению ВС методом раскрутки.

ЯП с общей АМ семантически равномощны, на их основе достижима сравнимая эффективность процессов вычислений.

Трудоёмкость реализации АМ с помощью конкретной машины можно оценить на основе материала по низкоуровневому программированию, а также при сравнении АМ с Пи-кодом и RISK-машиной, предложенными Н. Виртом при разработке учебных языков Pascal и Oberon [2,4], и учебными машинами MIX и MMIX, описанными Д. Кнутом.

Семантический спуск от полного ЯП к его БС характеризуется снижением трудоёмкости реализации ядра ЯП и его АМ, сопровождаемое увеличением трудоёмкости применения частичной реализации ЯП.

Трудоёмкость применения можно оценивать числом понятий, возникающих при программировании сверх тех, что присущи решению типовых задач.

Цели раскрутки:

- снижение трудоёмкости программирования;
- увеличение потенциала СП, т.е. числа типовых задач, решение которых обладает приемлемой для практики трудоёмкостью.

2.6 Компиляция программ

При изучении требований к компиляции программ и анализе схем определения компилятора обращает на себя внимание тот факт, что для многих ЯП такое определение может быть представлено средствами самого ЯП. Венская методика определения языков программирования, а именно, отображение абстрактного синтаксиса языка на язык абстрактной машины, позволяет компиляцию программ рассматривать как один из методов оптимизации процессов, осуществляемый как символьное преобразование – трансляция с исходного языка высокого уровня на язык низкого уровня, допускающий оптимизирующую кодогенерацию.

Компилятор – это программа, которая транслирует конструкции уровня ЯП, определяющие процесс вычисления, в эквивалентный им объектный код, представляющий вызовы машинных подпрограмм. Это средство оптимизации, позволяющее ускорить работу программ от двух до ста раз в сравнении с простой интерпретацией. Опыт показывает, что скомпилированная программа может работать намного быстрее, чем интерпретируемая программа, в зависимости от ее природы. Скомпилированные программы могут быть и экономичнее с точки зрения расхода памяти, требуя 50–80% от полного объема.

При компиляции программ обычно составляется план распределения памяти для значений переменных в зависимости от их типов данных, выполняется размещение локальных данных в памяти, частичный контроль доступа к переменным и совместимости операций и операндов по типам данных, вычисление значений выражений (констант, переменных, элементов структур данных, результатов операций и вызовов функций), манипуляции по управлению вычислениями [17].

Различают два подхода к организации процесса компиляции, отличающиеся выбором в качестве компилируемой единицы всей программы или части её функций или процедур. При компиляции всей, полной программы создаётся автономно исполняемый код, функционирование которого зависит только от входных данных. Раздельная компиляция функций или процедур предполагает, что исполнимый код подпрограммы является многократно используемым компонентом, встраиваемым в разные программы, включая исходную СП, превращаемую, таким образом, в реализацию проблемно ориентированного расширения ЯП.

При определении компилятора на уровне абстрактной машины обычно выделено описание реализационного минимума ЯП, послужившего базой для раскрутки языка и основой для функционального программирования. Необходимо лишь ввести некоторые ограничения, гарантирующие при переходе к низкоуровневому программированию сохранение важнейших свойств функционирования программ. При компиляции выражений эти ограничения формулируются как чистый результат правильного выражения:

- все аргументы убраны из стека;
- результат выражения записан в стек.

При задании операционной семантики важно отследить корректность обработки порождаемых структур данных, что может быть сформулировано как свойство чистого результата. Согласно этому свойству задается четкая дисциплина манипуляций со стеком при обработке данных: каждая операция берет из стека в точности все свои аргументы и обязательно располагает в нем свой единственный результат.

Рассматривая компиляцию программ как один из методов статической оптимизации процессов, декомпозицию программы на категории функций с разным уровнем отладки можно использовать как отправную точку при выборе оптимизационных решений, связанных с сокращением времени компиляции недостаточно отлаженных программ.

Когда компилятор вызывается для компиляции функций. Он находит определение функции в таблице идентификаторов. Компилятор транслирует найденное определение в объектное выражение, которое представляет собой подпрограмму на языке ассемблера – АМ. Ассемблер после этого ассемблирует код подпрограммы. Затем в таблице для этой функции размещается ссылка на код подпрограммы.

Таким образом, обработка каждой функции происходит в три шага. Во-первых, выражение, задающее функцию, транслируется в текст на уровень ассемблера. Во-вторых, текст программы на уровне ассемблера транслиру-

ется в код программы. И, наконец, если никаких ошибок не обнаружено, то программа может быть исполнена. Когда некоторые ошибки указывают на появление необъявленной переменной, компилятор предупреждает об этом и продолжает работу, допуская свободные переменные. Такая диагностика будет дополнительно уточнена при анализе значений переменной на этапе выполнения программы.

Переменная связана, если она встречается в списке формальных аргументов. Когда переменная используется как свободная, это значит, что она должна быть связана в другой функции на более высоком уровне. При интерпретации функций может быть обнаружена переменная, не связанная вообще, о чем система известит пользователя соответствующим диагностическим сообщением об ошибке.

При написании большой программы лучше отлаживать отдельные функции, используя интерпретатор, а компилировать только те из них, которые уже хорошо изучены и отлажены.

Программист, планирующий выбор между компилятором и интерпретатором, должен обратить внимание на следующие моменты.

Нет необходимости компилировать все функции, которые используются лишь эпизодически. Содержащие пару «интерпретатор-компилятор» СП обеспечивают интерпретатору доступ к скомпилированным функциям. Компилированные функции, использующие интерпретируемые функции, могут вычислять их непосредственно при счете.

Порядок выполнения компиляции не имеет значения. Нет даже необходимости определять все функции до тех пор, пока они не понадобятся при счете.

Свободные переменные в компилируемых функциях должны объявляться до компиляции функций.

Последнее требование проясняет роль типового контроля в стандартных СП, ориентированных на компиляцию программ без интерпретации. Компиляция всех объектов осуществляется без анализа отсутствующих на этапе компиляции фактических данных, а это и означает, что на момент компиляции переменные являются свободными. Интерпретация располагает более полной информацией, связывающей необходимые для вычислений переменные с конкретными значениями, тип которых определен.

Обычно в СП существует механизм пакетов, позволяющий управлять составом функций и объектов, включаемых в комплект. При удалении части системы освободившаяся память может быть использована повторно. Имеются реализации, в которых выделено минимальное ядро системы, все остальные функции загружаются по мере необходимости, а процедура ос-

вобождения памяти может рассматривать память, занятую неиспользуемыми функциями, как свободную.

При трансляции функций в подпрограммы концепция переменных отображается в распределении памяти, в которой размещаются значения аргументов. Для обычных переменных распределение памяти – это стек. Другие функции не могут знать адреса таких переменных, что и не позволяет рассматривать их как свободные.

Свободные переменные удобны для коммуникации между компилируемыми программами, но не всегда могут четко служить коммуникации между интерпретируемыми и компилируемыми программами.

Еще один тонкий аспект – функциональные константы и функциональные аргументы, полезные при определении и применении функций высших порядков, таких как операторы и отображения, в которых отображающие функции могут быть фактическими аргументами, значение которых – определение функции.

Определение универсальной семантической функции, обеспечивающей корректную трансляцию абстрактного синтаксиса программы в ее абстрактный код на АМ, требует реализации соответствия между именами и их значениями в зависимости от контекста и предшествующих определений. При интерпретации такое соответствие представлялось таблицей атомов (ТА), в которой хранятся связи вида Имя-Смысл, преобразуемой по принципу стека, естественно отражающего динамику вызова функций. При компиляции не принято сохранять имена на время исполнения программы: их роль выполняют сопоставленные именам адреса. Поэтому вместо ТА вида ((a . 1)(в . 2)(c . 3)...) применяется два списка Name = (a в c ...) и Adr = (1 2 3 ...), хранящих имена переменных и адреса их значений на согласованных позициях. Обрабатываются эти два списка синхронно: оба удлиняются или сокращаются на одинаковое число элементов.

Формально операторы могут рассматриваться как функции, преобразующие полное состояние памяти V. Пусть функция E списочному представлению оператора сопоставляет эквивалентную ему функцию, использующую контекст, где N – свободная переменная, задающая список имен, известных в программе.

Важно обратить внимание на учет изменения контекста при последовательном выполнении шагов программы, а также на несовпадение порядка в тексте с очередностью выполнения композиций функций.

В процедурных ЯП принято раздельное хранение значений и кодов подпрограмм, поэтому используется более сложный контекст, где свободная переменная N задаёт список переменных, а M – список процедур, из-

вестных в программе. В таком случае последние две строки таблицы несколько видоизменяются.

