Городняя Л. В., Андреева Т. А.

ПАРАДИГМА ПРОГРАММИРОВАНИЯ
Учебное пособие

2025

УДК 004.43 (042.4)
ББК 32973-018
ISBN

Рецензент
науч. сотр. лаб. моделирования сложных систем ИСИ СО РАН
канд. физ.-мат. наук *Д. С. Мигинский*

Аннотация

Учебное пособие посвящено проблеме системной информатики, связанной с исследованием и разработкой методов анализа парадигм и языков программирования. Материал пособия сформирован на базе многолетнего опыта преподавания курса «Парадигмы программирования» на базе Интернет-Университета Информационных технологий, а также спецкурсов ФИТ и ММФ НГУ. Показаны особенности базовых и фундаментальных парадигм, описаны подходы к классификации языков и систем программирования, поддерживающих конкретные парадигмы. Содержание представляет интерес для тех, кто интересуется проблемами современной информатики, системного программирования и информационных технологий.

Введение

Учебное пособие расширяет и упорядочивает материал, представленный в ранее изданных пособиях[1] и монографиях[2], а также в дистанционных курсах, поставленных на базе Интернет-Университета[3]. Изменения сложились в практике многолетнего преподавания на базе кафедр «Программирование» ММФ и «Системы информатики» ФИТ НГУ, а также результатов исследования языков и парадигм программирования (ПП), традиционно выполняемых в лаборатории информационных систем ИСИ СО РАН им. А. П. Ершова в рамках тематики, связанной с разработкой средств и методов преподавания программирования[4]. Дополнения затрагивают постановку лабораторных работ. Последовательность обучения приведена в соответствие с международными рекомендациями обучения программированию в университетах.[5] При подготовке материала учтены вопросы студентов, проходивших курс «Парадигмы программирования» в 2022 году на базе кафедр «Программирование» ММФ и «Системы информатики» ФИТ НГУ.

Изложение начинается с общего представления о разнообразии задач программирования и решений, принимаемых при программировании (лекция 1). Затем следует краткий экскурс в методы определения языков программирования (ЯП), а также вопросы построения систем программирования (СП) как основных средств поддержки ПП (лекция 2). Рассматриваются особенности полной ПП и проявления отдельных ПП в языках программирования разного уровня, а также вопросы классификации ПП. В лекциях 3 —9 описаны конкретные средства и методы реализации языков низкого уровня (ЯНУ), языков высокого уровня (ЯВУ), параллельных вычислений (ПВ) и языков сверх высокого уровня (ЯСВУ). Приведены примеры фундаментальных парадигм, имеющих образовательное значение для подготовки высококвалифицированных специалистов. По ходу курса описана динамика кристаллизации парадигм[6] и намечена техника парадигмальной характеристики языков и систем программирования (ЯиСП) в процессе их анализа и сравнения. В заключении подводятся выводы относительно идентификации парадигм программирования.

В центре внимания находятся исторически значимые и концептуальные языки программирования (КЯ), в которых видны ключевые идеи и практические следствия их реализации в системе программирования (СП), включая средства поддержки полного жизненного цикла программ (ПЖЦП). Любые средства и методы программирования могут быть применены или смоделированы в любой парадигме, на любом ЯП, разница в продуктивности программирования. Цель курса — наметить подходы к анализу и сравнению языков, парадигм и систем программирования на основе ясно удостоверяемых условий, допускающих проверку.

Благодарность. Авторы выражают благодарность студентам ФИТ и ММФ НГУ за обратную связь в виде вопросов, показавших позиции и границы необходимых улучшений текста — А. Базарову, Д. Бакуневичу, И. Бакшееву, А. Баранову, Е. Блинову, К. Галишевой, А. Дорну, М.

---

1 Городняя, Л. В. Парадигма программирования : учебное пособие для вузов / Л. В. Городняя. — 2-е изд., стер. — Санкт-Петербург : Лань, 2021. — 232 с. — ISBN 978-5-8114-6680-1

2 Городняя Л.В. Парадигмы программирования: анализ и сравнение. Сиб. Отделение Рос. Акад. наук, Ин-т систем информатики им. А.П. Ершова. — Новосибирск: Изд-во СО РАН, 2017. — 232 с.

3 Городняя Л.В. Основы функционального программирования. — М.: Интернет-Университет Информационных технологий. − http://www.intuit.ru, 2004. − 272 с.
Городняя Л.В. Парадигмы программирования (поддержка Академии Microsoft) https://intuit.ru/studies/courses/1109/204/info

4 Городняя Л.В. Парадигмы параллельного программирования в университетских образовательных программах и специализации //Всероссийская научная конференция "Научный сервис в сети Интернет: решение больших задач − Новороссийск-Москва, 2008. − с. 180-184.
Городняя Л.В. Парадигмы программирования в профессиональной подготовке информатиков. // Проблемы специализированного образования. − Новосибирск, 1998. − с. 115-124

5 Computing Curricula 2001. Computer Science Volume.

6 **Проявление**, постепенное наращивание — это процесс перехода от нечётких представлений к ясному, структурированному пониманию

Карпову, М. Кобцеву, М. Копылову, Д. Королеву, Н. Косареву, М. Куликову, Ю. Нуштаеву, М. Пашенцеву, В. Ремезову, Д. Савельеву, И. Сорокину, Д. Суворову, П. Щербаковой.

Часть 1 Общее представление

«Движенья нет, сказал мудрец брадатый»
А.С.Пушкин «Движенье»

Лекция 1. Предварительные сведения

Альтернативные, отличающиеся от других, подходы к обработке информации, сложившиеся при создании и применении языков программирования (ЯП) и систем программирования (СП), принято называть *парадигмами программирования* (ПП). Изучение и чёткая классификация уже сложившихся и новых ПП призваны помочь как обоснованному выбору ЯП и СП, так и созданию новых компьютерных языков при формировании программных проектов, а также совершенствованию информационных технологий (ИТ).

Языки и системы программирования (ЯиСП) представляют знание о потенциале ИТ. В момент своего создания ЯП отражает некий прогноз относительно востребованной области приложения ИТ, СП конкретизирует и уточняет такое знание в форме отлаженных программ с комплектами данных для них и прецедентами успешного их применения в компьютерном эксперименте. Единственно достоверный документ о свойствах ЯП — это работающая ЯиСП с комплектом отлаженных программ. Интересно отметить, что рейтинг популярности ЯП отличается от рейтинга ЯП, использованных в успешно завершенных проектах.

Мировая практика насчитывает уже десятки тысяч ЯП, число парадигм — несколько десятков. Долгоживущие, учебные и новые ЯП преимущественно мультипарадигмальны. Понимание «правильности» программ по разному рассматривается в теории и практике. Оценка продуктивности программирования обычно содержит систематическую ошибку в 2-8 раз.

Данный курс даёт походы для ответов на ряд вопросов:
- Почему стремительно растёт число ЯП?
- Какие параметры позволяют определять какую ПП поддерживает ЯП?
- Что отличает «парадигму программирования» от технологии, методики, стиля, техники, метода и др. близких понятий?
- Как прогнозировать продуктивность разработки программы на базе определённой ЯиСП?
- Что позволяет обосновывать выбор ЯиСП для решения конкретной задачи?
- Насколько выбор ЯиСП и ПП определяет производительность программ?

Среди множества опубликованных определений термина «парадигма программирования» самое конструктивное принадлежит Питеру Вегнеру (Peter Wegner), автору книги по языку АДА[7], переведённой в своё время на русский язык. В работе «Concepts and paradigms of object-oriented programming» П. Вегнер предлагает парадигмы программирования определять как «правила классификации языков программирования в соответствии с некоторыми **условиями, которые могут быть проверены**». В этой его статье выполнен анализ ООП и сравнение его с другими парадигмами, преимущественно с ФП. Остаётся лишь выбрать такие условия, помогающие отвечать на следующие вопросы:
- Какие условия или параметры позволяют различать парадигмы?
- Что позволяет проверять, что парадигма имеет поддержку программистов?[8]
- Какова роль парадигм в определениях мультипарадигмальных ЯП?
- Насколько удаётся измерять зависимость успеха программирования от выбора парадигмы (продуктивность и производительность)?

---

[7] Вегнер П. Программирование на языке Ада. — М.: «Мир», 1983. — С. 240.

[8] *Вопрос:* В чём разница между парадигмой программирования и образом мышления программиста?
Ответ: Парадигма программирования — это образ мышления, поддержанный языком программирования и корпусом отлаженных программ.

1.0. Появление термина

Именно парадигмам программирования Роберт Флойд посвятил свою Тьюринговскую лекцию, в которой обратил внимание на значимость этого понятия в контексте проблемы обучения программистов[9]. Авторитетный ученый, заложивший основы теорий анализа и верификации программ, автор ряда 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. На своем программистском опыте Р. Флойд сделал интересное наблюдение: искусство программирования включает в себя **расширение репертуара** используемых парадигм. Его внимание привлекла задача подготовки рекурсивных сопрограмм[10], удобно формулируемых в терминах недетерминизма, неэффективность которого практично преодолевается макротехникой. Р. Флойд высоко оценил созданные в MIT[11] многочисленные примеры высокого потенциала программирования на универсальном языке Lisp[12], показывающие путь от простейших списков до универсальных структур данных (СД), способствующих манипулированию программами как данными. Р. Флойд выразил уверенность, что можно наладить ясное обучение ряду семантических методов для всех уровней программного проекта так, чтобы у обученных студентов хватало головы на решение полного спектра проблем от упрощённой учебной постановки задачи до непрерывно расширяющегося класса практических задач.

Такая уверенность подтверждается историей языка Lisp, на котором выполнены первые эксперименты и созданы первые системы по логическому выводу, математическим системам, верификации и др. Создано много больших программ по обработке химической и биологической информации, медицинской диагностике, лингвистике, работе с естественными языками и пр. Всё это узко профессиональные линии, это не массовое ПО. В 1980-ых годов именно на языке Lisp были созданы наиболее сложные в мировой практике программы. Самой живучей оказалась система инженерного проектирования AutoCAD. Для неё создали свой диалект языка Lisp — AutoLisp. По её образу у нас в ИСИ СО РАН когда-то сделали систему bCAD, её применяли в НЭТИ и архитектурном институте.

Обработка списков в первых СП для языка Lisp выполнялась медленнее, чем обработка векторов на языках C и Fortran, хотя давно уже, примерно к середине 1980-ых, созданы версии языка Lisp с быстрой обработкой векторов[13]. Многих смущает немонотонность времени счёта из-за сборки мусора, но это теперь уже практика, принятая во многих новых ЯП, особенно в сетях, где имеются и другие причины немонотонности.

Многие современные инструменты появились сначала как прототипы, созданные именно на языке Lisp. Это и гипертексты[14], и онтологии[15], и базы знаний[16], и программы-собеседники[17], нейросети и компьютерные игры, включая управление ОС[18]. Lisp прекрасно работает как скриптовый язык[19] вместо bash для Linux. База данных в нём имеется

---

[9] file:///C:/Users/ak/Downloads/1283920.1283934.pdf

[10] При подготовке сопрограмм на Фортране, обладающем поддержкой раздельной компиляции, заранее не известно в каком порядке будут происходить переходы от одной сопрограммы к другой, это может зависеть от входных данных. Поэтому их можно рассматривать как вариант недетерминизма. Для рекурсии это даёт возможность повторных прогонов сопрограммы, что не вполне решает проблемы реализации рекурсивных вызовов, но всё-таки позволяет их делать.

[11] Massachusetts Institute of Technology

[12] https://psachin.gitlab.io/lisp-java-notes.html — сравнение C++, Java, Lisp
и статья https://renato.athaydes.com/posts/revisitё6+ing-prechelt-paper-comparing-languages.html

[13] https://www.cons.org/cmucl/ — Язык программирования Cmucl

[14] *Хювёнен Э., Сеппянен Й.* Мир Лиспа. В 2-х т. / Пер. с финск.. — М.: Мир, 1990

[15] https://www.cyberforum.ru/lisp/thread2365760.html — Работа с фреймами

[16] https://therapy.school/feed/istoriya-meditsiny-iskusstvennyy-intellekt/

[17] https://en.wikipedia.org/wiki/ELIZA

[18] https://users.mccme.ru/raskin/litm-slides-2021.pdf

[19] http://rus-linux.net/MyLDP/algol/LISP/lisp01.html

встроенная — это списки свойств атомов. Имеются реализации фреймворков[20] и серверов[21], базы данных[22] и другое.

Полная парадигма программирования представляет собой расширяющийся ряд отдельных ПП, поддерживающих продуктивность программирования решений конкретных классов задач. Проявляется парадигма программирования как способ мышления, влияющий на технологичный инструмент грамматического описания фактов, событий, явлений и процессов, возможно, не существующих одновременно, но интуитивно объединяемых в общее понятие в рамках деятельности по созданию свободных от ошибок программ решения нужных задач. В Приложении 1 можно видеть хронологию появления большого числа ПП.

Существуют разные подходы к классификации ПП. Чаще всего выделяют базовые, основные и фундаментальные ПП без претензий на ортогональность. Базовые парадигмы имеют корни в компьютерных архитектурах, поэтому при их характеристике учитывают особенности набора регистров, используемых при определении операционной семантики ЯП. Фундаментальные парадигмы ориентированы на научные экспериментально-исследовательские и образовательные проблемы стратегического характера, поэтому при их характеристике учитывают выразительную силу, точнее мощность пространства вычислительных процессов, допускаемых при реализации программ. Происходит переход к многопроцессорным комплексам и обработке определений языков программирования как типов данных. Основные парадигмы включены в технологии производственного программирования, поэтому для них важен вопрос квалификации и способностей программистов, способных надёжно обеспечивать продуктивность программирования и производительность программ. Зависимость этих факторов от квалификации и способностей программиста исследовалась в фирме IBM уже в начале 1970-ых годов. Измеримые результаты показали разброс 1 к 30 по разным параметрам без чёткой зависимости от опыта и образования. На задачах повышенной сложности разброс примерно 1 к 3, причём заметно в пользу опыта, возраста и образования[23].