Таблица 15

Пример спецификации компилятора операторов как выражений

	(λ (v)exp)	Вычисление новых состояний памяти
X	(λ (v)(assoc X N v))	N – свободная переменная, задающая список имен переменных, известных в программе
(quote C)	(λ (v)c)	
(sum A1 A2)	(λ (v)(+ ((E A1)v) ((E A2)v)))	
(eq A1 A2)	(λ (v)(= ((E A1)v) ((E A2)v)))	
(NIL)	(λ (v) v)	Состояние памяти V неизменно
(set X A)	(λ (v)(replace N v X ((E A)v)))	Замена происходит по указанному адресу
(step S1 S2)	(λ (v) (E S2 (E S1 v)))	
(cond (P ST) (T SF))	(λ (v) (funcall (COND (((E P)v)((E ST)v)) (T((E SF)v))) v)	
(while P S)	(λ (v) ((λ (w)(COND (((E P)v) (w ((E S)v))) (T v))) #'(λ (v)(COND (((E P)v) (w ((E S)v))) (T v)))))	Циклу соответствует безымянная функция, строящая внутренний контекст
(proc F (V …) S)	(λ (v) (rplace N v F (λ (X) S)))	
(F A…)	(λ (v) ((assoc F N v) (A ...)))	

Традиция не сохранять информацию о ТД на период исполнения обременяет компиляторы вспомогательной функцией контроля и вывода ТД. Эта функция может рассматриваться как спусковая, в зависимости от которой происходит кодогенерация или диагностика ошибок.

Таблица 16

Пример спецификации компилятора операторов описания и вызова подпрограммы

		(λ (v)exp)	Вычисление новых состояний памяти
proc Pr (X…) S	(let Pr (X …) S)	(λ (v) (rplace M v Pr (λ (X) S)))	Встраивание процедуры в контекст программы. M – свободная переменная, задающая список имен процедур, известных в программе
Pr (A…)	(Pr (A…))	(λ (v) ((assoc Pr M v) (A ...)))	Применение ранее определённой процедуры

X = <type, adr, g/l>
Adr = g : абсолютный в общей памяти
L : относительный в рабочем стеке аргументов и локальных переменных.

Выделены синтаксические позиции, требующие контроля при компиляции.

Таблица 17

Реализационное дополнение AMP

Ident = letter {letter | digit}	Первое вхождение идентификатора должно быть его описанием
integer = digit {digit}	Величина числа не должна превышать машинного слова или четырех байт
selector ={"[" expression "]"}.	Индекс при выбор элемента вектора должен быть целым числом
Factor = ident selector | integer | "(" expression ")".	Индекс при выбор элемента вектора не должен выходить за границы вектора
expression = factor [("="|"<" | "+"|"-" | "*" | "DIV") factor]	Операнды по типу данных должны соответствовать операциям, т.е. быть числами
statement = [assignment | ProcedureCall | IfStatement | WhileStatement]	Виды операторов

assignment = ident selector ":=" expression	Присваивание левой части должно быть разрешено и согласовано по типу с правой частью
StatementSequence = statement {";" statement}	
IfStatement = "IF" expression "THEN" StatementSequence ["ELSE" StatementSequence] "END"	Управляющий ветвлением предикат должен быть логического (булевского) типа
WhileStatement = "WHILE" expression "DO" StatementSequence "END"	Управляющий циклом предикат должен быть логического (булевского) типа
IdentList = ident {"," ident}	Идентификаторы в списке должны различаться
ArrayType = "ARRAY" expression "OF" type	Размер вектора задается целым числом
Type = ident | ArrayType	Идентификатор типа данных или встроен в СП или описан в программе
FPSection = ["VAR"] IdentList ":" type	Идентификаторы в списке должны различаться
FormalParameters = "(" [FPSection {";" FPSection}] ")"	Идентификаторы в списке должны различаться
ProcedureHeading = "PROCEDURE" ident [FormalParameters]	Заголовок процедуры
ProcedureBody = declarations ["BEGIN" StatementSequence] "END" ident	Тело процедуры
ProcedureDeclaration = ProcedureHeading ";" ProcedureBody	Определение процедуры
declarations = ["CONST" {ident "=" expression ";"}] ["TYPE" {ident "=" type ";"}] ["VAR" {IdentList ":" type ";"}]	Все идентификаторы при описании должны различаться
ActualParameters = "(" [expression {"," expression}] ")"	
ProcedureCall = ident [ActualParameters]	Список фактических параметров соответствует по длине и последовательности типов данных списку формальных параметров процедуры
module = "MODULE" ident ";" declarations ["BEGIN" StatementSequence] "END" ident "."	

Контроль форматов и ТД

Таблица 18

Вспомогательная семантика для контроля типов данных и границ областей памяти

Конструкция	*Синтаксис*	*Пример*	*Условие согласования*	*Диагностика при отсутствии согласования*
Selector	ident "[" expression "]"	X [ind]	Величина индекса не превосходит размер вектора	Выход индекса за пределы вектора. Выбор элемента из скаляра.
Expression	ident [("="|"<"|"+"|"-"|"*"|"DIV") ident]	A=3 A<9 3+a 5-b 2*a 7 DIV x	Числа заведомо допустимы, а для идентификаторов переменных надо проверить типы их значений	Тип операнда не соответствует операции
Assignment	ident selector ":=" expression	X [ind] := 6	Величина индекса не превосходит размер вектора и типы левой и правой части согласованы	Выход индекса за пределы вектора. Выбор элемента из скаляра. Несоответствие типов левой и правой части
IfStatement	"IF" expression	If a=3 then x:=a else y:=a	Тип значения выражения булевский	Предикат должен иметь значение TRUE или FALSE
WhileStatement	"WHILE" expression			
ArrayType	"ARRAY" expression	ARRAY 10 OF integer	Длина вектора задана целым числом	Длина вектора должна быть целым числом
FPSection		VAR x,y : integer	Все имена различны	
FormalParameters	"(" [FPSection {";" FPSection}] ")"	(a,b: integer ; VAR x,y : integer)	Все имена различны	В списке параметров совпадают имена переменных
Declarations		CONST id = 2+3;	Значение выражение известно при	В списке описаний совпадают идентификаторы.

			компиляции.	Константа невычислима при компиляции
		TYPE arr10 = ARRAY 10 OF integer;	Длина вектора задана целым числом	
		VAR a1,a2,a3 : integer; xx,yy : arr10 ;	Все имена различны	
ProcedureCall	ident [ActualParameters]	Pr (1,2, a,b)	Длина списка фактических параметров совпадает с длиной списка формальных параметров и типы данных в соответствующих позициях согласованы.	Список аргументов длиннее или короче списка параметров. Типы аргументов не согласованы с типами соответствующих параметров

При определении компилятора на уровне абстрактной машины должно быть выделено описание реализационного минимума ЯП, послужившего базой для раскрутки СП и основой для практического программирования. Необходимо лишь ввести некоторые ограничения, гарантирующие при переходе к низкоуровневому программированию сохранение важнейших свойств функциональных программ. Эти ограничения формулируются как чистый результат правильного выражения:

- все аргументы убраны из стека;
- результат выражения записан в стек;
- при выходе из процедуры её формальный результат исключается из стека.

Обозначение:

{X/N} результат компиляции выражения X в контексте N.

Таблица 19

Определение Лисп-компилятора на Лиспе

S	Ветвь compile	Код secd
X	(list 'LD (adr S N))	(LD N[X])
(quote C)	(list 'LDC (cadr S))	(LDC C)
(car X) (cdr X) (cons X Y)	(append (comp-(cadr S) N)'(CAR)) (append (comp-(cadr S) N)'(CDR)) (append (comp-(caddr S) N) (comp-(cadr S) N) '(CONS))	({X/N}CAR) ({X/N}CDR) ({X/N}{Y/N}CONS)
(atom X) (eq X Y)	(append (comp-(cadr S) N)'(ATOM)) (append (comp-(cadr S) N) (comp-(caddr S) N) '(EQ))	({X/N}ATOM) ({X/N}{Y/N}EQ)
(set X A)		
(if P ST SF)	(let ((then (list (comp-(caddr S) N) 'JOIN)) (else (list (comp-(cadddr S) N) 'JOIN))) (append (comp-(cadr S) N) (list 'SEL then else))))	({P/N} SEL ({ST/N} JOIN)({SF/N} JOIN))
(lambda (X …) E…)	(list 'LDF (comp-(caddr S) (append (cadr S) N)) 'RTN)	(LDF {E/((X …) . N)} RTN)
(defun F E)		
(let E ((V Def)…))	(let* ((args (value (cddr S))) (mem (cons (var (cddr S)) N)) (body (append (comp-(cadr S) mem) '(RTN)))) (append (map #'(lambda(x)(comp- x N)) args) (list body 'AP)))	Определяющие выражения Определяемые переменные Компиляция тела блока в новом контексте Соединение скомпилированных определений с телом блока ({Def/N} … (({E/((V …) . N) RTN)} AP))
(labels E ((F Def)…)	(let* ((args (value (cddr S))) (mem (cons (var (cddr S)) N)) (body (append (comp-(cadr S)	Рекурсивные определения Определяемые функции

	mem) '(RTN)))) (append '(DUM (map #'(lambda(x)(comp- x mem)) args) (list 'LDF body 'RAP))))) (DUM LDF {E/((E) . N) F RTN RAP)	Рекурсивные определения компилируются в новом контексте и соединяются с телом блока (DUM {Def/((F ...). N) … (LDF (E{/((F ...). N)} RTN) RAP))
(F A1 … AK)		(LDC NIL {AK/N} CONS … {A1/N} CONS LDF {F/N} AP)
	(T (append (map #'(lambda(x)(comp- x N)) (cdr S)) (list body 'AP)))	

Исследование разных схем частичных, смешанных, «ленивых» вычислений и метакомпиляции приводит к выводу о целесообразности совмещения таких схем в рамках общей СП с целью использования их преимуществ на разных уровнях изученности решаемых задач.