В качестве базовых парадигм программирования обычно признают ИП, ФП, ЛП и ООП. Фундаментальные парадигмы, направленные на разрешение кризиса технологий программирования, дополняют множество базовых парадигм мета-программированием и ПВ, отмечая, что ЛП и ООП требуют знаний и навыков в конкретных областях программирования. Основные парадигмы занимают определенную нишу в технологиях программирования, дополняя множество фундаментальных парадигм парадигмами событийно-ориентированного, параллельного и распределенного программирования[24], иногда декларативного, часто встречающихся в современной практике.

Новые парадигмы появляются при проявлении трудно решаемых проблем и различаются приоритетами принятия решений в процессе подготовки и отладки программ. Игорь Васильевич Поттосин[25] говаривал: «Программист — смелый человек. Когда математик докажет, что некая проблема неразрешима, программист берёт и решает эту проблему в пределах, необходимых для практики». Результатом в программировании является работоспособная программа. Так программирование реагируют на расширение пространства решаемых задач, влекущее пополнение ряда ПП, повышающих продуктивность программирования, что и объясняет феномен расширения репертуара парадигм, замеченный Р. Флойдом.

---

20 https://lispcookbook.github.io/cl-cookbook/web.html, https://www.linuxlinks.com/free-open-source-lisp-web-frameworks/, https://www.cliki.net/web%20framework

21 https://habr.com/ru/post/131586/, https://habr.com/ru/post/131780/, https://www.opennet.ru/opennews/art.shtml?num=21893, https://courses.cs.northwestern.edu/325/readings/web-servers.php

22 https://thecode.media/lisp/, https://habr.com/ru/company/sberbank/blog/655509/

23 Weinberg G.M. The Psychology of Computer Programming.- New York: Van Nostrand Comp., 1971. - 279 p.

24 Основным парадигмам посвящена вторая часть курса.

25 https://www.iis.nsk.su/person/ivp, https://www.nsu.ru/n/media/events/nauka/xxiii-otkrytaya-vsesibirskaya-olimpiada-po-programmirovaniyu-im-i-v-pottosina/

Обычно классификацию ПП начинают с различия между линиями императивного, декларативного и стратегического программирования, определяемыми по важности ответов на вопросы «Что?», «Как?» и «Кто?». Императивное сконцентрировано на вопросе «Как?», декларативное на вопросе «Что?», а стратегическое на вопросе «Кто?». Более детально:

- **Что** следует получить как результат решения задачи?
- **Как** можно выполнить решение задачи?
- **Кто** сумеет запрограммировать решение задачи на компьютере?

В этом отношении императивно-процедурное программирование (ИП) в качестве ответа на важнейший вопрос «Как?» выбирает реализацию алгоритма над машинными словами конечного размера в общей памяти. От программиста требуется способность видеть цель процесса на компьютере, добиваться его гомоиконности алгоритму. Это значит, что из ответа на важнейший вопрос «Как?» следует ответ на не столь важный вопрос «Что?», в любом случае будет решена какая-нибудь полезная задача. Функциональное программирование отвечает на важный вопрос «Что?» представлением компьютерных программ символьной обработки, требуя от программиста способности к абстрагированию и полноте решения задачи, оставляя ответ на менее важный вопрос «Как?» открытым, допускающим любые решения, лишь бы был получен нужный результат. Логическое программирование, отвечая на вопрос «Что?», довольствуется возможностью получения демонстрируемого результата из коллекции имеющихся фактов, предполагая способность программиста доказать его правильность. Ответ на вопрос «Как?» допускает бесхитростный перебор известных вариантов в произвольном порядке. ООП отвечает на вопрос «Как?» в условиях существования готовых программ решений задач, претерпевающих развитие, частично изменяющее ответ на вопрос «Что?». От программиста требуется способность сохранять достигнутую правильность, наследовать опыт программирования, уточняя его по отношению к изменившимся аспектам. Линия стратегического программирования, связанная с вопросом «Кто?», не претендует на быстрые ответы, оно требует от программиста способности обучаться и прогнозировать развитие постановок задач (эволюция вопроса «Что?») и областей приложения, а также методов разработки компьютерных программ (усовершенствование ответа на вопроса «Как?»). Таким образом, одна ПП может одновременно быть и базовой, и фундаментальной, и основной.

1.1. Иллюстрация понятийной сложности

Результаты анализа большого числа ПП и подходов к их классификации показывают, что наиболее чёткое их различие связано с пониманием важности, выраженной как упорядочение приоритетов принятия решений в процессе подготовки и отладки программ с учётом прогноза развития постановки решаемой задачи и условий применения разрабатываемой программы в рамках общепринятой системы понятий. Это несколько отличается от популярной классификации парадигм по изобразительным средствам, моделям и механизмам их поддержки в ЯиСП[26].

Для зримой оценки сложности систем понятий, используемых при принятии решений, рассмотрим аналогии с разными сферам деятельности и проиллюстрируем оценку их сложности в виде кубиков Рубика разного размера.

Существует качественное различие между решением простых задач на небольшом числе примеров, представимых легко обозримыми данными, и решением сложных задач, требующих анализа большого разнообразия редко встречающихся ситуаций и трудно обозримых данных, осложнённых повышенными требованиями к правильности, надёжности и другим критериям:

Легко — зависимость от одного параметра.

Часто — взаимодействие двух независимо изменяющихся параметров.

Редко и *трудно* — взаимодействие большего числа параметров.

---

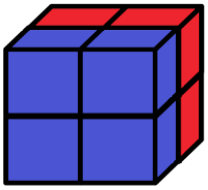
[26] https://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng108.pdf

В обычной жизни чаще всего работает дихотомия принятия легко поляризуемых решений:

Известное — возможное.
Знание — интуиция.
Привычное — удивительное.

Или:

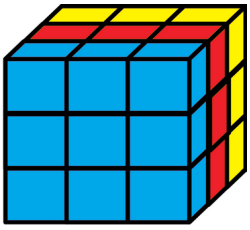
Постоянное — изменчивое.
Данные — действия
Простое — сложное



Получается комбинаторика принятия решений (2*2*2 = 8), что можно сопоставить по сложности понятий с кубиком Рубика из 8 элементов. Обычно сложность ограничена тремя направлениями — длина, высота, ширина.

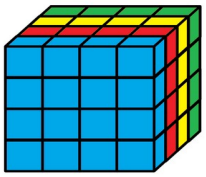
Несколько сложнее выглядят триады формирования знаний в науке и образовании:

Гипотеза — проверка — результат
Теория — практика — эксперимент
Аксиомы — теоремы — доказательства



Здесь комбинаторика принятия решений (3*3*3 = 27), что можно сопоставить по сложности понятий с кубиком Рубика из 27 элементов. Выбор научных и образовательных задач дозируется возможностями регистрируемой фактографии, воспроизводимостью исследовательских экспериментов и понятностью формальных моделей коллегам или учащимся. В науке допустим отрицательный результат.

Достоверность решений в программировании опирается на компьютерный эксперимент, доступный для восприятия широкой аудитории, что приводит к тетрадам работоспособного взаимодействия машинно-ориентированных понятий (количество сочетаний 4*4*4 = 64):

Данные — хранение — контроль — комплексы. Вычисления — память — управление — структуры. Устройства — программы — комплексы — сети.	

Для мультипарадигмальных ЯП парадигмы программирования — аналог проективной геометрии или плоских проекций в инженерии. Инженеров учат строить проекции, по которым можно вообразить и сделать объёмную конструкцию.

Целью программирования является создание правильно работающих программ, но реально основной объём трудозатрат при программировании заключается в поиске и исправлении ошибок, связанных с несоответствием постановки задачи, программы её решения и обрабатываемых данных - специальных тестов. Имеет место ключевая разница с математикой и другими видами интеллектуальной деятельности, задающими культурно-образовательный пресс на безошибочность с игнорированием природы ошибок в любом поведении, особенно в непредсказуемо изменяющейся обстановке. Нет признания, что в задачи программирования входит поиск соответствия осознанных и интуитивных представлений уровня скрытой грамматики деятельности, а не только постановки задачи и создаваемой программы. Кроме того, практические задачи программирования обременены критерием актуальности, непрерывным расширением пространства решаемых задач и сложностью удостоверения правильности программ. Именно эти усложнённости приводят к понятию «парадигма программирования».

В практике применения программ их правильность важнее всех остальных её свойств, включая эффективность, производительность и пр., но при разработке программ и ЯП чаще доминирует продуктивность программирования. Отметим сразу, решения нужных задач актуальны в пределах определённого времени. В социальной системе обычно темп принятия решений важнее, чем их правильность. Это достаточная причина, чтобы продуктивность программирования была ведущим фактором развития средств и методов программирования, включая ЯиСП. Продуктивность программирования имеет три взаимодействующие линии — достижение правильности решений, расширение класса решённых задач, сокращение сроков разработки практичной программ. Правильность программы сводится к её соответствию ожиданиям пользователя. Формальная правильность программы — одно из возможных ожиданий пользователя. Зависимость правильности от ожиданий пользователя определяет власть социальных факторов в оценке результатов программирования.

1.2. Расширение репертуара парадигм

Важнейшая задача любой технологии программирования — обеспечение сходимости процесса отладки программы, т.е. гарантированное 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 — «утончение[27]».

Отмеченное Р. Флойдом расширение репертуара ПП в процессе создания программ связано со структурой расширяемого пространства, в котором принимаются решения при практическом программировании. Постепенно сформировался целый ряд различных моделей полного жизненного цикла программ (ПЖЦП), многие из которых описаны в работах И.Н. Скопина по менеджменту программистских проектов[28].

С физической точки зрения, программа не имеет износа, может жить вечно. Неожиданности в ее функционировании могут быть связаны не только с незамеченными при разработке недочётами, но и с выявлением несоответствия принятых решений новым возможностям оборудования, развитию квалификации программистов и пользователей, улучшению методов программирования. Поэтому программы подвержены моральному устареванию, которое провоцирует продолжение разработки программ с ранее завершенным ПЖЦП, что приводит к изменению перечня этапов разработки программы.

Для больших систем собственно программирование сводится к реализации или выбору 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[29]. Сборка программ из готовых модулей имеет дополнительную нагрузку на ответственное исследование взаимодействия модулей и устройств и их совместного функционирования, особенно если модули не являются универсальными решениями, не достигают полноты решения задачи, ограничены конкретными типам данных.

Известно, что развитие постановок задач по степени изученности не обладает монотонностью трудоёмкости реализации текущей версии программы. А именно, после небольших трудозатрат на разработку макетного образца, где главная цель — показать достоинства идеи, создание экспериментального полигона требует принципиально больших, значительных трудозатрат на полноту исследования и разработки средств и методов решения задачи и определения границ практичности её реализации. Интуитивная оценка трудоёмкости реализации производственной версии в статусе практичного компромисса

---

[27] Термин А.Н. Агафонова, переводчика книги Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Пер. с англ. В 2-х т. Т.1: Синтаксический анализ. Т.1. - 1978. 612 с.

[28] https://intuit.ru/goods_store/ebooks/8180

[29] *Ф. П. Брукс мл.* Как проектируются и создаются программные комплексы. (Серия: «Библиотечка программиста») = The Mythical Man-Month. — М.: «Наука», Главная редакция физико-математической литературы, 1979. — 152 с илл.

существенно меньше, что обычно провоцирует экономию на предварительных исследованиях в надежде угадать границы практичности, но обычно приводит к необходимости многократного версифицирования решения задачи.

Здесь отметим ключевые положения практики применения программ, что можно назвать «эксплуатационная прагматика»:

— критерии качества программ могут изменяться в процессе программирования, что возможно повлечёт существенный пересмотр многих ранее принятых решений (вопрос «КАК?»);

— развитие систем представления знаний сводится к чередованию фаз восходящего и нисходящего уточнения информации, что влияет на выбор технологии программирования (вопрос «ЧТО?»);

— степень изученности решаемых задач — ведущий фактор прогноза трудоёмкости достижения работоспособной программы, пригодной для не авторского применения;

— технологии программирования обеспечивают гарантированное получение результата разработки при условии соответствия технологическим требованиям[30], включая требования к квалификации программистов (вопрос «КТО?»);

— тестирование и отладка программы требуют специального обустройства, средства которого за редким исключением не представлены в ЯП, но могут содержаться в СП[31];

— сходимость ПЖЦП[32] обеспечивается грамотным выбором соответствия степени изученности решаемой задачи и схемы, допускающей при подходящей технологии выполнение отладки программы, удовлетворяющей заданному критерию качества;

— ранг работоспособности и жизнеспособности реализованных решений зависит от полноты и универсальности программных компонентов, созданных при разработке программы.

В практике программирования встречается целый ряд схем ПЖЦП, отличающихся по способу мышления при постановке задач и подходам к методам решения задачи, что даёт ещё один ответ на феномен расширения репертуара парадигм и существования мультпарадигмальных ЯП. Поэтому выбор ПП не только влияет на трудоёмкость или продуктивность программирования, но и в заметной степени определяет работоспособность, производительность и живучесть разрабатываемой программы.

1.3. Подходы к систематизации ЯП

Следует отметить, что за сорок пять лет, прошедших со времени Тьюринговой лекции Р. Флойда, число различных ЯиСП возросло с нескольких сотен до десятков тысяч[33]. При этом число парадигм не столь велико. В разных источниках называют от двадцати до семидесяти парадигм, нередко включая в их перечень отдельные техники и методы, фактически лишь дополняющие или уточняющие базовые парадигмы.

История технологий программирования представляет собой шаги повышения продуктивности программирования ценой снижения производительности программ и требований к квалификации программистов ради расширения сообщества пользователей и класса решаемых задач. Так произошёл переход от прямого кодирования программ к ассемблеру, затем к языкам программирования. Далее от языков низкого уровня к языкам высокого уровня и языкам сверхвысокого уровня. Теперь идёт переход от накопления системных библиотек к конструированию проблемно-ориентированных языков. Параллельно идёт формирование ПП: от императивного стиля к функциональному, затем к

---

[30] Каждая технология (ТП) для гарантии надёжности декларирует нормативы, отражающие важность принимаемых решений. Разные парадигмы отличаются приоритетами выбора решений. Для успешности ТП нужна согласованность приоритетов выбора решений и нормативов ТП, что и зависит от выбора ПП.

[31] Пример — реализация языка Haskell.

[32] Успех ПЖЦП связан со степенью изученности решаемой задачи в соотношении примерно 1 к 8, что трудно улавливать интуитивно.