Частичные вычисления допускают прогон программы при неполном комплекте входных данных. В результате выполняются те операции, для которых имеются данные, и строится остаточная программа, которую можно выполнить с недостающими данными и получить итоговый результат, такой, как если бы все данные были заданы изначально.

Смешанные вычисления допускают произвольную разметку программы на выполнимые и задержанные действия. Выполняются маршруты, которым задержанные действия не препятствуют и выводится остаточная программа, которая может быть выполнена после снятия блокировки с задержанных действий [7].

Ленивые вычисления выполняются в зависимости от необходимости операций для получения результата с запоминанием промежуточных значений выражений для экономии повторных вычислений [21,27,31].

Метакомпиляция обрабатывает программу совместно с комплектом типовых данных [32].

Традиционно система программирования может содержать пару «интерпретатор – компилятор». Между этими понятиями трудно установить формальную границу. Любой интерпретатор содержит элементы, реализация которых описывается в машинных терминах: структура памяти, реали-

зация двоичных деревьев и т. п. Любая компилированная программа содержит интерпретируемые элементы, например, обращения к файловой системе и другим элементам ОС. На практике достоинства интерпретации проявляются при отладке программ, а преимущества компиляции – при эксплуатации готового программного продукта. Более подробное обсуждение этой темы заслуживает отдельного разговора.

Теория программирования утверждает, что определение компилятора может быть выведено из определения интерпретатора методами смешанных вычислений. Это методы, допускающие частичную обработку программ при неполных или избыточных данных с построением остаточной программы, которую можно применять по мере уточнения данных. Компилятор по такой методике получается как остаточная программа при смешанном вычислении интерпретатора. Теоретически для определения языка программирования достаточно построить определение интерпретатора, хотя практичность реальной системы программирования обычно обеспечивается оптимизирующей компиляцией и кодогенерацией программ. Но здесь речь идет не об эффективном компиляторе, а лишь о понятном описании семантики.

2.7 Диалекты языков программирования

Ряд проблем создает неоднозначность семантической декомпозиции сложных определений, трудоемкость определения реализационных особенностей полных интерпретаторов и компиляторов, а также немалое число команд в абстрактных машинах для реальных ЯП. Кроме того, как показывает опыт определения классов объектов в языках ООП, критерии функционального назначения и обусловленность типами данных недостаточны для выделения простых семантических систем, удобных для сравнительного анализа программных средств. Поэтому возникает необходимость в разложении определения языка на концентры, уровни или слои. Для классификации наиболее важно выделение минимального учебного концентра и реализационного ядра. В результате множество ЯП можно структурировать на классы реализационно подобных и содержательно сравнимых языков, обладающих общими, точнее – эквивалентными, концентрами или слоями[10]. Нередко реализационное ядро языка содержит ряд понятий, не имеющих прямого представления в анализируемом ЯП. Дополнительные

[10] Согласно Венской методике определения ЯП эквивалентными считаются языки, сводимые к одному абстрактному синтаксису.

понятия обеспечивают реализационную полноту языка, существенно влияющую на понимание механизмов эффективного программирования.

Практика применения ЯП часто порождает проблемно-ориентированные расширения востребованных подмножеств ЯП, что является питательной почвой для формирования диалектов ЯП. Сохраняя первозданную сущность ЯП, диалекты могут внешне существенно варьироваться и различаться реализационными особенностями, сопутствующими библиотеками, подразумеваемым оборудованием.

Достаточно ясно различимы следующие виды диалектов:

- учебные концентры для ознакомления с основными идеями;
- практичные подъязыки для программирования решений типовых задач;
- проблемно-ориентированные вариации для расширения сферы применения;
- полные языки для исследования и выбора улучшений.

При комплектации ядра системы программирования проявляются технические детали организации ее рабочего цикла, функциональные средства оперативного мониторинга за фактическим составом системы и ее взаимодействия с внешним миром, необходимые для отладки СП.

Моделирование языка программирования на идеализированном базовом Лиспе (Pure Lisp) вполне может послужить иллюстрацией быстрого определения операционной семантики сложных языков программирования.

Ядро интерпретатора может быть реализовано следующим образом:

- выбирается реализация списков в виде бинарных деревьев, листья которого рассматриваются как атомы, а узлы используются для выстраивания списков (левые ветви – элементы списка, правые – продолжение списка или конец списка, т.е. пустой список);
- выбирается реализация атомов как объектов, внутренняя структура которых при определении и исполнении функций не всегда существенна, но при необходимости доступна специальным операциям;
- встраивается специальный атом, являющийся реализацией пустого списка ();
- встраивается операция, связывающая различные данные с атомами, и ассоциируется с атомом DEF;
- определяются правила доступа к параметрам встроенных операций с размещением их результата и встраивается специальная операция (монитор), выполняющая применение операций к правильно размещенным аргументам (SUBR);

- встраивается операция, строящая из атомов и списков более сложные структуры (списки и узлы из любых элементов), и ассоциируется с атомом CONS;
- встраиваются операции, выполняющие разбор и анализ структур, и ассоциируются с атомами CAR, CDR, ATOM, EQ, представляющими эти операции;
- встраиваются специальные операции (псевдо-функции), выполняющие блокировку вычислений, выбор ветви и конструирование определений функций, и ассоциируются с атомами QUOTE, COND и LAMBDA, соответственно;
- универсальная функция ассоциируется с EVAL;
- определяется рабочий цикл передачи данных интерпретатору и вывода результата интерпретации.

Такое ядро представляет собой машинно-зависимую часть интерпретатора. Встраивание операции в ядро системы – это включение в его реализацию исполнимого кода, который является реализацией этой операции. Адрес такого кода ассоциируется с именем атома, с помощью которого будет организовано выполнение операции при интерпретации программ.

При ассоциировании атомов с произвольной информацией можно использовать специально организованный ассоциативный список, построенный из пар, содержащих атомы и их определения.

Ассоциативный список работает как стек: при многократных определениях доступно самое новое из них, т.е. расположенное ближе к началу списка. Если мы знаем адреса кода операций, встроенных в ядро системы, то можем соответствие между именами операций и адресами их кода хранить в ассоциативном списке. Можно считать, что начальное состояние ассоциативного списка содержит таблицу соответствия между именами и адресами операций.

СП часто строят пошаговым образом (метод раскрутки).

При реализации экспериментально-учебных языков и систем программирования цель применения раскрутки – минимизация стартовых трудозатрат, основанная на учете формальной избыточности средств языков программирования. Можно выделить небольшое ядро, на основе которого методично программируется все остальное. Каждый шаг реализации по схеме раскрутки должен обеспечивать:

- уменьшение трудоемкости последующих шагов,
- отладку прототипов сложных компонентов,
- подготовку демонстрационного материала.

Выбор конкретных шагов раскрутки можно соотнести с декомпозицией определения языка программирования на синтаксические и семантические, функциональные и машинно-ориентированные, языково-ориентированные и системные аспекты. При такой декомпозиции можно на первых шагах как бы «снять» синтаксическое и семантическое разнообразие языка, как имеющее чисто технический характер. Именно в этом смысл выделения элементарного Лиспа. Такая методика может быть успешна при освоении любого класса задач, информацию о котором можно представить в виде частично формализуемых текстовых и графовых форм [6].

Реальный состав системы и возможности ее компонентов можно исследовать с помощью специальных функций, предоставляющих информацию о включенных в систему объектах и их свойствах.

Особенности работы с файлами, основные приемы их открытия, задания специфики их функционирования и обмена данными с обычными символьными объектами поддерживают организацию рабочего цикла программы независимо от базовых средств ЯП.

Диалекты ЯП часто бывают реализационно равнозначны, возможно семантически эквиваленты и равномощны, но могут быть лексически и синтаксически различимы.

2.8 Структуры данных

Кроме распределения памяти при компиляции достаточно популярны следующие подходы, обеспечивающие эффективность работы с памятью в процессе исполнения программ:

- new – delete – динамические запросы к системе памяти типа «куча»;
- компактизация – уплотнение пространства с целью размещения крупных целостных объектов;
- «близнецы» – метод укрупнения памяти и профилактики её чрезмерной фрагментации при распределении на разно объемные блоки;
- стек – дисциплина доступа FILO
- Setl – более 17-ти разных СД, динамически выбираемых СП для представления множеств в зависимости от характера их обработки и наличия свободной памяти.

Типичная реализация структур данных во многих системах программирования:

1. Векторы с паспортом, хранящим при компиляции сведения о размерах и типе элементов.
2. Запись или структура, обеспечивающая доступ к заданному перечню разносортных элементов по статически определённым ключам.
3. Объединение заданного перечня разных ТД, размещаемых в разное время по определённому адресу.
4. Множество небольшого числа перечислимых элементов, обработки которых не выводит за пределы машинных команд над битовыми кодами.
5. Стек, допускающий две дисциплины доступа, – FILO и вектор.

Представление списков и повторное использование памяти.