[33] https://www.levenez.com/lang/ http://ru.progopedia.com/

логическому, от императивного к ООП, затем к аспектно-ориентированному и другие подобные цепочки. Конкретно стоит обратить внимание, что в своё время популярность языка Си была связана с решением проблемы машинно-ориентированного переноса программ. Ядро Си было калькой с DEC-овской архитектуры. Перенос программ через Си означал воспроизведение этой архитектуры на других системах команд, что сопровождалось потерей эффективности, но вознаграждалось темпом освоения новых архитектур и свободой от необходимости их понимать[34]. С тех времён появились другие архитектуры, но LLVM привязан к прежним решениям Си, что на практике никого не смущает[35]. Доделаны лишь небольшие уточнения для многопроцессорных комплексов и сетей. По вопросу о практическом смысле непрерывно растущего числа ЯиСП можно провести аналогию с устойчивым ростом числа моделей автомобилей и велосипедов, хотя непрерывно слышен призыв велосипедов больше не изобретать[36].

Различается отношение к источникам ошибок и мерам их профилактики в разных ПП и разных сферах приложения. 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 отклоняющейся точки. Исправление ошибки — это преобразование определения функции к виду, дающему её уточнённый график. Если достигается состояние, в котором уточняемые сущности соответствуют друг другу, то утверждается, что завершена отладка программы. Иногда вместо дальнейшей отладки программу сопровождают шаблонами сценариев и образцами данных, на которых достигнута и удостоверена правильная работа 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 проектирования, рамками надёжных стандартов, использованием удобно реализуемых моделей и конечно стандартных библиотек. Здесь многое даёт парадигма ФП. 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[37].

Известно, что проверка соответствия новому стандарту компилятора C++ выполняется более чем на 6000 тестов, причём определение стандарта подвержено регулярным уточнениям.

Таким образом, существует ряд ПП, появление которых обусловлено развитием ТП, представленных в форме ЯиСП, 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 программ — ПЖЦП.

Первые парадигмальные противопоставления начались с идеологических баталий между сторонниками функционального и императивного программирования, хотя более ясная и более объективная граница пролегает между процедурным и декларативным программированием. Такая граница легко иллюстрируется на методах решения задач на

---

[34] Теория эволюции гласит, что побеждает не сильнейший, а более приспособленный.

[35] Это происходит из-за бизнес-экономического приоритета захвату ниши перед созданием шедевра, обладающего предельной эффективностью или производительностью.

[36] Все против, но велосипеды становятся лучше. Библиотеки процедур в своё время определили доминирование языка Fortran, поддержка академическим сообществом позволила языку Pascal лет на десять стать производственным языком. Всё это важно, но продолжает изменяться электроника и расширяется сфера потребления. Теперь доминирует создание проблемно-ориентированных ЯП (DSL).

[37] Известен прецедент обнаружения ошибки в Intel-овском компиляторе примерно на 170-ом витке из-за ошибок округления.

графах. Процедурный подход опирается на обход графа в глубину, а декларативный — на обход в ширину. Если императивно-процедурное программирование (ИП) апеллирует к спецификации типов данных[38] (ТД), обеспечивающих возможность статического контроля при компиляции[39] программ, то ФП предпочитает полный динамический контроль любых условий, гарантирующих корректность вычислений. ФП начинается с утверждения, что нет границы или принципиальной разницы между представлением функций и значений — и то, и другое не более, чем данные, что как раз соответствует принципу архитектуры фон Неймана, декларирующего, что программа и данные могут храниться в одинаковой памяти. Уже давно имеются прецеденты эффективной реализации ФП, что как-то не влияет на практику ФП, а как раз показывает, что дело не в эффективности.

Кроме базовых, основных и фундаментальных ПП, бывают специализированные парадигмы для решения узких классов задач, вспомогательные или дополнительные парадигмы для расширения возможностей ведущей парадигмы и комбинированные парадигмы, объединяющие особенности двух достаточно разных парадигм.

Интерпретировать или компилировать программы — этот вопрос не зависит от парадигмы, только от традиции. Любой ЯП полезно и интерпретировать, и компилировать. Интерпретировать для нужд отладки и компилировать для массового распространения готового программного продукта. Что-то одно — это грошовая экономия на разработке СП. Многие традиционно компилируемые ЯП обладают интерпретаторами и наоборот, интерпретируемые ЯП в реализации содержат компиляторы как средство оптимизации для повышения эффективности программ.

1.4. От проблем к парадигмам

Знакомое из философии и лингвистики слово «парадигма» имеет в информатике и программировании узко профессиональный, более широкий смысл. Парадигма программирования как способ мышления диктует исходную концептуальную схему постановки задач и методов их решения, определяет инструмент грамматического описания фактов, событий, явлений и процессов, возможно, не существующих одновременно, но интуитивно объединяемых в общее понятие.

Язык Fortran — ветеран прикладного программирования, постепенно в 1970-е годы стал несколько уступать в этой области языкам Pascal[40] и C, а на суперкомпьютерах — языкам параллельного программирования, таким как функциональный язык параллельного программирования Sisal. Происходит включение средств ФП в языки Java, C#, Scala, Kotlin или средств ИП в языки Clojure, Racket. Стандартные подмножества языков Algol и Pascal, послужившие рабочим материалом для теории программирования, сменились более удобными для экспериментирования аппликативными, чисто функциональными или специальными языками, такими как ML, Miranda, Scheme, Haskell, Real, C-light и другие.

Важно отметить, что ПП по-разному проявляют себя на разных классах задач, и поэтому выбор ПП влияет на продуктивность программирования, успешность технологии программирования, надёжность применения программ, их эффективность и производительность. Зависимость ПП от появления трудно решаемых проблем можно видеть на истории проявления стратегических парадигм программирования в Новосибирском ВЦ СО АН СССР[41] . При систематизации ПП[42] бросается в глаза, что многие

---

[38] Обычно, удобных для распределения памяти компилятором.

[39] Компиляцию можно рассматривать как оптимизирующее преобразование программы, повышающее её производительность в сравнении с интепретацией.

[40] Fortran использовался для решения реальных задач, а Pascal представляет собой учебный язык, превосходящий Fortran в скорости изучения и получения как бы работающего решения несложной задачи, если не гнаться за эффективностью. После обретения популярности в университетской среде Pascal лет 10 был производственным языком, но не потеснил Fortran и Си, хотя обладал весьма эффективной реализацией.

[41] https://www.sorucom.org/articles/materialy-mezhdunarodnoy-konferentsii-sorucom-2020/454

[42] Достаточно конструктивное определение понятию «парадигмы программирования» дал Питер Вегнер, связывая его с условиями, которые могут быть проверены. Р. Флойд ввёл этот термин без формального определения, просто показал на

системообразующие и перспективно-стратегические ПП не просто входили в сферу интересов Андрея Петровича Ершова[43], но в формирование многих из них ему удалось сделать значительный вклад, другие — получили развитие трудами его учеников.

Эксплуатация Альфа-системы показала одну неожиданную проблему — проблему доверия правильности реализации оптимизирующих преобразований. При наличии поддержки тензорных операций в языке Альфа[44], большинство программистов-вычислителей вместо использования выражений вида «A + B» для преобразования двух-трёх мерных матриц предпочитали тщательно выписывать многоуровневые циклы. Самый яркий и дискуссионный проект того времени конечно Ершовский проект много-языковой системы построения трансляторов БЕТА[45].

В конце 1968 года в кабинете А. П. Ершова Джон Маккарти прочел серию лекций, посвященных языку Lisp. Возможность практичной реализации такого языка даже на нашей лучшей ЭВМ БЭСМ-6 вызывала сомнения у иностранных специалистов. Тем не менее, была реализована версия Лисп-системы[46], включавшая в себя русифицированную лексику, универсальную обработку свойств объектов и механизм перераспределения памяти с выгрузкой стека во внешнюю память. Во время очередного визита в Новосибирск Джон Маккарти собственноручно написал тест для проверки функциональной полноты реализационной семантики новосибирской Лисп-системы. Система выдержала эту проверку. Основное применение системы — исследования в области верификации программ, эксперименты по организации недоопределенных вычислений, синтаксически ориентированных преобразований программ и конструированию семантических моделей. Можно сказать, что система поддерживала решение задач уровня исследовательского полигона. Была продемонстрирована целесообразность ограничивать программистский перфекционизм и начинать решение слишком новых программистских задач с создания макетного образца и использования методики раскрутки СП, что теперь называют инкрементальная реализация.

В конце 1960-х годов были достигнуты первые успехи в автоматизации проверки правильности доказательств теорем. Естественно возник соблазн и программу рассматривать как большое доказательство теоремы, утверждающей, что из специально описанных входных данных можно вывести выходные данные, обладающие свойствами, распознаваемыми по тексту программы. Для первых экспериментов в качестве таких, сравнительно удобно удостоверяемых свойств, были выбраны представления типов данных. Экспериментальный полигон обычно был ограничен сравнительно рафинированными подмножествами ЯП, свободными от усложнённых конструкций. Хотя некоторые важные свойства выводить удалось, полученные таким образом системы верификации дискредитировали себя из-за диссонанса между заметной сложностью системы и слишком малой её пропускной способностью. Сложившееся в результате предубеждение не преодолено до сих пор, несмотря на многократные подтверждения пользы от способности систем верификации, особенно поддерживающих проверку на логических моделях, обнаруживать достаточно тонкие ошибки, не заметные при непосредственном тестировании. В основном прижилась практика проверки статических типов данных, считающаяся хорошим тоном при определении новых языков и реализации статических анализаторов в оптимизирующих компиляторах.

Опыт оптимизирующей компиляции, а также появление переносимых компиляторов и средств автоматического конструирования СП методами синтаксической генерации привёл к

примерах.

43 https://ershov.iis.nsk.su/

44 Реализация мультипликативных операций была более близкой к математической традиции в сравнении с аналогичными решениями векторных операций в более поздних языках.

45 Ершов А. П., Касьянов В. Н., Покровский С. Б., Поттосин И. В., Степанов Г. Г. Методика разработки многоязыковых трансляторов на примере системы BETA // Математическая теория и практика систем программного обеспечения. — Новосибирск, ВЦ СО АН СССР, 1982. — С. 64-80.

46 Несколько раньше была выполнена реализация подмножества семантики языка Lisp в ВЦ АН СССР.

парадигме смешанных вычислений или трансформационного программирования, позволяющей повышать продуктивность разработки компиляторов формальными преобразованиями синтаксически определённого интерпретатора. Известно, что трудоёмкость реализации интерпретатора в 3-10 раз меньше, чем разработки компилятора. Вопросы эффективности можно решать уже наработанными методами оптимизирующей компиляции.

Неудовлетворённость освоением потенциала парадигмы трансформационного программирования — мета-вычислений - показана в докладе Андрея Валентиновича Климова, продвигающего подобную идею на материале языка РЕФАЛ. Не исключено, что техника проекций, реализуемая методами смешанных вычислений, ещё приведёт к технологии декомпозиции программ на многократно используемые компоненты для надёжного конструирования сложных систем, отвечающих трудно удостоверяемым критериям качества.

Резкая смена компьютерно-коммуникационных технологий стала серьёзно опережать возможности их рационального освоения опытными программистами. Проявилась проблема эстафеты поколений, успешно решённая системой Новосибирский школ юных программистов (ШЮП), механизм которой включал в себя быстрое выполнение интересных любительских проектов, результаты которых заранее никому не были известны, даже руководителю. В те времена в мировой практике не существовало опыта преподавания программирования в общеобразовательной системе. Были лишь рекомендации ЮНЕСКО слабо развитым странам использовать Logo-миры в факультативной работе с детьми, успешно показавшие себя в развитии так называемых юго-восточных азиатских драконов (Гонконг, Тайвань, Сингапур и др.). Традиции летних школ юных программистов (ЛШЮП) до сих пор поддерживаются ИСИ СО РАН им. А.П.Ершова и являются центром притяжения для любителей детского программирования из разных городов[47].

Среди стратегических ПП особо важна миссия системного программирования[48], заключающаяся в изобретении или открытии переходов от необъятного к обозримому или от неизвестного к понятному. Самые очевидные из таких произошедших переходов представлены в таблице 1.

Таблица 1
От проблем к парадигмам

Годы	*Проблема*	*ПП*	*Примечание/Решение*
1946-1957	Машинная арифметика	ИП	*Затрудняет разница с привычной классической математикой:* *- Существует самое большое целое.* *- Теряется универсальность коммутативности и ассоциативности сложения и умножения.* *- Логика «истина-ложь» заменяется логиками «0-не 0)» и «успех-неожиданность».* *- При вычислении возникают прерывания разной природы, допускающие обработчики для продолжения вычислений.* *- Обработка вещественных чисел усложнена учётом ошибок округления.* *- Системы команд и размеры ячеек на разных компьютерах различны.* За этот период появилось 8 парадигм, ИП — ведущая.
1958-1970	Границы вычислимости:	ФП	*Ограничения машинной арифметики заметно затрудняют достижение границ вычислимости, декларированные машиной Тьюрина и λ-исчислением А.Чёрча.* *- Символьные вычисления позволяют выйти за пределы машинной арифметики, восстанавливают право структур, строк и чисел обладать произвольным размером.* *- Провозглашена универсальность понятия «функция», к которому можно свести любые понятия программирования.*

[47] https://ssyp.ru/ Сайт Летней школы юных программистов, проводимой ИСИ СО РАН.

[48] АПЕ «самоубийственная профессия — чем лучше выполняет свою работу, тем меньше заметен его вклад»

			- Предпочитается древовидное представление структур данных — списки. *- Вектора и ассоциативные таблицы (хэш-таблицы) работают на уровне реализационной прагматики.* *- Изобретен специальный метод работы с неизменяемыми данными («Сборка мусора» - garbage collection) и динамический контроль хода вычислений.* *- Практикуется самоопределение языка программирования, позволяющее при разработке и раскрутке СП приобрести опыт применения нового языка.* За этот период появилось 23 парадигмы, ФП — ведущая.
1970-1977	Преодоление детерминизма	ЛП	*Проявились интересные задачи, решение которых в виде автомата или алгоритма ещё не сложилось.* *- Программирование решений, допускающих недетерминизм в порядке вычислений, недоопределённость и даже невычислимость отдельных значений, мягкость границ точности типов данных — программирование в ограничениях.* *- Используется сопоставление с образцом, древовидное представление структур данных и автоматический перебора вариантов с возвратами (бэктрекинг).* *- Вместо строгой постановки задачи предлагается рассматривать имеющуюся коллекцию фактов и правил вывода в произвольном порядке, не придавая значения недоопределённости, препятствующей применению обычных методов вычислений.* За этот период появилось 16 парадигм — самая яркая - ЛП
1980-1984	Улучшаемость программ	ООП	*Оказалось, что в процессе отладки большой программы часто достигается состояние, при котором её дальнейшая правка опасна порождением наведённых ошибок.* *- Предлагается методика представления программ как иерархии классов с поддержкой механизмов наследования, полиморфизма и инкапсуляции.* *- Выделяются типовые события или ситуации с поддержкой стандартных реакций и взаимодействия объектов с помощью сообщений или методов.* За этот период появилось 6 парадигм — популярная ООП.
1985-1993	Параллелизм	ПВ	*Пришлось осознать кризис сложившихся технологий программирования, порождающих эксплуатационно не вполне качественные программы, не удовлетворяющие требованиям эффективности при решении особо важных задач.* *- Концептуально основной резерв производительности компьютеров видится в использовании разных моделей параллелизма — высокопроизводительные вычисления на базе суперкомпьютеров.* *- Создаются и исследуются многочисленные модели параллельных вычислений.* *- Эксперименты по использованию математики множеств.* *- Предлагаются ЯП для представление естественного параллелизма задач и процессов.* *- Разрабатываются средства распараллеливания программ.* *- Появляется чистое ФП, смягчающее затруднения из-за присваиваний, а заодно и устройств.* За этот период появилось 12 парадигм — ведущая — ПВ с механизмами распараллеливания.
1994-2000	Сети и мобильные устройства	РП	*Прогресс элементной базы достиг высокой миниатюризации устройств и скорости информационной обработки, позволяющих использовать компьютерные средства в любых условиях. Кроме того, произошло резкое расширение возможностей сенсорного, гироскопического и голосового управления на уровне пользовательского интерфейса.* *- Переход от накопления правильности в форме библиотечных модулей к созданию диалектов или подъязыков, обладающих*

			правильностью по построению. *- Появляется ниша программирования бытовых электронных устройств и клиентских приложений с серверным программным обеспечением* *- Возникли распределённые приложения, рассчитанные на исчерпывающую информацию о желаемом ответе сервера.* За этот период появилось 8 парадигм — ведущая — рефлективное программирование (РП) на базе JVM.
2000-2021	Минимизация программирования	МП	*Пользовательские бизнес-приложения привели к росту числа разработчиков-любителей вместо квалифицированных программистов, особенно из-за задержек, связанных с коммерческими ограничениями, пандемией и торговыми санкциями. Кроме того, работодателям легче понимать ценность интерфейсных решений, чем выигрыш от системных модернизаций обработки сложно устроенных данных.* *- Появились технологии, позволяющие бизнес-пользователям дорабатывать приложения, не написав при этом ни одной строки кода — минимальное программирования (МП).* *- Предлагаются интуитивно понятные инструменты перетаскивания фрагментов программы, альтернативные методы разработки приложений без необходимости написания кода последовательно, вообще не требующие умения писать код. (Low-code, No-code, zero-code и т. п.)* *- Снова возникают попытки использовать декларативность на базе логических предикатов для задания ограничений по пропускной способности (constrait programming).* *- Ограничения отличаются тем, что они определяют не последовательность шагов для исполнения, а сохраняемые свойства решения, подобно аксиоматическим системам в абстрактных типах данных.* За этот период появилось 11 парадигм — ведущая — МП.

Часто любят утверждать, что основная задача программиста — это кодирование созданных математиками алгоритмов, выполнение подготовленных менеджерами технических заданий или реализация разработанных аналитиками спецификаций. Череда ПП, пройденных в ВЦ СОАН СССР под руководством Андрея Петровича Ершова, показывает, что истинная миссия программирования — **решение новых задач при отсутствии готовых средств и типовых методов**. Именно такое движение показано и в таблице 1. Развитие современного программирования проходит фазу формирования парадигм параллельного программирования, связанных с проблемами обработки больших массивов данных, организации суперкомпьютерных вычислений и эксплуатации сетевых компьютерно-коммуникационных взаимодействий в многопроцессорных комплексах.

1.5. Приоритеты приятия решений

Наиболее очевидная классификация ЯП по уровню абстрагирования слабо влияет на упорядочение приоритетов при выборе программируемых решений. Меняется лишь уровень требований к квалификации программиста и качеству программ.

Языки низкоуровневого программирования (ЯНУ) — возможность реализации эффективных решений ценой использования **общего доступа** к любым слабо защищенным структурам данных. Сочетание перспектив эффективности программ с низкой продуктивностью программирования. Программист обязан чётко понимать особенности и возможности компьютерной системы команд.

Языки высокого уровня (ЯВУ) — использование **иерархии**, независимые компоненты которой защищены от бесконтрольного взаимодействия. Повышение продуктивности программирования ценой снижения производительности и эффективности программ. Программисту достаточно понимать логику обработки данных и уметь её представлять в терминах языка программирования.

Языки и средства сверх высокого уровня (ЯСВУ) - БНФ, языки спецификаций, языки верификации программ, языки параллельного программирования, системы представления знаний и т. д., здесь важна полнота **пространства решений** по отдельным направлениям проблем, связанных с разработкой и применением программ. Перспектива удовлетворения особо сложных требований к программам ценой снижения продуктивности программирования и повышения квалификационного ценза. Программист должен обладать способностью к математическому абстрагированию.

Предметно-ориентированные языки (DSL) — переход от накопления опыта в виде библиотек к интеграции подъязыков или созданию диалектов (C-lang → LLVM). Перспектива продуктивности создания специализированных ЯП ценой ограничения их семантики и прагматики в пользу разнообразия синтаксиса, слабо влияющего на продуктивность программирования, но решающего проблему смены поколений[49]. Программист нацелен на создание изобразительных средств или интерфейса, достаточного для оперирования программой пользователем.

Более заметна разница в приоритетах принятия решений по важности разных средств и методов работы с данными, характерная для базовых ПП. В качестве стартового примера опишем такую разницу между базовыми парадигмами программирования на одно-процессорных конфигурациях. Для наглядности направления принятия решений обозначим цветами и символами:

- — ***E*** — вычисления,
- — ***M*** — объявление и распределение памяти,
- — ***C*** — управление вычислениями,
- — ***S*** — структуры данных.

Характеристика типовой постановки задач ИП:

1. Определён алгоритм решения актуальной задачи.
2. Необходимо получить программу реализации алгоритма с практичными пространственно-временными характеристиками на доступном оборудовании.
3. Программирование начинается с принятия решений относительно распределения памяти (**M**) и логики изменения её состояний (**C**) — хранимых в памяти данных. На фоне этого выбираются и размещаются необходимые вычисления (**E**) и, если понадобится, структуры данных, преимущественно вектора (**S**). Результатом является итоговое состояние памяти.

— MCES — Императивно-процедурное программирование (ИП)

Характеристика типовой постановки задач ФП:

1. Исследуется предметная область, обладающей задачами с недостаточной степенью изученности для производственного программирования.
2. Следует выбрать символьное представление данных для этой области и отладить систему универсальных функций, пригодных для создания прототипов решения актуальных задач из этой области.
3. Программирование начинается с принятия решений относительно необходимых вычислений (**E**) и структур данных (**S**) — иерархия списков. Потом может быть выполнено именование многократно используемых неизменяемых данных (**M**) и при необходимости приняты решения по порядку вычислений с однозначным строгим результатом (**C**).

— ESMC — Функциональное программирование (ФП)

Характеристика типовой постановки задач ЛП:

1. Доступна коллекция фактов и отношений для решения актуальной не вполне изученной задачи.

[49] Первичное обучение новичков в 3-5 раз быстрее, чем вторичное обучение опытных и зантоков.

2. Надо привести эту коллекцию к форме, демонстрирующей возможность ответов на практичные запросы относительно данной задачи для обоснования дальнейших исследований, повышающих степень изученности задачи.
3. Программирование начинается с принятия решений относительно недетерминированной схемы перебора имеющихся вариантов с помощью организации бэктреккинга (возвратов) и других методов при неудачном выборе хода вычислений (**C**). Затем происходит подбор подходящей схемы вывода результата из имеющихся фактов и отношений (**E**). При необходимости возможно создание структур данных (**S**) и выполняется именование некоторых многократно используемых формул (**M**). Результатом считается значение успешно выполненного варианта, если такой удалось найти.

— **CESM** — Логическое программирование (ЛП)

Характеристика типовой постановки задач ООП

1. Доступна работающая программа в виде иерархии классов объектов с работоспособными методами решения ряда задач некоторой зрелой сферы приложения ИТ.
2. Требуется без лишних трудозатрат быстро уточнить эту иерархию, чтобы приспособить её к решению новых востребованных задач этой области, её расширения или ей подобной.
3. Программирование начинается с принятия решений относительно иерархии классов объектов и методов их обработки (**S**). Предпринимается размещение объектов в памяти (**M**). Затем принимаются решения по управлению наследованием элементов полученной конструкции программы (**C**). После этого задаётся модель для выполнения необходимых вычислений (**E**), которую можно легко уточнять и заменять на другую без вмешательства или воздействия на общую конструкцию ранее отлаженной программы.

— SMCE — Объектно-ориентированное программирование (ООП)

Можно обратить внимание, что подходящие для базовых ПП задачи отличаются степенью изученности, влияющей на упорядочение приоритетов при выборе программируемых решений, что можно выразить в виде основной отличительной характеристики ПП. Получила ли парадигма поддержку программистов можно определить по наличию ЯиСП, поддерживающей ПП, и корпуса отлаженных с её помощью работоспособных программ.

Выводы

- Парадигма программирования представляет собой расширяющийся ряд отдельных ПП, поддерживающих продуктивное программирование решений конкретных классов задач.
- Расширение ряда ПП происходит как реакция программирования на появление трудных проблем, продуктивность решения которых не достаточна.
- Выбор ПП влияет на успех ТП, трудоёмкость ПЖЦП и продуктивность программирования решений нужных задач, а также на повышение степени изученности решаемой задачи, включая методы её решения, на организованность, работоспособность и живучесть разрабатываемой программы.
- ПП проявляется в удобстве понимания задачи, уточнения её постановки, продумывания и улучшения методов её решения, декомпозиции программы на фрагменты, модули, аспекты или компоненты, а главное, представления принятых в проекте решений в терминах лексикона программирования и ЯиСП, поддерживающих данную ПП.
- Классификация ЯиСП по поддерживаемым ПП начинается с анализа их эксплуатационной прагматики, но не сводится к результатам такого анализа.

- В наши дни признают в качестве базовых парадигм программирования: ИП, ФП, ЛП и ООП, опирающиеся на архитектуры компьютеров, и отмечают фундаментальное, образовательное значение таких парадигм программирования как ФП, ПВ и ИП, рассматривая ЛП, ООП и мета-программирование как дополнение фундаментальных парадигм, требующее знаний и навыков в конкретных областях программирования.
- В качестве основных парадигм, занимающих определенную нишу в технологиях программирования, признают ИП, ФП, ООП, а также событийно-ориентированное, параллельное и распределенное программирование, логическое, декларативное, чаще всего встречающиеся в практике
- Важно отметить, что ПП по-разному проявляют себя на разных классах задач, поэтому **выбор ПП влияет на продуктивность программирования,** успешность технологии программирования, эксплуатационные особенности программ, надёжность и безопасность применения программ.

Практика 1. Движущие силы программирования[50]

Подготовьте реферат с презентацией, посвящённый учёному, сделавшему значительный вклад в парадигмы и языки программирования. Для лауреатов Тьюринговской премии следует указать, что лауреат счёл важным отметить в своей тьюринговской лекции.

Грейс Хоппер - Cobol
Кеннет Айверсон - APL
Джон Бэкус — Fortran, БНФ
Питер Вегнер — конструктивное определение термина «парадигмы программирования»
Николас Вирт — Pascal, Пи-код, Oberon, Modula, Euler (расширение Algol средствами языка Lisp), структурное программирование (с Хоаром и Дейкстрой), система Lilith.
Андрей Петрович Ершов — школьная информатика, смешанные вычисления, системы АЛЬФА, БЕТА.
Роберт Флойд - «Парадигма программирования», логика Флойда-Хоара, редактор книг Д.Кнута «Искусство программирования».
Пол Грем (Paul Graham) — ANSI Common Lisp, диалекты языка Lisp - Arc , Bel.
Дональд Кнут - «Искусство программирования», языки MIX, MMIX.
Святослав Сергеевич Лавров — первая реализация Lisp в СССР (с Гиви Силагадзе), языки АБВ, Декарт и Спора, защита функционального программирования.
Питер Лэндин (P.J. Landin) — абстрактная машина secd и язык ISWIM
Джон Маккарти (*John McCarthy*) — универсальный язык программирования Lisp, функциональное программирование, искусственный интеллект, логика здравого смысла (Common Sense), технология разделения времени, повторное использование памяти (garbage collection)
Петер Наур (Peter Naur) — БНФ, термин «datalogy (информатика)»
Джон фон Нейман (*John von Neumann*) — принципы архитектуры фон Неймана, алгебра фон Неймана, аксиоматизация канторовской теории множеств.
Игорь Васильевич Поттосин — алгоритмы оптимизации программ, проект СОКРАТ, Открытая Всесибирская олимпиада по программированию им. И.В. Поттосина.
Алан Тьюринг (*Alan Mathison Turing*) - машина Тьюринга, теорема Чёрча — Тьюринга, тезис Чёрча — Тьюринга.
Тони Хоар (Tony Hoare) - CSP, Communicating Sequential Process), логикаХоара (*Hoare Logic*),
Алонзо Чёрч (*Alonzo Church*) — лямбда-исчисление, теорема Чёрча — Тьюринга, тезис Чёрча — Тьюринга.
Э́дсгер Дейкстра (*Edsger Wybe Dijkstra) - концепция* структурного программирования *(С Хоаром и Виртом), «*алгоритм Дейкстры*»,* идея семафора, дисциплина программирования, многопрограммная ОС THE, защищённые команды.
Фредерик Брукс-младший (*Frederick Phillips Brooks, Jr.) - «*Мифический человеко-месяц*» на основе опыта* OS/360 *в* IBM.
Барбара Лисков (Barbara Liskov) — язык CLU, абстрактные типы данных
Фрэнсис Аллен (Frances Elizabeth Allen) — оптимизирующая компиляция программ, распараллеливание.
Джекоб Шварц (Jacob T. Schwartz) – Теоретико-множественныйязык программирования Setl, язык сверхвысокого уровня.
Адольф Львович Фуксман — технология вертикального слоения программ (предшественник аспектно-ориентированного программирования), ежегодные школы по системному программированию и искусственному интеллекту на Чегете, на Эльбрусе, в спортивных лагерях «Лиманчик» и «Моряк» в окрестностях знаменитого Абрау-Дюрсо.
Валентин Фёдорович Турчин — РЕФАЛ, «Физики шутят»

Список можно продолжить.