В машине списки хранятся не как последовательности символов, а как структурные формы, построенные из машинных слов как частей деревьев, подобно записям в Паскале при реализации односвязных списков.

Преимущества списков для хранения данных в памяти:

1. Размер и даже число выражений, с которыми программа будет иметь дело, можно не предсказывать. Кроме того, можно не организовать для размещения выражений блоки памяти фиксированной длины.
2. Ячейки можно переносить в список свободной памяти, как только исчезнет необходимость в них. Даже возврат одной ячейки в список имеет значение, но если выражения хранятся линейно, то организовать использование лишнего или освободившегося пространства из блоков ячеек трудно.
3. Выражения, являющиеся продолжением нескольких выражений, могут быть предоставлены однократно.

В любой момент времени только часть памяти, отведенной для списков, действительно используется для хранения полезных данных. Остальные ячейки организованы в простой список, называемый списком свободной памяти.

Самым интересным, можно сказать, революционным, механизмом работы с памятью в языке Lisp, бесспорно, стала «сборка мусора». С начала 1960-х годов методам такой работы посвящены многочисленные исследования, которые продолжаются до наших дней и сильно активизировались в связи с включением похожего механизма в реализацию языка Java.

Общая идея всех таких методов достаточно проста:

– пока памяти хватает, о ней можно не беспокоиться и располагать новые данные в новых блоках памяти;
– если памяти вдруг не оказалось, то надо выполнить «сборку мусора», в процессе которой, возможно, найдутся ставшие бесполезными для программы блоки;
– если память нашлась, ее снова можно тратить.

Специальная программа «Сборщик мусора» выполняет анализ достижимости всех блоков памяти простой пометкой узлов, видимых из конечного числа рабочих регистров системы программирования. К таким регистрам относятся промежуточные результаты вычислений, активная часть стека, ассоциативный список, таблица атомов и др. После пометки все непомеченные узлы объединяются в список свободной памяти, передающий их для повторного использования новым вызовам функции CONS. Такая автоматизация не лишена недостатков, но они обнаруживаются лишь в сравнительно сложных процессах, требования которых мы сейчас не учитываем.

Первые реализации освобождения памяти действовали по следующей схеме [27]:

Определенный регистр FREE содержит информацию о первой ячейке списка свободной памяти. Когда требуется слово для формирования дополнительной структуры данных, берется первое слово списка свободной памяти, а код в регистре FREE заменяется на информацию о втором слове списка свободной памяти. Не требуется никаких программных средств для того, чтобы пользователь программировал возврат ячеек в список свободной памяти. Этот возврат происходит автоматически при прогоне программы, где бы ни исчерпался список свободной памяти. Программа, восстанавливающая память, называется "Сборщик мусора". Любая часть списка, доступная программе, рассматривается как активный список и не затрагивается мусорщиком. Активные списки доступны программе через фиксированный ряд базисных ячеек, таких как таблица атомов и регистры, хранящие промежуточные результаты вычислений.

Сложные структуры списков могут быть произвольной длины, но каждое активное слово должно быть достижимо от базисной ячейки цепью CAR-CDR. Любая ячейка, недостижимая таким образом, недоступна для программы и не активна, поэтому ее содержимое не представляет интереса. Неактивные, то есть недоступные программе ячейки восстанавливаются мусорщиком в списке свободной памяти следующим образом. Во-первых, каждая ячейка, к которой можно получить доступ через цепь CAR-CDR,

метится установлением отрицательного знака. Где бы ни выявилось отрицательное слово в цепи во время процесса пометки, мусорщик «знает», что остаток раскручиваемого списка содержит уже помеченные ячейки. Затем мусорщик предпринимает линейное выметание освободившегося пространства, собирая ячейки с положительным знаком в новый список свободной памяти и восстанавливая исходный знак активных ячеек.

Иногда структуры списка указывают на полные слова, такие как печатные имена и числа. Мусорщик не может пометить эти слова, потому что знаковый разряд использован. Мусорщик должен прекратить работу, потому что указатели, расположенные в адресе и декременте таких слов, бессмысленны.

В этом случае проблемы повторного использования памяти решаются расположением полных слов в зарезервированной области памяти, называемой областью полных слов. Мусорщик должен прекращать прослеживание, как только покидает поле свободной памяти. Пометка в поле полных слов осуществляется с помощью отдельной таблицы битов.

Такая реализация экономична в отношении памяти, но она имеет ряд неприятных следствий: непредсказуемые длинноты (время работы) при поиске очередной порции ячеек и «перегрев системы», если такие порции слишком малы для продолжения счета. По мере роста производительности оборудования разработаны простые и не столь обременительные алгоритмы повторного использования памяти на базе параллельных процессов и профилактического копирования активных структур данных в дополнительные блоки памяти. Такие методы не требуют сложной разметки и анализа достижимости. Содержательная аналогия с мастерским мытьем посуды, то есть не допуская переполнения раковины, вполне отражает метод stop© принятый в современных реализациях.

Обычно при обработке программ в памяти располагаются разносортные результаты, возникающие при разборе и анализе текста программы и ее данных, при построении ее внутреннего кода и при формировании результата. В Lisp-системах традиционно при всех этих видах работ принято придерживаться принципов логики здравого смысла:

- новые значения строятся на новом месте,
- предпочитаются интересы малых программ,
- автоматизация повторного использования памяти на первых шагах разработки освобождает программиста от необходимости уделять внимание техническим проблемам распределения памяти.

Кроме того, почти исключается необходимость присваиваний, они в программах заменяются именованием.

Память обычно распределена по блокам, содержащим ряд слов, образующих структуры данных. Физический объем памяти, логическая длина данных и состав информации, полезной для продолжения вычислений, могут существенно различаться. Минимизацию потерь в результативности работы с памятью дает динамическая обработка бинарных деревьев – нет простоев из-за незаполненности части полей. Каждый узел такого дерева имеет небольшой объем, достаточный для хранения двух типизированных указателей (CAR и CDR, левый и правый). Типизация указателей нужна для оперативного динамического контроля соответствия данных и операций по их обработке. NIL, атомы, списки, числа, строки – все это реализационно различимые типы данных. Утверждение о бестиповости языка Lisp устарело и в наши дни заменилось признанием полноты типового контроля в динамике исполнения программ. Ранее бестиповостью называлось отсутствие статического связывания в тексте программы имен переменных с типами их значений. Для компиляции приходится дополнять Lisp-программы сведениями о типах значений свободных переменных, но далеко не каждая программа доживает до компиляции. Существуют реализации, поддерживающие автоматический вывод типов данных. Языку Lisp свойственна функциональная классификация значимых типов данных, т.е. именно реализационно различимых.

Реализация бинарных деревьев или односвязных списков описана в классических курсах по программированию, а реализацию атомов мы рассмотрим подробнее. Эффективность приведенного выше определения интерпретатора с использованием ассоциативного списка существенно зависит от числа различимых атомов в программе. Такая зависимость обычно смягчается механизмом функций расстановки (хэш-функций), обеспечивающим доступ к информации по ключу. В качестве ключа используется имя атома. В результате вся связанная с атомом уникальная информация становится легко и быстро достижимой. Структура такой информации называется списком свойств атома. Она представляет собой чередование так называемых «индикаторов» и «значений» свойств. Число свойств атома не ограничено. Свойства бывают встроенные, системные или вводимые программистом. Значения атомов, адреса встроенных операций, определяющие выражения функций – примеры встроенных свойств.

Обычно с машиной связывается представление о блоках информации фиксированного объема, таких как слова, байты, регистры. Функциональное программирование и ООП нацелены на более крупные построения –

структуры данных не ограниченной заранее длины. Такие структуры достаточно эффективно реализуются посредством специального стека, приспособленного к обработке произвольного числа компонентов текущего уровня иерархической структуры данных. От обычного стека он отличается выделением указателя на конец текущего уровня.

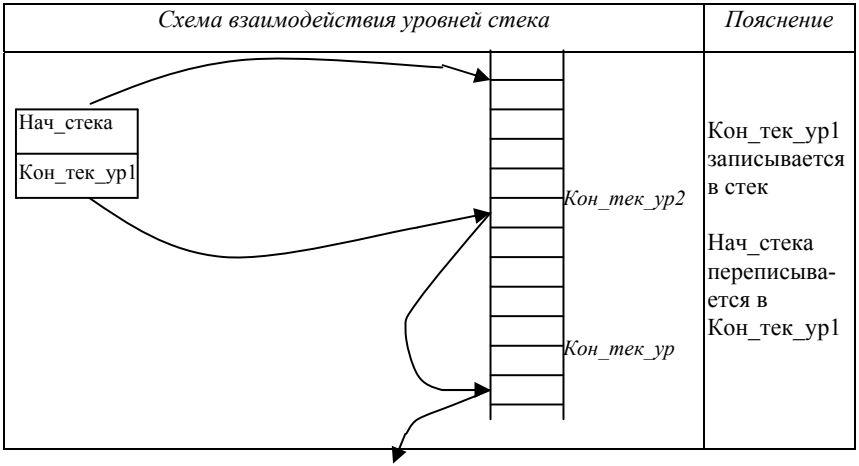


Рис. 7. Многоуровневый стек.