[50] Здесь перечислены упомянутые в данном пособии учёные, полезно привлечь и других.

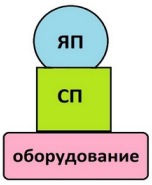
Лекция 2. Визуальная символика

Факультативная лекция. Поддержка языков, систем и парадигм программирования[51]

Задача этой лекции — разработка системы обозначений для объективного представления различий между парадигмами программирования, а также между языками и системами программирования, поддерживающими конкретные парадигмы. Такие обозначения нужны для быстрой идентификации особенностей парадигм, языков и систем программирования, освобождающей от изучения слишком пространных описаний, требующих дополнительной проверки. Объективность представления зависит от экспертной оценки и в свою очередь может быть проверена на подтверждающих примерах. В данной лекции описаны результаты эксперимента по выбору лаконичной визуальной символики и её сопоставление с механизмами поддержки языков, систем и парадигм программирования, а также с примерами на разных языках программирования. Эта символика понадобится при выполнении лабораторных работ по анализу и сравнению языков программирования.

Внешне простая символика может кодировать особенности определения языка программирования (ЯП) и его реализации в системах программирования (СП), поддерживающих разные парадигмы программирования (ПП). Символика нужна для лаконичного представления результатов анализа ЯП и быстрой, почти мгновенной, отбраковки неподходящих вариантов, подобно представлению экономических закономерностей в деловой графике. Попутно рассмотрим средства и методы, используемые при определении ЯП и их расширение при реализации СП. Для примеров используется материал языка Pure Lisp и средств структурного программирования на учебном подмножестве языка Pascal.

Учитывая, что в программировании критерием истинности является компьютерный эксперимент, естественно, жаргон практического программирования понятие «язык программирования» использует как «компилятор входного языка типовой системы программирования, функционирующей на базе определённой конфигурации оборудования» (ЯиСП).



Работающий компилятор для практика является единственно достоверным документом о реализованном ЯП. Пользовательская документация и описание стандарта ЯП - не более чем справочный материал, нуждающийся в проверке на соответствие тому, что фактически поддерживает система программирования. При непосредственных измерениях производительности программ вместо оценки программируемых решений[52] измеряется производительность комплекса из машины, компилятора и программы. Такие измерения слабо отражают вклад принятых программистом решений и выбранной ПП в производительность полученной программы, что при обучении программированию препятствует мотивации улучшать работающие решения.

---

[51] Материал этой лекции может оказаться сложным для преподавания, но он представляет интерес как дополнение для особо интересующихся.

[52] Появление практики, в которой принятие решения перекладывается на алгоритм, и это решение будет не чисто алгоритмическое как в классическом программировании, оно будет зависеть не от алгоритма, а от разработчиков нейронной сети и выбора учебного материала подобно тестированию, замаскированных как алгоритм. Парадигма здесь снова вне программирования: автоматизация принятия решений без ответственности за последствия — «сама машина сказала».

2.0. Лексика и синтаксис

Обычно определение ЯП задается раздельно на уровнях лексики, синтаксиса, семантики и прагматики. Лексика и синтаксис могут быть достаточно строго определены с помощью формул Бэкуса-Наура (БНФ), допускающих автоматическую генерацию распознавателей и генераторов языка. Определение семантики и прагматики имеет более сложную природу и выполняется в виде комплектов универсальных функций и абстрактных машин (см. 2.2). Кроме того, прагматика разлагается на эксплуатационную, определяющую особенности работы на уровне пользовательского интерфейса, и реализационную, определяющую особенности использования целевой аппаратуры, что можно называть машинным сопряжением, таким как интерфейс программируемого приложения (API)[53], которое позволяет использовать архитектурные возможности, не задумываясь о конкретной реализации конструкций ЯП[54]. Реализация СП для определённой семантики ЯП может быть выполнена как интерпретатор или компилятор или и то, и другое совместно. Компиляция может выполняться как для всей программы полностью, так и раздельно для указанных функций/процедур, в том числе «на лету».

Лексически близкие ЯП могут привлекать удобочитаемостью на первых шагах подготовки программ. При определении современных ЯП обычно избегают синтаксически сложно распознаваемых конструкций и сразу ориентируются на автоматическое построение распознавателей. Это способствует уменьшению числа ошибок при подготовке программ. Вариации лексики и синтаксиса влияют на удобочитаемость программ и распознавание принадлежности текста ЯП.

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

Таблица 2.1

Синтаксис бестипового учебного подмножества языка Pascal[55]

БНФ	**Пояснение**
Integer ::= digit {digit}	Целое без знака
Ident ::= letter {letter | digit}	Идентификатор
factor ::= ident | integer | "(" expression ")" **expression** ::= factor [("=" | "<" | "+"|"-" | "*" | "DIV") factor	Выражение
statement ::= [assignment | ProcedureCall | IfStatement | WhileStatement]	Оператор
assignment ::= ident ":=" expression	Присваивание
StatementSequence ::= statement {";" statement}	Последовательность команд
IfStatement ::= "IF" expression "THEN" StatementSequence ["ELSE" StatementSequence] "END"	Ветвление
WhileStatement ::= "WHILE" expression "DO" StatementSequence "END"	Цикл

53 API - ***a****pplication* ***p****rogramming* ***i****nterface*

54 Такая парадигма интерфейсной деятельности существует и вне программирования. Её можно называть «экономия мышления» и признавать, что «думать — человечеству труд тяжкий».

55 Понятие «тип данных» будет привлечено позднее.

БНФ	Пояснение
IdentList:: = ident {"," ident}. Declarations ::= ["CONST" {ident "=" expression ";"}] ["VAR" {IdentList;"}] FormalParameters ::= "(" [IdentList {";" = ["VAR"] IdentList }] ")". ProcedureHeading ::= "PROCEDURE" ident [FormalParameters]. ProcedureBody ::= declarations ["BEGIN" StatementSequence] "END" ident. **ProcedureDeclaration** ::= ProcedureHeading ";" ProcedureBody.	Определение функции
ActualParameters ::= "(" [expression {"," expression}] ")". **ProcedureCall** ::= ident [ActualParameters]	Вызов функции
Module ::= "MODULE" ident ";" declarations ["BEGIN" StatementSequence] "END" ident "."	Модуль

Можно видеть, что определение языка Pascal идёт от простого к сложному, от данных, удобно размещаемых в памяти машины, к выражениям, операторам, процедурам и модулям.

Ряд ЯП, такие как Lisp, РЕФАЛ, машинный код, языки символьной обработки и другие, представляют программы непосредственно как структуры данных, используемые при выполнении программы. В таком случае синтаксис на уровне форм Бэкуса-Наура выглядит проще. Используется его конкретизация для семантической сопоставимости с другими ЯП.

Таблица 2.2

Синтаксис диалекта Pure Lisp

БНФ	Пояснение
форма ::= переменная | (QUOTE S-выражение) | (COND {(форма форма)} | (функция {аргумент})	Переменная Константа Ветвление Вызов функции
аргумент ::= форма	
переменная ::= идентификатор	Имена, включая операции над списками
функция ::= название | (LAMBDA список_переменных форма) | (FUNC название функция)	 Безымянная функция Именование функции
список_переменных ::= ({переменная})	Любое количество, включая ноль
название ::= идентификатор	
идентификатор ::= атом	
S-выражение ::= атом | (S-выражение . S-выражение) | ({S-выражение})	Атом Консолидация (точечная пара) Список
атом ::= БУКВА {БУКВА | ЦИФРА}	

В этом случае определение движется от сложного к простому, от предельно возможного пространства форм и процессов к представлению данных, завершая атомами.

Смысл конструкций лексики и синтаксиса ЯП неявно связан с реализационной прагматикой (РП) той или иной ПП, поэтому, минуя их детальное рассмотрение, переходим к анализу особенностей семантики и РП. Исключение «синтаксического сахара»[56] в ЯП обеспечивает внутреннее понимание сущности реализуемого алгоритма при подготовке и отладке программы автором.

[56] «Синтаксический сахар» — это избыточность синтаксиса при одной и той же семантике, разные формы для записи одного и того же. Иногда это повышает удобочитаемость и изучаемость ЯП, чаще помогает смене поколений программистов.

Любое определение анализа текста программы выглядит как перебор **распознавателей**, передающих управление композициям из **селекторов**[57]. Достаточно определить набор распознавателей, выявляющих эти понятия, и селекторов, выделяющих их характеристики. Селекторы имеют смысл лишь при истинности соответствующего распознавателя. Основная проблема — слишком большой объём определения **универсальной функции**, определяющей отображение исходного текста программы в её исполнимый код, может быть преодолена рядом декомпозиций, приводящих к разложению на независимые составляющие, соответствующие отдельным ПП. Такая парадигмальная декомпозиция рассматривается как метод выбора критериев для согласованного выделения типовых компонентов в описании ЯП и определении СП, включая спецификацию реализационной прагматики, обеспечивающей варьирование используемых ПП в зависимости от фазы ПЖЦП решаемых задач и требований к продуктивности, эффективности и производительности программируемых решений.

Убедительный эксперимент по воплощению этого метода фактически произошёл при реализации СП для языка Lisp, использующей взаимодействие интерпретатора и компилятора при организации памяти со «сборкой мусора» с выделением базовой семантики в виде подъязыка Pure Lisp [10]. Несколько больше о языке Lisp будет сказано в лекции 4, посвящённой функциональному программированию. Роль абстрактного синтаксиса выполнил язык S-выражений, реализованных в виде списков, и позднее послуживший основой для парадигмы функционального программирования (ФП). В дальнейшем работы по языкам Алмо, Альфа, Сигма и внутреннему языку системы БЕТА показали перспективность многопроходной компиляции и оптимизации программ при ограниченных ресурсах [6]. Появление синтаксических конструкторов YACC и LEX над языком Си определило доминирование решения проблем машинно-зависимого переноса через сведение реализационной прагматики разных языков к ядру языка Си. ООП на базе классов объектов в C++ дало технику уточнения программных компонентов по мере эволюции постановки задачи. Целенаправленная попытка компонентного подхода к проблеме разработки СП на базе специально созданного внутреннего языка предпринята в проекте .Net и в других подобных проектах. По всем этим линиям был достигнут определённый успех, но он не образовал достаточную базу программотехники для авторов новых ЯП.

2.1 Семантика и прагматика

Проблема определения ЯП и СП наиболее тщательно проработана в Венской методике определения ЯП [166]. Эта методика разработана в конце 60-х годов. Основная идея — использование абстрактного синтаксиса (АС) и абстрактной машины (АМ) при определении семантики ЯП. 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. КЯ, АС, АМ, и КМ определяют один и тот же ЯП, их можно рассматривать как его диалекты разного уровня.

Диаграмма	Пояснение
КС ↔ АС → АМ → КМ	Существует отображение Конкретного Синтаксиса в Абстрактный Синтаксис и обратно. Абстрактный Синтаксис отображается в Абстрактную Машину. Абстрактная Машина реализуется с помощью Конкретной Машины.

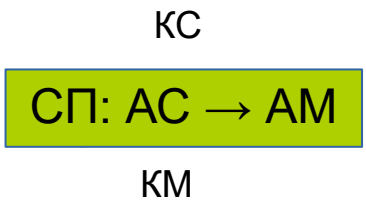
Диаграмма 2.1. Схема декомпозиции определения ЯП по Венской методике

Если КС удаётся нормализовать, то его перевод в АС и обратно (КС ↔ АС) можно построить автоматически, что можно сделать синтаксическими конструкторами типа YACC, LEX. Сущность определения языка концентрируется в виде так называемой универсальной се-

[57] Термины Венской методики определения ЯП.

мантической функции (УФ) языка (УФ : АС → АМ), выполняющей переход от абстрактного синтаксиса *к* абстрактной машине — трансляцию (интерпретацию и/или компиляцию). УФ и АМ образуют инвариант определения функциональной и операционной семантики ЯП. Такая абстрактная декомпозиция достаточна для установления первичной границы между фронт-эндом и бек-эндом. Каждая стрелка в этой диаграмме означает, что существует доказательство эквивалентности представлений программ.

Для производственных ЯП определение УФ достаточно сложно, что приводит к ошибкам в оценке трудоёмкости их реализации не менее чем в два раза. По этой же причине нередко определение семантики ЯП сводят к определению АМ, реализация которой чувствительна к навыкам низкоуровневого программирования, и потому её реализационная прагматика часто остаётся без формализации. Выбор АМ существенно влияет на пространство допустимых КМ. Сопоставление Венской методики с визуальной схемой термина «язык программирования» можно представить рисунком, символизирующим его понимание, с уточнёнными надписями.



Первый эксперимент по применению Венской методике был выполнен на материале весьма сложного языка PL/1 и показал высокую трудоёмкость. Тем не менее, основная схема Венской методики прижилась при слегка изменённой терминологии в системах поддержки реализации ЯП. Так АМ стали называть «виртуальная машина» или «пи-код», или «байт-код» или «бек-энд» и т. д. Появился термин «middleware», дополняющий АМ средствами управления сообщениями, обработки данных из разных источников, транзакциями, аутентификацией и авторизацией, интеграцией различных систем, обеспечением услуг, различных сервисов, кеширования, маршрутизации запросов и управления сессиями, включая клиент-серверные, распределенные системы и облачные решения. Аналогично, АС стали называть «синтаксический граф», «дерево разбора», «фронт-энд» «абстрактное синтаксическое дерево» и др.

Граница между АС и АМ может быть уточнена введением уровня базовых средств (БС), что гарантирует при тривиальности перехода БС ↔ АМ профилактику усложнённости АМ и взаимозаменяемость АМ и БС на первых шагах раскрутки реализации СП.

Диаграмма	Пояснение
КС ↔ АС ↓ **БС** ↘ ↕ **КМ** **АМ** ↗	При отображении Абстрактного Синтаксиса в Абстрактную Машину выделяется уровень Базовых Средств, просто отображаемых на Абстрактную Машину и обратно. Переход к Конкретной Машине может быть представлен и как реализация Базовых Средств.

Диаграмма 2.2. Декомпозиция определения ЯП с выделением уровня базовых средств

В состав БС включают операции ЯП и средства управления вычислениями, не сводимые к более простым средствам ЯП. Выделение БС гарантирует чёткое отделение языковоориентированного фронт-энда от машинно-ориентированного бек-энда, что даёт минимизацию трудоёмкости при переносе СП на новую аппаратуру. Такая декомпозиция была выполнена в первых реализациях языков Lisp и C. Можно ограничить БС одной областью данных, рассчитывая, что операции над другими областями данных несложно включить как дополнительные команды, расширяющие АМ. Тем не менее, даже для не слишком сложных ЯП число одноуровневых функций, образующих фронт-энд, легко превосходит тысячу, дополненную рядом библиотек и пакетов, нередко обладающих общим функциональным назначением при различных механизмах реализации. Попытки улучшения

СП обычно требуют повторного программирования заметного числа взаимосвязанных функций над общими данными или дополнения СП новой библиотекой, нацеленной на повышение производительности программ. В последнем случае программист получает рекомендацию отредактировать ранее отлаженную программу, особенно такое характерно при переходе к параллелизму.

Сложность целостной реализации универсальной функции **УФ: АС → БС → АМ** преодолевается декомпозицией её на прикладные семантические системы (ПС) — это **семантическая декомпозиция**. Прикладные семантические системы $\{ПС_1,\ldots,ПС_n\}$ представляют отдельные механизмы ЯП, в принципе эффективно сводимые к аппаратным решениям при реализации и развитии автомата АМ → КМ. ПС можно рассматривать как подъязыки ЯП.

Диаграмма	Пояснение
КС ↔ АС = {ПС₁,…, ПСₙ} ————————— ↓ **БС** ↘ ↕ **КМ** **АМ** ↗	При анализе Абстрактного Синтаксиса выделяются Прикладные Семантики, реализация которых возможна с помощью Базовых Средств — семантический спуск. Предварительная реализация Прикладных Семантик может быть выполнена средствами реализуемого ЯП.

Диаграмма 2.3. Декомпозиция определения ЯП с разложением на прикладные семантики

Это позволяет реорганизовать УФ в комплекс из более простых универсальных под-функций **$УФ_i$: $ПС_i$ → БС → АМ**. Примерно так устроено описание языка Basic, представляющее собой набор подязыков, обеспечивающих доступ к отдельным средствам аппаратуры, пополнение состава которых выполняется добавлением соответствующего под-языка, слабо связанного с остальными.

Комплект основных видов $ПС_i$ в качестве типовых компонент ЯП обычно предоставлен средствами работы с памятью (M -), организации вычислений (E -), обработки структур данных (S -) и управления вычислениями (C -), реализация которых в разных архитектурах и ЯП обладает общей спецификой. Возможно выделение средств диагностики, типового контроля, средств укрупнения действий[58] и т. д. Такая семантическая декомпозиция ЯП имеет шансы достичь независимости развития соответствующих ей типовых компонентов СП. Обычно ЯП содержит около 20-ти видов ПС, причём число ПС одного вида может приближаться к десятку. Выделение таких ПС обусловлено различиями в схеме применения операций к операндам, связанными с реализационной прагматикой СП и наличием их аппаратной поддержки, что можно выражать как понятие «семантическая система», выделяющая конкретный тип данных (ТД), набор базовых операций (БО) над ним и правило применения операций (ПО) к данным.

Интеграция новых компонентов ЯП в ранее реализованную версию СП требует включения ряда согласованных определений на всех уровнях определения ЯП. На уровне АМ это означает дополнение её регистрами и специальными командами.

Для перехода от определения ЯП к реализации СП ключевое значение имеет семантика, но для уяснения ПП, поддерживаемых ЯП, требуется понимание вариантов реализационной прагматики, которая может быть не представлена в определении или стандарте ЯП, но подразумеваться традиционно или воспроизводится по прототипу. Выбор РП существенно влияет на пространство процессов выполнения программ.

 Вычисления характеризуются объявлением типов результата и аргументов. Традиционные формы вычислений позволяют строить потоки вычислений и конвейерные процессы, управляемые готовностью данных, без именования промежуточных значений.

[58] «Средства укрупнения действий» — это макросы, функции и процедуры, а также библиотечные модули или пакеты. Обычно при рассмотрении методов компиляции подразумевается, что шаблоны компиляции строятся из конструкций более низкого уровня, чем конструкции ЯП. Иногда оказывается, что соотношений уровней ЯП и целевой машины в некоторых позициях не соответствует такой гипотезе. Тогда возникает необходимость в тексте программы выделять укрупнённые конструкции, соответствующие машинным командам, если они важны для обеспечения эффективности кода.

Такие системы можно различать по мощности множества допустимых значений, набору встроенных операций над ними, возможности конструировать новые операции и правила применения операций к данным.

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[59] (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, прерываний, 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/непрерывные и распределенные/дискретные структуры данных, статическое и динамическое размещение данных в памяти, аналитические, счётчиковые и программные методы повторного использования памяти для структур данных.

Реализационная прагматика, влияя на принятие решений на всех уровнях определения ЯП, в основном предоставляет решения в области конкретной организации вычислений, 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.

Для основных ПП определение АМ сводится к системе команд (СК) над небольшим числом регистров, обычно от двух до шести. СК может быть получена как отображение части БС, а состав регистров и особенности их функционирования отражают реализационную прагматику обработки данных.

Диаграмма	Пояснение
КС ↔ АС = { ПС$_1$,…, ПС$_n$ } ↓ ↓ ↓ **БС = { БО$_1$,…, БО$_n$ }** ↕ ↕ ↕ **АМ ~ < СК$_1$,…, СК$_n$ > → КМ** **~ ----------** **~ регистры**	ПС = < ТД, БО, ПО > - Тип Данных, Базовые Операции, Применение Операций При анализе Абстрактной Машины для эффективной реализации Прикладных Семантик определяются Системы Команд над конкретными регистрами.

Диаграмма 2.4. Декомпозиция определения ЯП по интерфейсу с реализационной прагматикой

Компоненты $_{уф}$**i:** $_{ПС}$**i** → $_{БО}$**i** → $_{СК}$i обретают более чёткую форму, но следует отметить, что разные $_{СК}$i могут внешне совпадать, обладать пересечениями или при реализации сводиться к различным

[59] Ассоциативно именуемые значения типичны для языка Lisp: подразумевается, что существует ассоциативный список, в котором имена связаны в пары с их значениями. Например. ассоциативный список вида ((А . 1) (Б . 2) (С . 3)) при интерпретации программы означает, что А=1, Б=2, С=3. Одно и то же имя может водить неоднократно. Именованные переменные означают, что существует вектор переменных, например вектор [А, Б, С], а значения переменных хранятся в другом, также устроенном векторе [1, 2, 3] - прообразе памяти для хранения значений переменных при исполнении кода программы после её компиляции. Имена переменных уже не хранят, а значения доступны. Зная номер переменной, по этому же номеру достаём её значение при выполнении кода.

парадигмальным вариантам. Кажущаяся простота системы команд (СК) обладает замаскированной сложностью, вызванной тем, что в отличие от понятий уровня ЯП, обладающих сравнительно вольной комбинаторикой, элементы СК могут быть подчинены жёстким связям, от учёта которых может существенно зависеть корректность выполнения программы. Поэтому для наглядности кроме образа «кубик Рубика» по началу для характеристики приаппаратных средств архитектурной модели будем использовать образы тетраэдра, в котором можно показывать взаимодействия любых составляющих друг с другом, и его треугольных проекций как геометрически жёстких фигур, иллюстрирующих декомпозицию прагматики на отдельные плоскости, связанные с используемыми архитектурными решениями на уровне базиса ЯП. Тетраэдр удобно приспособлен к обзору его по отдельным сторонам, парам сторон и тройкам, а при необходимости можно сделать его развёртку на треугольник из четырёх сторон.

Множество прикладных семантических систем можно разбить на подмножества, реализуемые в рамках одинаковых парадигм. Обычно РП может поддерживать одну ПП, но встречаются $ПС_i$, требующие совместного применения ряда парадигм. Например, в языке F# реализация ленивых вычислений выполнена на стыке функционального и объектно-ориентированного программирования.

Диаграмма	Пояснение
КС ↔ АС = { $ПС_1$,…, $ПС_n$ } ↓ ↓ ↓ **БС = { $БО_1$,…, $БО_n$ }** ↕ ↕ ↕ **АМ ~ < $СК_1$,…, $СК_n$ >** ↕ ↕ ↕ **РП ~ < $РП_1$,…, $РП_n$> → КМ**	Определение Абстрактной Машины дополняется отображением её Систем Команд на ряд парадигмальных вариантов Реализационной Прагматики, представляющих собой подмножества комплектов, поддержанных в системах программирования этих парадигм.

Диаграмма 2.5. Декомпозиция определения ЯП с парадигмальными вариантами реализационной прагматики

Трудоёмкость реализации и жизнеспособность компонент СП для развивающихся и мультипарадигмальных ЯП зависит от того, удаётся ли декомпозировать реализационную прагматику ЯП на парадигма-зависимые варианты реализации отдельных ПС.

Таблица 2.3

Основные конструкции ядра языка Pascal над целыми числами

Конструкция	Пояснение
`X`	Переменная
`123`	Константы
`C`	
`(A1 + A2)`	Выражение
`(A1 = A2)`	Отношения

(A1 < A2) (A1 <= A2) (A1 > A2) (A1 >= A2)	
;	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

Таблица2.4

Основные конструкции ядра Pure Lisp над списками[60]

Конструкция	Пояснение
X	Атом (переменная)
(quote C)	Превращение в константу. Константой может быть любое символьное выражение
(atom X)	Проверка символьного выражения на неделимость
(eq X Y)	Совпадение атомов
(cond (P ST)…(T SF))	Ветвление *
(lambda (X …) E…)	Безымянная локальная функция
(defun F E)	Именование глобальной функции[61]
(F A …)	Вызов функции

Унификация значений и функций в языке Lisp позволяет не включать в базовые средства именование функций, рассматривая такой механизм как вспомогательную

[60] Может сам работать как свой АС.

[61] Вместо *defun* в Pure Lisp использовался конструктор *label* – именование локальной функции, учитывалось, что глобальные определения — это самые внешние локальные.

семантику «Категории объектов», которые появятся в более полном определении ЯП. Формально, Defun можно свести к передаче параметра с помощью Lambda[62].

Следует принять во внимание, что **ЯП с общей абстрактной машиной (АМ) равномощны по пространству порождаемых процессов.**

Таблица 2.5

Представление АС ядра языка Pascal над целыми числами в форме списков

Конструкция в языке Pascal	**Представление в виде списка**
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)
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 АС позволяет распознаватели разных конструкций (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), вырабатывающие параметры семантической функции, сводятся к композиции шагов доступа, выполняемых после

[62] Такой сценарий полезен при анализе возможностей тех или иных конструкций языка с целью выделения семантического базиса, реализация которого может рассматриваться как основа СП. Остальные конструкции языка можно для начала определять в терминах семантического базиса, а потом реализовывать их более эффективные эквиваленты.

соответствующего распознавателя. Такие определения, сводящиеся к выбору 2-го, 3-го или 4-го элементов списка, практически не требуют отладки, работают с первого предъявления.

Конструкцию while можно в базовую семантику (БС) не включать, т.к. она сводима к func[63]. Можно выделить вспомогательную семантику (ВС) «Представление итераций», которое в полном ЯП будет включать и другие форматы циклов.

ЯП с общим АС семантически эквивалентны, они сравнимы по трудоёмкости или продуктивности отладки программ[64].

Различают два стиля задания семантики ЯП: *семантика вычисления значений* (ФП) и *семантика изменения состояний памяти* (ИП). Граница между этими стилями достаточно условна. Ничто не препятствует семантике вычислений в качестве одного из аргументов функций использовать общее состояние памяти, а семантике изменения расширить понимание присваивания на переход от аргументов функции к её результатам. Основой определения интерпретатора для семантики вычисления значений во многих ЯП является функция EVAL (evaluation), вычисляющая произвольные выражения языка с учетом состояния таблицы атомов (ТА), устроенной как список или стек[65] или хэш-таблица. Для семантики изменения состояний памяти можно определить функцию EXEC (execution), выполняющую ту же работу на базе двух устроенных как векторы таблиц ТА и TF, раздельно хранящих значения переменных и определения процедур, соответственно. Вместо списков произвольной длины используются вектора фиксированного размера.

Универсальная функция, или интерпретация — это функция, которая может вычислять значение любой формы, включая формы, сводимые к вычислению любой 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 до исчерпания стека.)

Рассмотрим особенности универсальной функции eval от аргумента expr. Выражение expr является произвольной вычислимой формой языка Lisp. Универсальная функция должна предусматривать все основные виды вычисляемых форм, задающих значения аргументов, а также представления функций в соответствии со сводом правил языка. При интерпретации выражений учитываются следующие решения, различимые в определении АС:

- 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. Они могут быть связанными, перечисленными в списке параметров функции, или свободными, просто входящими в определение;
- константы, представленные как аргументы функции QUOTE, можно просто извлечь из списка ее аргументов. Например, значением (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

---

63 Сокращение от слова Function.

64 АС представляет семантику ЯП, а основная работа по отладке программы выполняется преимущественно на уровне семантики. При синтаксических различиях семантически эквивалентные языки порождают одно и то же семейство процессов вычислений. Многое ещё зависит от прагматики, но современная практика старается её скрывать или игнорировать ради «упрощения» работы программиста. Не исключена ситуация, когда ЯП с общим АС семантически эквивалентны, но при этом семантически не равномощны (АС общий, а АМ — нет), могут быть разные АМ.