При переходе на новый внутренний уровень Кон_тек_ур1 записывается в стек, Нач_стека переписывается в Кон_тек_ур1, а адрес новой вершины стека становится значением Нач_стека. В результате стек хранит ссылки на границы уровней, что обеспечивает возможность возврата на любой нужный уровень, в частности, восстановления процесса обработки в случае неожиданных ситуаций.

Значительный резерв производительности функциональных программ дают деструктивные функции, являющиеся формальными аналогами чистых функций, но при выполнении сопровождаемые побочными эффектами. Такой подход позволяет при необходимости повышать эффективность программ, отлаженных в стиле без использования присваиваний, простым привлечением деструктивных аналогов функций под ответственность программиста, точно знающего границы допустимых изменений хранимых данных.

Непосредственная польза от сопоставления графического вида с представлением списков в памяти поясняется при рассмотрении функций, работающих со списками, на следующем примере из [27]:

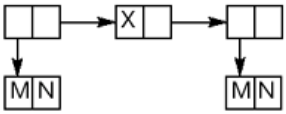
Диаграмма	*Пояснение*
	При создании структур многократные вхождения одинаковых данных получают независимое расположение в памяти.

Рис. 8. Пример ((M . N) X (M . N))

Возможное для списков использование общих фрагментов ((M . N) X (M . N)) может быть представлено графически:

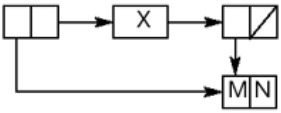
Диаграмма	*Пояснение*
	Слияние многократных вхождений одинаковых данных.

Рис. 9. Графическое представление эффективного размещения данных

Непосредственное текстовое представление в точности такой структуры невозможно, но ее можно построить с помощью одного из выражений:

(LET ((a '(M . N))) (SETQ b (LIST a 'X a)))

((LAMBDA (a) (LIST a 'X a))'(M . N))

2.9 Реализационная прагматика

При классификации ЯСП ключевое значение имеет семантика, но для уяснения ПП, поддерживаемой ЯП, требуется понимание реализационной прагматики (РП), которая может быть не представлена в определении или стандарте ЯП, но подразумеваться традиционно.

Реализационная прагматика, затрагивая все уровни определения ЯП, в основном представляет решения в области конкретной организации работы

с памятью, уточняющей решения и принципы, провозглашенные в определении АМ. В первую очередь это относится к вопросам защиты областей памяти и их конечности, т.е. реагирования на дефицит памяти.

Реализационная прагматика, поддерживающая разные ЯП:
ФП – списки, мусорщик, списки свойств атома
СП – Пэф, векторы, ТД (переменная-значение)
ЛП – разностные списки, последовательный перебор, откатка
ООП – ссылки, Вирт и абс методы и классы, множественное наследование

ЯП с общей РП реализационно равнозначны, они сравнимы по трудоёмкости реализации СП.

Выводы:

1. Поддержка ПП при определении ЯП и реализации СП выражается в реализации средств организации вычислений, механизмов обработки параметров и использования СД и их размещения в памяти, методов контроля за вычислениями и управления ходом вычислений. Успех применения ПП зависит от того, в какой мере используемые ЯСП поддерживают выбранную парадигму.
2. Лексически близкие и синтаксически подобные ЯП могут иметь существенные различия на уровне семантики и реализационной прагматики.
3. Диалекты ЯП часто бывают реализационно равнозначны, возможно семантически эквиваленты и равномощны, но различимы по эксплуатационной прагматике, лексике и синтаксису:
    - Учебные концентры для ознакомления с основными идеями.
    - Практичные подъязыки для программирования решений типовых задач.
    - Проблемно-ориентированные вариации для расширения сферы применения.
    - Полные языки для исследования и выбора улучшений.

3.ХАРАКТЕРИСТИКА ПАРАДИГМ ПРОГРАММИРОВАНИЯ

Функциональный подход к исследованию и спецификации основных парадигм программирования позволил проанализировать и методично описать особенности языков программирования разного уровня от ассемблера до языков параллельного программирования [3]. В результате сложилось

сравнительное описание эксплуатационной и реализационной прагматики основных парадигм. Предложена схема описания и определения парадигматических характеристик языка программирования. В качестве иллюстрации использованы фрагменты языков программирования разного уровня, относящиеся к парадигмам машинно-ориентированного, системного, императивного, функционального, логического, объектно-ориентированного, параллельного и высокопроизводительного программирования.

3.1 Прагматика

Парадигмы программирования различаются нишей в жизненном цикле программ, приоритетами при оценке качества программ, выбором инструментов и методов обработки данных. Значимость используемых при этом критериев по существу зависит от условий применения и методов реализации программируемых решений, что можно называть эксплуатационной и реализационной прагматикой. Упорядочение критериев нередко претерпевает изменения по мере развития сферы применения программы, роста квалификацией пользователей, модернизации оборудования, информационных технологий и программотехники, что и приводит к появлению новых парадигм.

Первые парадигмы автоматного и машинно-ориентированного императивного программирования сформировались как кодирование готовых алгоритмов, сложившихся и отлаженных в докомпьютерную эпоху [15]. Системное и функциональное программирование появляются при расширении класса задач, требующих разработки и отладки новых алгоритмов и приведения их в удобную для обычного, непосредственного, программирования форму [2, 4, 27]. Затем приходит пора языков представления знаний и, соответственно, декларативного (логического) программирования рецептов решения новых задач, где факт существования решений важнее их эффективной реализации [13]. Профессионализация программирования приводит к объектно-ориентированному программированию, поддерживающему повторное использование запрограммированных готовых решений в разных системах. На повестке дня – создание удобных языков параллельного и высокопроизводительного программирования [25].

Новые возможности компьютерных сетей, суперкомпьютеров, общедоступных баз данных, массовое распространение мобильных устройств – все это меняет сферу применения информационных технологий и влечет кристаллизацию новых и более общих парадигм компьютерных языков. В данной работе представлена попытка характеризовать парадигмы про-

граммирования как взаимодействие эксплуатационной и реализационной прагматик. Эксплуатационная прагматика формируется как экспертная оценка требований к результатам применения парадигмы программирования. Реализационная прагматика представляется как уточнение операционной семантики, начиная с четырех основных семантических систем: обработки данных, их хранения и структурирования, управления обработкой данных в системах программирования.

При отработке методики проявления парадигматической характеристики ЯП в виде уточнения взаимодействия основных семантических систем[11], таких как обработка данных, их хранение и структурирование, управление обработкой данных, выделились три уровня парадигм, отражающие расширение языковой поддержки жизненного цикла программ и рост реализационной сложности определения ЯП:

- низкоуровневое кодирование;
- программирование на языках высокого уровня;
- подготовка программ на базе языков сверхвысокого уровня.

Низкоуровневое программирование характеризуется аппаратным подходом к организации работы компьютера, нацеленным на доступ к любым возможностям оборудования. В центре внимания – конфигурация оборудования, состояние памяти, команды, передачи управления, очередность событий, исключения и неожиданности, время реакции устройств и успешность реагирования [1, 5]. При хранении данных и программ используется глобальная память и автоматная модель управления обработкой данных.

Программирование на языках высокого уровня приспособлено к заданию структур данных, отражающих природу решаемых задач. Активно используется иерархия областей видимости структур данных и процедур их обработки, подчиненная структурно-логической модели управления, допускающей сходимость процесса отладки программ. Ассемблер в качестве предпочтительного изобразительного средства на некоторое время уступил языкам Паскаль и Си даже в области микропрограммирования.

Подготовка программ на базе языков сверхвысокого уровня нацелено на представление регулярных, эффективно реализуемых структур данных, при обработке которых возможны преобразования представления данных и программ, использование подобий и доказательных построений, гарантирующих высокую производительность вычислений и надежность процесса

[11] Понятие «семантическая система» предложено С.С. Лавровым [9].

разработки программ, приспособленных к вариациям архитектурных решений.

Для каждой парадигмы существуют причисляемые к ней, ей соответствующие, можно сказать, «референтные» или «опорные» языки программирования, и во многих языках представлен или реализован ряд парадигм. При исследовании и спецификации парадигматических свойств таких языков естественно определение языка представлять объединением определений монопарадигматических подъязыков, являющихся фрагментами исходного языка.

3.2 Семантические системы

Декомпозиция учебного концентра, объединенного с реализационным ядром, определения ЯП на вспомогательные семантические системы дает не слишком сложные параметры для сравнительного анализа и установления парадигматических отношений между ЯП. Относящиеся к одной парадигме подъязыки разных языков легко сравнить на уровне структурированной таким образом операционной семантики, анализируя определения их абстрактных машин [1, 3, 26] и интерпретаторов. Это позволяет с каждой парадигмой связать варианты реализационных особенностей, уточняющих основные семантические системы и правила их взаимодействия, что можно назвать реализационной прагматикой. К основным системам операционной семантики отнесены выполнение вычислений, работа с памятью, управление вычислениями и организация структур данных. Выделение таких систем обусловлено различиями в схеме применения операций к операндам, связанными с методами реализации СП и наличием их аппаратной поддержки.