65 Разница в структуре данных для таблиц вызвана исключительно историческими традициями выбора базовых средств в соответствующих системах программирования. Теоретически и вместо стека можно использовать вектора и, наоборот, вместо векторов — стеки. Да и стеки можно реализовывать по-разному.

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 представление их определения, которое находит подобно переменной по имени в таблице атомов — в контексте[66];

- 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 представлением ее определения так, чтобы корректно выполнялись рекурсивные вызовы функции. Определения функций накапливаются в «хвосте» системной переменной ТА, то есть работают как глобальные определения[67].

Таким образом, интерпретация выражений в языке Pure Lisp осуществляется как взаимодействие четырех ВС:

S – обработка структур данных (cons, car, cdr);
E – выработка значений (atom, eq);
M – конструирование и идентификация объектов (lambda, список параметров, defun);
C – управление логикой вычислений и частичными вычислениями (композиции, quote, cond, eval).

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.

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

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

Универсальная функция EXEC для минимального подмножества языка Pascal немного отличается:

M: вместо атомов есть имена переменных или констант или функций/процедур;
M: константы могут быть безымянными, предпочитается скаляр в качестве значения;
M: определения переменных и функций или процедур хранятся раздельно.
C: условное выражение состоит из предиката и двух позиций для выбора нужной ветви;
C: существуют специальные формы для циклов, определения и вызова процедур и функций и другие схемы управления вычислениями;
C: выражения обрабатываются с учетом таблицы приоритетов операций и расстановки скобок;
E: вычисления строятся как выражения над переменными, константами, элементами векторов и функциями;
S: структуры данных устроены как вектора конечного размера.

Основные различия:

---

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

67 Это значит, что при изменении области видимости мы достаем из стека то, что мы положили в этой области видимости, но функции остаются, потому что мы их положили в самый конец (Примечание Карпова Матвея).

EXEC не поддерживает безымянные/анонимные функции;

EVAL не поддерживает присваивания глобальным переменным;

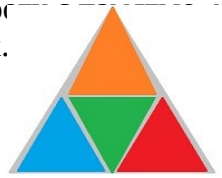
EXEC использует раздельное хранение значений переменных и определений функций/процедур;

EVAL допускает конструирование определений функций в процессе вычислений и хранение определений равноправно со значениями переменными.

2.2. Абстрактная машина

При сравнении императивного и функционального подходов к программированию, П. Лэндин (P.J. Landin) предложил специальную абстрактную машину secd **в качестве АМ для ЯП функционального подхода. Подробное описание этой машины можно найти в книге П. Хендерсона по функциональному программированию. Машина secd работает над четырьмя независимыми регистрами (Stack, Environment, Control list, Dump)[68]: стек для результатов (S), контекст — локальная память для именованных значений (E), управляющая программа (C), резервная память (D), работающая похоже на транзакции -** . Символически и её можно изобразить треугольной развёрткой тетраэдр

а, в котором голубой цвет символизирует вычисления, красный — рабо кёлтый — управление вычислениями, зелёных — обработку структур данных.



Для ЯП императивного подхода аналогичная машина должна быть устроена иначе. И она может работать над четырьмя регистрами, но вместо независимых регистров произвольного размера используется концепция конечной общей памяти ((Stack, Environment, Control program, Memory): стек для значений (S), иерархическая структура локальных контекстов[69] для аргументов функций, локальных переменных и регистра возврата (E), управляющая программа (C), общая память (M), изменяемая безвозвратно — машина secm, в которой вместо бесконечного дампа для хранения неизменяемых данных «D» используется общая память «M» конечного объёма, хранящая данные, подверженные произвольным изменениям - , что можно представить внешне таким же тетраэдром, но его элементы памяти будут иметь другое наполнение, иной смысл. Таким образом, устройство АМ для разных парадигм имеет заметные пересечения, но и специфику, для анализа которой требуется более детальное представление механизмов обработки данных, что более подробно будет показано в лекциях 3—9. Здесь, для примера, рассмотрим и отчасти сравним лишь основные два подхода к заданию семантики — функциональный и императивный.

Особенности процесса компиляции достаточно сложны даже для простых языков, поэтому спецификация результата компиляции часто задается в терминах языково-ориентированных абстрактных машин (АМ), более строго задающих прагматику ЯП. 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. Такое определение может быть машинно-независимым и переносимым. Не зависимо от парадигмы АМ различает обычно следующие категории команд:

E = s : вычисления над безымянными операндами в стеке при обработке выражений;

M = e : засылка значений из памяти/контекста в стек;

M = e : пересылка значений из стека в память/контекст с учетом локализации;

---

[68] Лэндин не раскрашивал регистры

[69] Противопоставляется глобальной памяти для хранения переменных.

C = c : организация ветвлений и циклов;
C = c : организация вызовов процедур и функций
S = d : сохранение/восстановление контекста.
Могут быть и другие категории команд.

Абстрактная машина secd удобна для спецификации машинно-зависимых аспектов семантики языка Lisp. Регистры приспособлены к хранению выражений в форме указателей на атомы или списки. Состояние машины полностью определяется содержимым этих регистров. Поэтому функционирование машины можно описать достаточно точно в терминах изменения содержимого регистров при выполнении команд, что выражается как переходы 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 компилируемый код. Это позволяет формально считать равноправными встроенные и программируемые функции. Компилятор по исходному тексту программы строит код программы, функционально эквивалентный тексту.

Стек устроен традиционно по схеме «первый пришел, последний ушел » — FILO. 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 (STOP) d → s e (STOP) d

Система команд АМ может быть расширена простым дополнением правил. Так, можно её механизм распространить на разные типы данных[70], например, на 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) d → ((a + 1) . s) e c d
(a . s) e (MIN1 . c) d → ((a - 1) . s) e c d
(a . s) e (NUMBER . c) d → (t . s) e c d

t — истинностное значение NIL или T.

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

Таблица 2.7

Расширение Pure Lisp и secd 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)	

[70] Чистое определение Pure Lisp работает только с абстрактными атомами, без чисел, литер или строк.

	(DIF x y)	(X - Y)	Добавление операций над новым типом данных
	(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

Эти два похода к АМ обретают небольшое расширение при переходе к парадигмам логического и объектно-ориентированного программирования. Для работы с вариантами в ЛП добавляется вспомогательная память — регистр «R», а для ООП используются две дисциплины работы с памятью, объединяющие возможности регистров «M» и «D». Для ЯП логического программирования определяется машина secdr, дополняющее определение secd регистром R для хранения не опробованных вариантов. При переходе от императивного программирования к ООП строится машина secdm включением резервной памяти D в машину secm. Можно сделать вывод, что набор регистров АМ позволяет различать базовые ПП на уровне архитектурной модели.

secd — регистры абстрактной машины ФП.
secm — регистры абстрактной машины ИП.
secdr — регистры абстрактной машины ЛП.
secdm — регистры абстрактной машины ООП.

Предстоящие шаги инкрементального, т. е. пошагового, процесса раскрутки СП могут быть связаны с расширением типов обрабатываемых данных, подключением удобных форматов управления вычислениями, специализацией особых категорий функций, созданием программного инструментария и т. д. до полного или практически достаточного покрытия ЯП его реализацией в СП. Каждый шаг можно рассматривать как реализацию диалекта ЯП.

Для производственного применения ЯП требуется расширение такой семантики средствами обмена данными с периферийными устройствами, позволяющими вводить программы и видеть результаты её выполнения. В secd и то, и другое происходит чудом. Машина secd достаточна для обеспечения полноты вычислений по Тьюрингу, а для соответствия архитектуре Фон Неймана нужна определённость средств ввода-вывода данных. Такие средства Н.Вирт включил в описание P-кода для языков Pascal и Oberon[71], изображённого здесь как аналогичная машина secm и дополнительный регистр «W» для работы с периферийными устройствами. Такое дополнение разумно и для secd → secd-w, поскольку реально при отладке программ используются функции Print и Read. Машина secm-w работает над пятью регистрами (Stack, Environment, Control_program, Memory, World): стек для промежуточных значений (S), контекст для размещения аргументов, иерархии локальных значений переменных и регистра возврата (E), управляющая вычислениями программа (C), память (M) и внешние устройства (W). Состояние машины полностью определяется содержимым этих регистров. Поэтому поведение машины можно описать достаточно точно в терминах изменения содержимого регистров при выполнении команд.

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

Разница между secd-w и secm-w в основном сводится к операциям над данными. Кроме того, secm-w имеет дополнительные команды по непосредственной работе с памятью для реализации присваиваний, передаче управления для реализации итераторов и обмена с внешними устройствами. Обе абстрактные машины, как secd-w, так и secm-w, содержат, кроме образа базовых средств ЯП, дополнительные команды, обеспечивающие пересылки данных между регистрами и реализационные средства, поддерживающие эффективность

[71] Theory and Techniques of Compiler Construction. Addison-Wesley, Reading, 1996. ISBN 0-201-40353-6

реализации ЯП (rplaca, передачи управления, 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 СП.

Дальнейшее развитие СП заключается в пошаговом расширении спектра обрабатываемых данных и присоединении ВС до полного покрытия ЯП его реализацией в СП, возможно в форме функционально эквивалентных диалектов ЯП. Каждую новую 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 или компилятора;
- дополнение АМ новыми командами.

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

Данные = array of

AM: 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 ТД переменным. Таким образом, реализационное замыкание ЯП над secm включает в себя представление ТД для всех переменных, включая параметры процедур.

Таблица 2.8

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 значения 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`

Новая конструкция	Пояснение	Пример
`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 может быть сведено к подключению ТД и присоединению ВС методом раскрутки — пошаговой разработки. Цели раскрутки:

- повышение продуктивности начального этапа программирования;
- увеличение потенциала СП, т.е. числа типовых задач, решение которых обладает приемлемой для практики трудоёмкостью.
- ЯП с общей АМ семантически равномощны, на их основе достижима сравнимая эффективность процессов вычислений.
- Трудоёмкость реализации АМ с помощью КМ можно оценить на основе материала по низкоуровневому программированию (Лекция 3), а также при сравнении АМ с Р-код-ом и RISK-машиной, предложенными Н. Виртом при разработке учебных языков Pascal и Oberon, и учебными машинами MIX и MMIX, описанными Д. Кнутом.
- Семантический спуск[72] от полного ЯП к его БС характеризуется повышением продуктивности реализации ядра ЯП и его АМ, сопровождаемое увеличением трудоёмкости применения такого диалекта — частичной реализации ЯП.
- Трудоёмкость применения можно оценивать числом понятий, возникающих при программировании сверх тех, что присущи решению типовых задач.
- Исследование разных схем частичных, смешанных, «ленивых» вычислений[73] и мета-компиляции приводит к выводу о целесообразности совмещения таких схем в рамках общей СП с целью использования их преимуществ на разных уровнях изученности решаемых задач.[74]

Теория программирования утверждает, что определение компилятора может быть выведено из определения интерпретатора методами смешанных вычислений. Компилятор по такой методике получается как остаточная программа при смешанном вычислении интерпретатора.

2.3. Структуры данных

Кроме распределения памяти при компиляции различимы дисциплины, 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;

---

72 Переход от сложных конструкций ЯП к их более простым эквивалентам.

73 Ленивые вычисления возможны в любой парадигме, не только в ФП, это всего лишь вопрос выбора времени счёта. Во времена фон Неймана они обсуждались, но не хватало памяти для хранения отложенных программ. Лишь повышение качества магнитных носителей позволило через несколько лет использовать для этого разные оверлеи. ФП сделало возможным ленивые вычисления, но лет 15 мало кто это замечал.

74 Так начали делать при реализации языка Scheme.

- «близнецы» — метод укрупнения памяти и профилактики её чрезмерной фрагментации при распределении на разно объемные блоки [75];
- 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 СП:
- векторы с паспортом, хранящим при компиляции сведения о размерах и типе элементов, не доступны при исполнении программы;
- запись или структура, обеспечивающая доступ к заданному перечню разносортных элементов по статически определённым ключам;
- объединение заданного перечня разных ТД, размещаемых в разное время по определённому адресу;
- множество небольшого числа перечислимых элементов, обработки которых не выводит за пределы машинных команд над битовыми кодами;
- стек, допускающий две дисциплины доступа — FILO и вектор.

В машине списки хранятся не как последовательности символов, а как структурные формы, построенные из указателей на машинные слова как на части деревьев, подобно записям в языке Pascal при реализации односвязных списков. В результате:
- 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;
- ячейки можно переносить в список свободной памяти, как только исчезнет необходимость в них. Даже возврат одной ячейки в список может быть полезен, но если данные хранятся линейно, то организовать использование лишнего или освободившегося пространства из блоков ячеек трудно;
- 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, стала автоматизация повторного использования памяти - «сборка мусора» (GC - *garbage collection*). С начала 1960-х годов методам такой работы посвящены

---

[75] Компактизация обычно выполняется перемещением данных в непрерывную область памяти, желательно без пустых промежутков. Это затратный и не вполне безопасных процесс. Не исключена потеря данных при сбоях. Метод «близнецы» выполняется на фоне предварительного распределения памяти на области дозированного объёма, обычно это степени 2. Данные располагаются в подходящей по объёмы области. Если такую область надо освободить, то она перемещается к области такого же объёма, если такая найдётся, и они объединяются в одну область большего размера. Возможен и обратных процесс разбиения на более мелкие области, но всегда при сохранении адресов для начал областей в виде степени двойки. Получается сокращение времени при поиске пустого пространства. Это не очень популярный метод, т. к. его реализация сложнее.

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 свободной 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 «сборка мусора» выполняет анализ достижимости всех блоков памяти простой пометкой узлов, видимых из конечного числа рабочих регистров системы программирования[76]. К таким регистрам относятся промежуточные результаты вычислений, активная часть стека, ассоциативный список, таблица атомов и др. После пометки все не помеченные узлы объединяются в список свободной памяти[77], передающий их для повторного использования новым вызовам функции CONS. Такая автоматизация не лишена недостатков.