Вычисления характеризуются совпадением типов результата и аргументов. Традиционные формы вычислений позволяют строить потоки вычислений и конвейерные процессы, управляемые готовностью данных, без именования промежуточных значений. Такие системы можно различать по мощности множества допустимых значений, набору встроенных операций над ними и возможности конструировать новые операции.

Работа с памятью осуществляется как операции над неявной таблицей, связывающей адреса и хранимые значения. Встречаются разные ограничения на обращение к этой таблице, формулируемые как дисциплина доступа к памяти или правила видимости именованных данных. Кроме того, различается техника обработки таблицы и ее структура. Чаще всего встречаются системы памяти, ориентированные на работу с ассоциативно именуемыми

значениями (value-oriented), с именованными переменными (name-oriented), с прямыми указателями (pointer-oriented) и неявными копиями значений в стеке (stack-oriented).

Управление вычислениями позволяет варьировать ход порожденных программой процессов в зависимости от данных или событий, символизирующих определенную логику корректности вычислений или успеха функционирования системы, связанных со спецификой реализации предикатов, представления различимых событий и реакций на события. Противопоставляются системы с фиксированным или программируемым набором схем управления. Встречаются системы с защитой процессов и/или данных. Возможна организация приостановок, учета временных отношений, обработки прерываний, приёма сообщений, синхронизации действий и взаимодействия потоков.

Организация структур данных обеспечивает конструктивность сложных построений, возможность восстановления составляющих вплоть до элементарных данных и их эффективной обработки «по частям». Противопоставляются целостные и распределенные структуры данных, статическое и динамическое размещение данных в памяти, аналитические, счётчиковые и программные методы повторного использования памяти для структур данных.

Допустимые вариации при определении соответствия конкретной парадигме на уровне семантических систем в компьютерных языках можно проиллюстрировать на примерах парадигм низкоуровневого кодирования, системного, функционального, логического и объектно-ориентированного программирования, а также языков сверхвысокого уровня, показывающих характерные идеи, средства и методы, послужившие основанием для кристаллизации отдельной парадигмы. Пары «интерпретатор (универсальная функция)» и «абстрактная машина», задающие операционную семантику фрагментов рассматриваемых языков, используются как параметр при классификации парадигм и сравнении языков, анализе их подобия и сопоставимости. Определения ограничены рассмотрением фрагментов языков, включающих следующие сквозные понятия:

Таблица 20

Список понятий ЯП, различаемых операционной семантикой, определяемыми для сравнения

Понятие	*Пояснение*
Атом/Скаляр	Атомы и скаляры могут быть разных категорий или типов, различаемых динамически или декларативно.
Структура данных	Возможны ограничения на характер элементов структуры, их число и динамику их изменений
Переменная	Может быть инициирована до вычислений, ограничена предписанным типом данных, заданным статически или выводимым по программе
Значение	Разные типы значений связаны с различными операциями их обработки и синтаксическими позициями в тексте программы, допускающими или требующими их вхождение
Выражение	Форма, результат которой может быть вычислен и использован как параметр в других формах
Действие/Операция	Встроенная команда или подпрограмма, рассматриваемая как элементарная база при организации вычислений
Условие/Логика	Концепция истинностных значений может требовать как специального типа данных, так и рассматриваться как нагрузка обычных значений (0, NIL)
Функция[12]	Возможно параметризованный фрагмент программы, представляющий укрупнённую единицу, используемую наравне с операциями при организации вычислений
Аргумент	Фактический параметр используемой функции.
Вызов функции	Форма, используемая для исполнения функции при заданных параметрах
Определение функции	Форма, представляющая фрагмент программы, предназначенный для использования в качестве функции
Идентификатор/Имя	Уникальная форма, создаваемая как синоним многократно используемого элемента данных

[12] Операторы управления, процедуры, макросы и т.п. рассматриваются как отдельные категории функций – укрупнение действий.

При неформальной характеристике стиля программирования отмечаются различия в акцентах при ответе на следующие вопросы:

- В чем заключается основной метод обработки программы при отладке?
- Как пронаблюдать результативную активность программы?
- Когда принимается решение о продолжении незавершённых вычислений?
- В каких пределах планируется функционирование участков повторяемости?
- Каким способом гарантирована корректность сложной информационной обработки?

Варианты ответов:

В чем заключается основной метод обработки программы при отладке?

- Интерпретация текста программы, приводящая к результату её выполнения.
- Интерпретация структуры программы, приводящая к результату её выполнения.
- Компиляция текста программы, приводящая к коду программы, выполнение которого дает результат.
- Сборка кода программы из готовых типовых компонентов.
- Редактирование заранее подготовленных шаблонов.
- Генерация кода по верифицированной спецификации цели программы.

Как пронаблюдать результативную активность программы?

- Изменение состояния отдельных элементов памяти.
- Вычисление значения выражения.
- Протокол обмена данными между программой и пользователем.
- Изображение хода вычислений в виде диаграммы.

Когда принимается решение о продолжении незавершённых вычислений?

- Обработка прерываний.
- Получение диагностических сообщений.
- Повторный прогон программы.
- Подготовка обработчиков прерываний.

В каких пределах планируется функционирование участков повторяемости?

- Пока имеется свободная память.
- Задано максимальное число повторений тела цикла или функции.
- Известна временная граница для выполнения любой команды, включая вызов процедуры.

Какие механизмы гарантируют корректность сложной информационной обработки?

- Статическая проверка соответствия типа данных переменных и операций.
- Защитные условия и инварианты.
- Динамический контроль типа значений и допустимости операций.
- Верификация программ на моделях.
- Конструирование программ, корректных по построению.

3.3. Концептуальные языки

Активно используемые ЯСП обычно втягивают в себя средства разных парадигм, что затрудняет их изучение. Строго говоря, практически любой ЯП позволяет использовать библиотеку модулей, поддерживающих нужную парадигму. При сравнении удобно использовать определение ряда моно-парадигматических языков, по которым можно устанавливать в какой мере изучаемый ЯП поддерживает конкретную парадигму. Выбор элементов такого ряда концептуальных ЯП входит в задачу данного материала и отчасти осуществлен при описании основных парадигм программирования.

Описание концептуального языка, приспособленной для его сравнения с другими языками, содержит следующие части:

1. Список общих понятий – показывает уровень сопоставимости сравниваемых ЯП.
2. АС – позволяет оценить глубину проработки используемых понятий.
3. АМ – показывает масштаб переносимости СП.
4. РП – дает оценку трудоёмкости реализации СП
5. Схема интерпретации или компиляции ЯП – даёт показатель организованности процесса вычислений.
6. Примеры программ решения типовых задач для иллюстрации концепции ЯП позволяют продемонстрировать парадигматические вариации в решении типовых задач.

Кроме того, ЭП представляет экспертную оценку требований к условиям применения ЯСП и критериев успешности результата программирования, что дает основания для рекомендаций по выбору ЯП, поддерживающего требуемую ПП.

3.4 Определитель парадигмы

Первые языки программирования обладали машинной ориентированностью и поддерживали принципиально важную, но небольшую по длительности и трудозатратам часть ЖЦП – от 2-х до 5 %, заключающуюся в кодировании готовых алгоритмов в терминах автоматов. Появление ЯВУ расширило языковое покрытие ЖЦП примерно до 10 % для хорошо поставленных задач, имеющих алгоритмы решения над типовыми структурами данных, приспособленные для нисходящих методик программирования. Парадигма функционального программирования посягнула на ЖЦП для задач с исследовательским компонентом и расширила его языковое покрытие до 50 % благодаря механизмам хранения и накопления информации о свойствах информационных объектов, полезной при отладке и модификации программ, составленных из небольших универсальных компонент, допускающих как нисходящую, так и восходящую методику разработки. Логическое программирование распространило эти механизмы на не вполне определенные постановки задач, что дало языковую поддержку предварительному сбору фактического материала, созданию демонстрационных версий, пробному прототипированию, отчасти тестированию и довело общее языковое покрытие ЖЦП почти до 60 % при восходящей методике разработки. Появление объектно-ориентированного программирования преодолело итеративность ЖЦП для задач, связанных с развивающимися областями приложения, что довело языковое покрытие примерно до 80 %.

Далее полнота языкового покрытия ЖЦП обеспечивается компьютерными языками, возникающими в связи с информационными технологиями, телекоммуникациями, распределенными информационными системами, электронным общением, автоматизацией и самоорганизацией управления проектами, а также специализированными языками для решения конкретных классов задач. Анализ концепций и классификация представительного свода КЯ в контексте профессиональной речевой практики в настоящее время стихийно разворачивается при формировании открытых онтологий и энциклопедий, что расширяет возможности для решения проблемы глубинного анализа содержания обучения информатике и информационным технологиям

Для практичности следует принять во внимание следующие положения:

- Определение парадигматической характеристики ЯП начинается с формулировки ясной общей идеи, задающей принципы, по которым можно отнести язык к конкретной парадигме.
- В центре внимания – анализ понятий, общих для разных языков.
- Реализационная прагматика языка характеризуется списком фрагментов определения ЯА, при описании которых могут потребоваться уточнения, дополнительные понятия и механизмы, формально не регламентированные ЯП, но традиционно подразумеваемые при его реализации.
- Базовые возможности языка характеризуются его абстрактной машиной.
- Расширенные возможности языка задаются в форме интерпретатора или компилятора.
- Эксплуатационная прагматика характеризуется уровнем изученности решаемых задач и фазами ЖЦП, на которых успешно проявляется ПП, поддерживаемая ЯП,
- При декомпозиции определения языка на фрагменты и подсистемы используется фразеологическая характеристика языка, что позволяет прагматически выбрать критерии выделения компонентов, обладающих общностью для разных языков.

Определитель парадигмы языка программирования содержит следующие процедуры:

- Разложение языка на фрагменты по уровням/концентрам и слоям с целью выделения базовых средств языка и его реализационного ядра – семантический базис.
- Декомпозиция семантического базиса языка на основные семантические системы с минимизацией их сложности и, возможно, их описание относительно концептуальных языков.
- Определение АМ языка и интерпретатора, формально достаточного для построения расширений, эквивалентных исходному языку – нормализованное определение.
- Сравнение полученного определения с описаниями известных парадигм и концептуальных языков.
- Обоснование выводов относительно парадигмы исследуемого языка.
- Фразеологический словарь ПП, используемый при определении ЯП.

- Определение уровня языка и его ниши в жизненном цикле программ и деятельности программистов (цели и задачи), базовых языков, использованных при его создании и реализации, как основы для рекомендаций по выбору и применению ЯП и его СП.

Примеры представления результатов парадигматического анализа языков программирования можно выразить в табличной форме.

Таблица 21

Парадигматическая характеристика Pure Lisp

Параметр	*Конкретика*
Эксплуатационная прагматика ЯП[13]	Язык учебного назначения, созданный специально для изучения методов функционального программирования на языке Lisp, успешно применяется при решении задач с исследовательским компонентом, требующих быстрой отладки.
Особенности системы понятий	Сведены все понятия к разным категориям понятия «функция», унифицированы представления функций и значений, выполнение программы рассматривается как отображение списка аргументов в результат.
Перечень понятий, распознаваемых на уровне абстрактного синтаксиса	Символьное выражение (S-выражение), атом, вычислимая форма, переменная, константа, ветвление, элементарные функции, определение функции, именование функции, применение функции, аргумент функции, значение.
Базовые средства ЯП	NIL, CONS, CAR, CDR, ATOM, EQ, QUOTE, EVAL, COND, LAMBDA, DEFUN.
Семантические расширения[14]	Работа с числами и строками рассматривается как вспомогательная семантика. Ввод и вывод данных считаются псевдо-функциями, обслуживающими отладку. Отдельное расширение языка поддерживает функции с пост-вычислением аргументов – специальные функции.
Регистры абстрактной машины	Стеки для хранения результатов, локальных значений переменных, вычисляемой формы и дампа, обеспечивающего защиту контекста вычислений (S E C D)
Категории команд абстрактной машины	Засылки в стек, вычисления над стеком, пересылки из стека, ветвления, применение функции, выходы из

[13] Уровень языка, его ниша в полном жизненном цикле программ, соответствующая технология.

[14] Вспомогательные семантики, их описание относительно концептуальных языков.

	ветвлений и функций, восстановление контекста, поддержка рекурсии.
Реализационная прагматика [15].	Списки из бинарных узлов, содержащих пару тэгированных указателей. Тэг показывает тип данного, адресуемого указателем. Автоматизировано освобождение памяти служебной программой «Сборщик мусора».
Парадигматическая специфика [16]	Исторически основополагающий классический язык, поддерживающий функциональное программирование в полном объёме.

Ещё один пример столь же компактной парадигматической характеристики:

Таблица 22

Парадигматическая характеристика учебного концентра языка Pascal, поддерживающего методику структурного программирования

Параметр	*Конкретика*
Эксплуатационная прагматика ЯП	Язык учебного назначения, созданных для обучения студентов методам программирования решений задач, готовых для эффективной реализации при поддержке автоматным моделированием.
Особенности системы понятий	Программа и данных – раздельные сущности. Выполнение программы сводится к шагам изменения состояний памяти, хранящей данные. Используемые в программе идентификаторы подчинены иерархии областей видимости, задаваемых вложенностью определений процедур и функций. Процедуры и типы данных не являются значениями. Возможно конструирование новых типов данных и приведение типа данных к заданному.
Перечень понятий, распознаваемых на уровне абстрактного синтаксиса.	Скаляр, вектор, значение, ключевые слова, константа, переменная, тип данных, операция, выражение, операнд, элемент вектора, сравнение, действие, последовательность действий, ветвление, цикл, определение процедуры/функции, вызов процедуры/функции.
Базовые средства ЯП.	TRUE, FALSE, перечислимые значения, :=, a[i], IF, WHILE, PROCEDURE, st1 ; st2 .

[15] Структуры данных и традиционно подразумеваемые механизмы реализации

[16] Роль языка в формировании поддерживаемой им парадигмы и/или перечень поддерживаемых парадигм.

Семантические расширения	Разные виды чисел, записи, множества, указатели, метки, передачи управления, функции, переключатели, конструирование типов данных, ввод-вывод, файлы, строки, библиотеки.
Регистры абстрактной машины	Стеки промежуточных результатов, локальных переменных и параметров, выполняемой процедуры и вектор памяти (S E C M).
Категории команд абстрактной машины	Засылки в стек, вычисления над стеком и памятью, пересылки из стека и локальных переменных, ветвления, передачи управления, вызовы процедур.
Реализационная прагматика	Память распределяется статически по блокам заданного размера, обработка векторов использует вычисление смещений от базового адреса и подразумевает контроль границ отведенной под вектор памяти, при необходимости привлекаются библиотеки, поддерживающие ввод-вывод, работу с файлами и динамической памятью.
Парадигматическая специфика	Прецедент строго определения языка программирования, обладающего не слишком высокой сложностью в реализации, поддерживающего методику результативного структурного программирования в рамках парадигмы процедурно-императивного программирования. Может выполнять роль эталонного монопарадигматического языка при сравнительном анализе языков и определении их парадигматической характеристикию

ЗАКЛЮЧЕНИЕ

Исследования парадигматической характеристики языков и систем программирования сложились в 1970-е годы в сфере влияния академика А.П. Ершова, в рамках дискуссий на фоне конференций и семинаров при обсуждении разноплановых экспериментальных проектов, нацеленных на развитие средств, методов и технологий эффективной и надежной реализации программистского инструментария. В те годы число рассматриваемых языков едва превышало пару-тройку сотен, что упрощало выделение ключевых идей в языкотворчестве. В настоящее время актуальность проблемы классификации языков программирования и вообще компьютерных языков и информационных систем существенно возросла [12].

Средства и методы программирования складывались на фоне быстрого расширения сферы применения компьютерных технологий и стремительного взлёта эффективности элементной базы. Темп развития потребности в программировании намного опередил динамику кристаллизации используемых средств и понятий, что вызвало грандиозный разнобой в программистском инструментарии и значительные разночтения в программистском лексиконе.

Наряду с расслоением парадигм программирования в зависимости от глубины и общности технических решений по организации процессов обработки данных происходит их интеграция в рамках новых языков программирования, всё более полно поддерживающих жизненный цикл программ. В результате обнаружились явные сложности классификации языков программирования и определения их принадлежности конкретной парадигме программирования.

Рассмотрение технических особенностей языков и систем программирования через их парадигматическую характеристику может дать достаточно лаконичные формы их определения относительно ряда простых концептуальных языков.

В данном препринте рассмотрены вопросы проявления и поддержки парадигм программирования в процессах разработки программ, положенные в основу парадигматической характеристики языков и систем программирования, а также выработки рекомендаций по выбору парадигмы в зависимости от особенностей постановки задач и условий применения их программируемых решений.

Для иллюстрации подхода к парадигматической характеристике ЯП отдельно будет приведен аналитический обзор парадигм программирования, рассмотрены особенности применения языков программирования, отражающие методы их реализации и класс решаемых на их основе задач. Отмечены ключевые моменты развития парадигм программирования, анализируются закономерности освоения новых методов обработки данных посредством реализации систем программирования и других информационных систем. Определен ряд опорных языков для сравнительного анализа языков программирования при определении их соответствия конкретным парадигмам.

СПИСОК ЛИТЕРУРЫ

1. Айлиф Дж. Принципы построения базовой машины. – М.: Мир, 1973. – 119 с.
2. Вирт Н. От Модулы к Оберону. // Системная информатика. Вып 1. Проблемы современнного программирования. - Новосибирск: Наука. Сиб. отд-е, 1991. – С. 63-75
3. Городняя Л.В. Функциональный подход к описанию парадигм программирования. // Препринт № 152. Надзаг.: ИСИ СО РАН, Новосибирск, 2009. – С. 66.
4. Грогоно П. Программирование на языке Паскаль. – М.: Мир, 1982. – 382 с.
5. Дейкстра Э. Дисциплина программирования. – М.: Мир, 1978. – 275 с.
6. Евстигнеев В.А., Касьянов В.Н. Графы в программировании: обработка, визуализация и применение. – СПб: ЕХП-Петербург, 2003, – 1104 с.
7. Ершов А.П. Смешанные вычисления: потенциальные приложения и проблемы исследования. // Тезисы докладов и сообщений. Всесоюзная конференция "Методы математической логики в проблемах искусственного интеллекта и систематическое программирование", Ч.2. – Вильнюс, 1980. – С. 26-55
8. Захаров Л.А., Покровский С.Б., Степанов Г.Г., Тен С.В. Многоязыковая транслирующая система. – Новосибирск, 1987. – 151 с.
9. Лавров С.С. Методы задания семантики языков программирования // Программирование. N 6, 1978. – С. 3-10.
10. Лавров С.С., Городняя Л.В. Функциональное программирование. Интерпретатор языка Лисп. // Компьютерные инструменты в образовании. СПб, 2002. №5.
11. Леман М.М. Программы, жизненные циклы и законы эволюции программного обеспечения. – М.: Мир, ТИИЭР, 1980. Т. 68-9. – С. 26-45
12. Марчук А.Г., Городняя Л.В., Мурзин Ф.А., Шилов Н.В. Классификация компьютерных языков: состояние, проблемы, перспективы // Труды международной конф. “Космос, астрономия и программирование" (Лавровские чтения). СПбГУ. – СПб, 2008. – С. 15-22.
13. Непейвода Н.Н. Стили и методы программирования.—— М.: Интернет-университет информационных технологий. -: http://www.intuit.ru/department/se/progstyles/ , 2004 (доступно 2.11.2014)
14. Поттосин И.В. Система СОКРАТ: Окружение программирования для встроенных систем. – Новосибирск, 1992. – 20 с. (Препр./РАН. Сиб. отд-ние. ИСИ; N11)
15. Пратт Т., Зелковиц М. Языки программирования. Разработка и реализация. / Под общей редакцией А.Матросова. – СПб.: Питер, 2002. – 688 с.
16. Савенков К. Верификация программ на моделях. / Курс ВМК МГУ им. М.В. Ломоносова. http://savenkov.lvk.cs.msu.su/mc/lect02.pdf
17. Серебрянников В.А. Лекции по конструированию компиляторов. – М.: ВЦ РАН, 1994. – 175 с.

18. Стивен Р. Палмер, Джон М.Фелсинг. Практическое руководство по функционально-ориентированной разработке ПО. – М.: Вильямс, 2002. – 299 с.
19. Уоткинс Д., Хаммонд М., Эйбрамз Б. – Программирвоание на платформе .Net. – М. Вильямс, 2003. – С. 367.
20. Фуксман А.П. Технические аспекты создания программных систем. – М.: Статистика, 1979. – 180 с.
21. Хендерсон П. Функциональное программирование. – М.: Мир, 1983. – 349 с.
22. Черноножкин С.К. Методы тестирования программ. /Новосибирск: НГУ, 2004. 166 с.
23. Backus J. Can programming be liberated from the von Neumann style? A functional stile and its algebra of programs // Comm. ACM. V. 21 N. 8. 1978. P. 613-641.
24. Floyd, R. W. (1979). The paradigms of Programming .-- Communications of the ACM 22 (8): 455
25. .Knoop J. Compiler Construction / 20th International Conference, CC 2011Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011 Saarbrücken, Germany, March 26 ––April 3, 2011 // Lecture Notes in Computer Science. Springer V2011. V. 6601. 330 p.
26. Lucas P., Lauer P., Stigleitner H. Method and Notation for the Formal Definition of Programming Languages. IBM Laboratory. Vienna, TR 25.087, 1968.
27. McCarthy J. LISP 1.5 Programming Manual. – The MIT Press, Cambridge, 1963. – 106 p.
28. Schwartz, Jacob T., Set Theory as a Language for Program Specification and Programming / Courant Institute of Mathematical Sciences. New York University, 1970.
29. Weinberg G.M. The Psychology of Computer Programming. – New York: Van Norstand Reinhold Comp., 1971.
30. http://www.literateprogramming.com Сайт с материалами по грамотному программированию. Кнут Д. Literate programming. (доступно 2.11.2014)
31. http://haskell.org/ – Материалы по языку Haskell. (доступно 2.11.2014)
32. http://refal.org/rf5_frm.htm . Турчин В.Ф. Материалы по языку Рефал. (доступно 2.11.2014)

Приложение

ТЕРМИНЫ И ОБОЗНАЧЕНИЯ

Ассоциативный список – список пар, предназначенный для хранения соответствия между именами и их определениями (значения переменных, констант, определения функций).

Дамп – резервная память для хранения промежуточных результатов, которые могут понадобиться при дальнейших вычислениях.

Мера организованности программы – зависимость объёма отладки модифицируемой программы от объёма вносимых изменений.

Показатель отлаженности программы – частота обнаружения дефектов в программе или длина интервала между внесением исправлений в программу.

Ранг работоспособности программы – полнота множества возможных данных, на которые программа реагирует разумно.

Реализационное замыкание ЯП – расширение определения ЯП, создаваемое для его эффективной реализации.

Свободные переменные – получают значение или определение вне задаваемого фрагмента.

Семантическая декомпозиция – разложение определения на части, смысл каждой из которых может быть выражен на естественном языке.

Семантический спуск определения ЯП – исключение из определения ЯП компонентов, сводимых у более простым или общим средствам, что формально не влияет на полноту ЯП.

Список свободной памяти – структура данных, обеспечивающая динамический доступ к памяти.

Степень изученности задачи – оценка готовности к программированию постановки задачи и методов её решения.

Универсальная семантическая функция ЯП – функция вычисления результата любой правильно представленной на ЯП функции от допустимых данных.

Уровень абстрагирования понятий – характеристика независимости понятия от малосущественных свойств конкретных примеров.

Хэш-функции – методика хранения динамически изменяемого конечного множества из элементов бесконечного множества.

АББРЕВИАТУРЫ

Обозначение	*Расшифровка*
АМ	Абстрактная машина
АС	Абстрактный синтаксис
БНФ	Формы Бэкуса-Наура
БС	Базовые средства
ВС	Вспомогательная семантика
ЖЦП	Жизненный цикл программ
ИПП	Императивно-процедурное программирование
ИС	Информационная система
ИТ	Информационные технологии
КМ	Конкретная машина
КС	Конкретный синтаксис
КЯ	Концептуальные языки программирования
ЛП	Логическое программирование
ООП	Объектно-ориентированное программирование
ОС	Операционная семантика
ПЖЦП	Полный жизненный цикл программ
ПИП	Процедурно-императивное программирование
ПП	Парадигма программирования
РБНФ	Расширенные формы Бэкуса-Наура
РП	Реализационная прагматика
СД	Структуры данных
СП	Система программирования
СПП	Стандартное прикладное программирование
ТА	Таблица атомов
ТД	Типы данных
ТИ	Таблица идентификаторов/имён
ТП	Технология программирования
УФ	Универсальная функция языка программирования
ФП	Функциональное программирование
ЭП	Эксплуатационная прагматика
ЯП	Язык программирования
ЯСП	Язык и система программирования
AML	Абстрактная машина языка Lisp

AMP	Абстрактная машина языка Pascal
FDD	Функционально-ориентированное проектирование
SECD	Абстрактная машина языка Lisp
SECM	Абстрактная машина языка Pascal
UML	Универсальный язык моделирования UML
XP	Экстремальное программирование

ОБОЗНАЧЕНИЯ

(X . Y) – работает как (cons X Y) – X становится «головой» списка Y.

(x . l) – это значит, что первый элемент списка – x, а остальные находятся в списке l.

(x y . l) – первый элемент списка – x, второй элемент списка – y, остальные находятся в списке l и т.д.

([XL . YL] . AL) – работает как (pairlis XL YL AL) – функция аргументов XL,YL, AL строит список пар-консолидаций соответствующих элементов из списков XL, YL и присоединяет их к списку AL. Полученный список пар, похожий на таблицу с двумя столбцами, называется ассоциативным списком или таблицей атомов. Такой список может использоваться для связывания имен переменных и функций при организации вычислений интерпретатором.

(X | Y) – работает как (append X Y) – сцепляет списки в один общий список.

AL[X] – работает как (assoc X AL) – функция двух аргументов, X и AL. Если AL – таблица атомов, подобная тому, что формирует функция pairlis, то assoc выбирает из него первую пару, начинающуюся с X. Таким образом, это функция поиска определения или значения в таблице атомов.

[x] – содержимое памяти по адресу x

e[n] – содержимое n-го элемента контекста

A(Pr) – число аргументов процедуры Pr

L(Pr) – число локальных переменных процедуры Pr

@F – адрес подпрограммы, выполняющей функцию F.

@c – адрес позиции «c» в программе

_ – Произвольное значение (_ подчерк)

СОДЕРЖАНИЕ

Часть 1. Сравнение парадигм программирования

[17] Части 2-5 – отдельные препринты.

Л.В. Городняя
ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Часть 1.
Сравнение парадигм программирования

Препринт
172

Рукопись поступила в редакцию 10.10.2014
Редактор Т. М. Бульонкова
Рецензент Ф.А. Мурзин

Подписано в печать 18.11.2014
Формат бумаги 60 × 84 1/16 Объем 6.6 уч.-изд.л., 7.25 п.л.
Тираж 60 экз.

Центр оперативной печати «Оригинал 2»
г.Бердск, ул. О. Кошевого, 6, оф. 2, тел. (383-41) 2-12-42