Ряд неприятных следствий: непредсказуемые длинноты (время работы) при поиске очередной порции ячеек и «перегрев системы», если такие порции слишком малы для продолжения счета, то сборка мусора происходит слишком часто. 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. 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[78]. 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. Декларация о бестиповости языка Lisp устарела и в наши дни заменилась 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

---

[76] Исторически он именно в ФП появился примерно в 1960-ом году

[77] Автоматически в разных системах используют различные критерии вызова GC, кроме того, обычно имеется возможность вызвать его из программы. Утверждение про нехватку памяти является фигурой речи, показывающей цель GC, точнее можно было сказать «для профилактики нехватки памяти». Просто целью является избегание ситуаций нехватки памяти. Упомянутые ЯП появились, когда уже был накоплен изрядный опыт сборки мусора, были известны разные методы и подходы. Впрочем, возможность вызывать сборку мусора из программы имелась уже в начале 1960-ых.

[78] — В Си и большинстве систем программирования память выделяется блоками разной длины, предназначенными для хранения векторов с доступом по индексу. В результате возникает проблема, что если блок не полностью занят данными, то часть памяти как бы не используется - простаивает. Бинарные деревья из небольших узлов свободны от таких простоев, хотя на первый взгляд иногда кажется, что происходит избыточная затрата памяти на хранения адресов таких узлов.

— В результате возникают ситуации, когда заранее неизвестно сколько элементов нужно или число элементов изменилось так, что заметная часть блока ничем полезным не занята. Конечно, можно заводить меньший блок, но так редко кто делает, при этом не всегда старый вектор освобождает память или программист должен явно объявлять освобождение памяти, что требует дополнительного внимания. В системах программирования чаще всего используют модели со счётчиками, что требует дополнения в виде уплотнения памяти, иначе есть риск фрагментации памяти на мелкие кусочки так, что память вообще-то имеется, но нет непрерывных интервалов для удобного доступа к элементам вектора по индексу. Появляется возможность автоматического освобождения памяти, хотя не сразу сложились достаточно удобные методы.

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 на этапе исполнения программы.

Эффективность приведенного далее определения интерпретатора с использованием ассоциативного списка существенно зависит от числа различимых атомов в программе. Такая зависимость обычно смягчается механизмом функций расстановки (хэш-таблиц), обеспечивающим доступ к информации по ключу. В качестве ключа используется имя атома. 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 конструкции — 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.

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, структуро-разрушающие 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 описания Lisp 1.5:

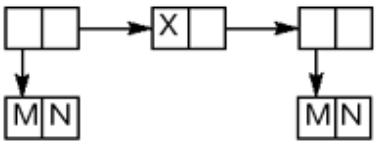
Диаграмма	Пояснение
	При создании структур многократные вхождения одинаковых данных получают независимое расположение в памяти.

Рис. 2.6. Пример ((M . N) X (M . N))

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

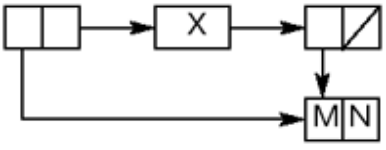
Диаграмма	Пояснение
	Слияние многократных вхождений одинаковых данных.

Рис. 2.7. 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))
```

2.4. Семантическая система

Теперь более подробно рассмотрим выбор формы для представления результатов сравнения ЯиСП, удобной для оценки выразительной силы ЯП и трудоёмкости или продуктивности реализации СП. Форма приспособлена к парадигмальному анализу определений ЯП и выбору практичных критериев декомпозиции программ, что можно рассматривать как подход к решению проблемы факторизации определений ЯиСП. Отметим сразу, что отношение между ЯП и ПП не является однозначным. Многое ЯП поддерживают комплекты ПП и большинство ПП поддержаны многими ЯП. В качестве основного параметра факторизации выбрана семантическая декомпозиция определений в рамках анализа ПП. Такой выбор позволяет выделять автономно развиваемые типовые компоненты СП, которые могут быть приспособлены к конструированию различных информационных систем. Многие работы по методам разработки программных систем зависят от практичности подходов к декомпозиции программ, что можно рассматривать как проблему факторизации программ на базе ЯП, отлаживаемых с помощью СП. Решение этой проблемы полезно для сравнения ПП, потенциала используемых схем и моделей, оценки уровня новизны создаваемых ЯП, а также при выборе критериев декомпозиции программ. Кроме того, их существование позволяет формировать методику обучения разработке компонентов информационных систем. Попутно показана дистанция в понятийной сложности между программированием и разработкой СП.

Долгоживущие программы, такие как СП, требуют объективных критериев декомпозиции программ, отражающих возможность автономного развития выделенных компонентов СП с целью профилактики повторного программирования при развитии реализуемого ЯП. Предлагается ряд решений для формализованного представления метрического пространства, полезного при измерении понятийной сложности конструкций, поддержанных в определениях ЯиСП. Такое пространство может быть применено при сравнении ПП, потенциала используемых при разработке программ схем и моделей, оценки уровня новизны создаваемых ЯП, а также при выборе критериев декомпозиции программ.

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

Алгебраическая система — это < A, F >, где:
A — основное множество значений, возможно бесконечное,
F — конечный набор операций над множеством A.

Примерно так можно формализовать библиотечные модули в СП и классы объектов в ООП до тех пор, пока при реализации ЯП не возникает необходимость использовать разные модели вычислений по одним и тем же формулам. С. С. Лавров предложил понятие «семантическая система», расширяющее понятие «алгебраическая система» заданием явного правила применения операций к данным, включая кумулятивные (накопительные) эффекты между составляющими системы.

Семантическая система — это < **V, F, R** >, где:
V — основное множество данных, возможно бесконечное,
F — расширяемый набор операций, возможно принадлежащих основному множеству,
R — правило применения операций к данным, возможно входящее в **F**.

Правила применения одинаковых операций могут существенно отличаться в разных ЯиСП. Это могут быть функции, допускающие произвольный метод отображения аргументов в результаты; процедуры, выполняющие вычисления как последовательность изменения состояний памяти; клаузы, определяющие логический вывод целей из комплекта доступных фактов, образцов и функций фиксированной арности; методы, наследуемые по иерархии областей видимости хранимых объектов; потоки, наполняемые перемещаемыми фрагментами, пригодными для выполнения на разных процессорах; макросы, работающие как открытые подстановки параметров любой природы. Кроме того, семантические системы могут различаться по отношению к областям видимости программируемых операций на статику представления определений и динамику вызовов.

Близкое понятие под названием «монады» введено в язык Haskell.

Большинство ЯП поддерживает разные варианты утончения понятия «семантическая система», например, где:

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

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

Соответствующее обозначение такого вида семантических систем можно представить как < **V; F; R** >. Формат < V; F; R > характерен для ЯП, обладающих чётким разделением данных и программ, а также для языков управления базами данных, ограничивающих кумулятивные эффекты.

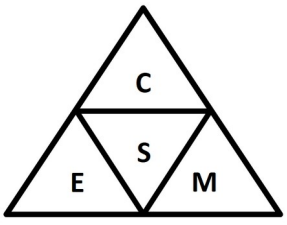
Обратите внимание на различия в записи (запятые и точки с запятыми):

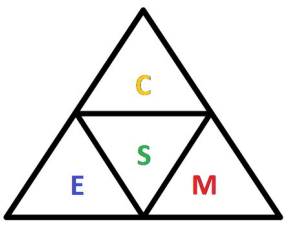
< V; F; R >	Строгое разделение значений и функций	Pascal
< V, F; R >	Разделены функции и правила применения функций	C++
< V; F, R >	Разделены значения и функции	Forth
< V, F, R >	Функции могут быть значениями, а правила — функциями	Lisp

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

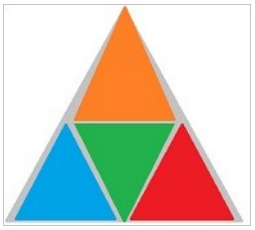
2.5. **Архитектурные модели**

При определении семантики ЯП складывается представление абстрактной или виртуальной машины, отражающее архитектурные особенности предполагаемых конкретных машин. Наиболее известные компьютерные архитектуры обычно представляют как конструкцию из вычислительного устройства, памяти, устройства управления процессами и средств коммуникации между устройствами и их элементами. Такие устройства поддерживают работу механизмов вычислений или обработки данных (E — Eval), размещения данных в памяти (M — Memory), управления вычисления (C — Control) и комплексации данных для их совместной или серийной обработки (S — Structures).





Для наглядности можно обозначения устройств выделить цветом:



E — управление процессами вычислений

M — объявление памяти и размещение данных

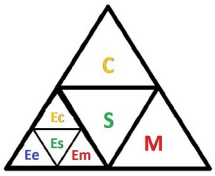
C — вычислительное устройство

S — структуры данных или средства коммуникации между устройствами и их элементами

Каждое из устройств может обладать достаточно сложной реализацией и комплекс таких устройств может быть частью более общего комплекса из периферийных устройств, поэтому, чисто образно, такую машину можно представить рекурсивным равносторонним треугольником, которых можно детализировать внутрь и укрупнять наружу. Такая конструкция допускает детализациею на уровне более тонких аппаратных решений и расширение на уровне периферийных устройств, что даёт первое приближение для выбора основных видов семантических систем в ЯП

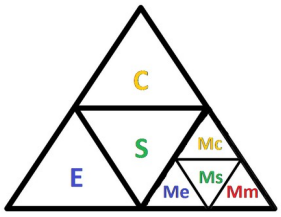
Детализация вычислений

Ee — битовые операции,
Em — номера разрядов,
Ec — потактовое управление,
Es — неявные регистры.



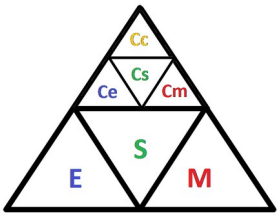
Детализация памяти

Me — адресная арифметика,
Mm — адреса и регистры
Mc — границы доступа
Ms — блоки



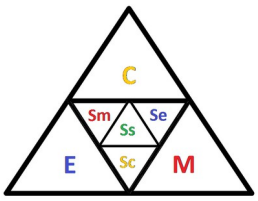
Детализация управления

Ce — условия исполнения,
Cm — метки,
Cc — текущий указатель программы, переходы,
Cs — очерёдность.

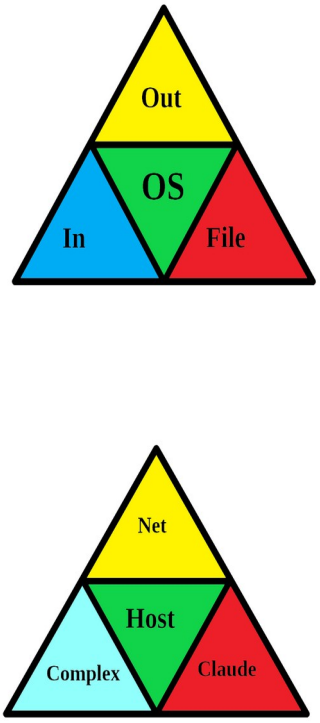


Детализация средств коммуникации

Se — сигналы,
Sm — шкала прерываний,
Sc — обработчики сигналов,
Ss — шины передачи данных.



Обобщение на средства ввода-вывода.

In – Средства ввода данных. Out – Средства вывода данных. File – Средства хранения данных. Можно и дальше наращивать треугольник, символизируя доступ к сети, облачному хранению, комплексам процессоров и т. д. Появление многопроцессорных комплексов влияет не только на стиль мышления, но на семантику и прагматику ЯП. Вместо АМ приходится определять абстрактный комплекс (АК), команды которого представляют возможности взаимодействия процессов. Это можно рассматривать как основание рассматривать ПВ равноправно с базовыми парадигмами. **Обобщение на средства работы в сети.** **Complex** – многопроцессорность. **Net** – сервера и распределённые системы. **Cloud** – облачные хранилища, репозитории.	

Рассматривая любые семантические системы, важно отметить разницу в характере выполнения функций таких систем. Так для любого множества значений V реализационно различимы виды функций для методов вычислений FE: (V* → V+)[79], средств доступа к памяти FM: (T : N → V), особенностей управления вычислениями FC: (F → {0, 1})* и обратимой комплексации и структурирования данных FS: (A → K) U (A ← K) = (A ↔ K).

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

Таблица 2.9

Шкала классов семантических систем

[79] V* - произвольное число вхождение , V+ - хотя бы одно вхождение

Горизонтальная шкала: функции	V	E	M	C	S,
Вертикальная шкала: диалекты	V	FE: V* → V+	FM: (T : A → V)	FC: (F → {0,1})*	FS: A ↔ K,

2.6. Понятийная матрица

Одним и тем же наборам функций могут соответствовать разные правила **R**, определяющие методы вычислений, влияющие на результативность выполнения программ и на продуктивность программирования. Для ассоциации с ролью правил в процессе разработки и отладки программ иногда будем использовать дополнительные, альтернативные обозначения строк в понятийной матрице, чтобы чётче отличать обозначения строк от столбцов:

B — **B**ase — базис, kernel, core — ядро (E). Это обычные арифметические вычисления, отображающие произвольный ряд **значений** аргументов в хотя бы один результат — непосредственное применение действий.

X — practical e**X**tension — расширения, макросы (M). Здесь символьные вычисления, подставляющие **представления** аргументов, возможно без их предварительного вычисления — расширение или укрупнение используемых средств.

D — **D**iagnostics — boundary — границы вычислений, исключения (C). Сюда относятся частичные или смешанные вычисления, лавирующие между вычислением и подстановкой в зависимости от разных условий — ограничения, прерывания и диагностика.

P — **P**roductivity — univers — всеобщность, внешний мир, устройства (S). Так включаются обобщённые и параллельные вычисления, оперирующие организацией процессов как множеством потоков над комплексами из разных устройств — продуктивность и производительность.

Представление таких различий можно выразить дополнив горизонтальную шкалу видов функций вертикальной кумулятивной шкалой моделей вычислений или методов применения функций к значениям (R), возможно отражающий фазы изучения ЯП — диалекты ЯП, что будет выглядеть как матрица семантических систем — понятийная матрица. Различия классов семантических систем на уровне правил применения функций к значениям в ЯВУ выражены таблицами 2.10-2.15. Первая строка этой матрицы представляет уровень базовой семантики ЯП, существенные различия видов функций семантических систем которого можно представить Таблицей 2.10.

Таблица 2.10

Семантическая декомпозиция минимального ядра ЯП

Горизонтальная шкала: функции	V	E	M	C	S,
Вертикальная шкала: диалекты	V	FE: V* → V+	FM: (T : A → V)	FC: (F → {0!1})*	FS: A ↔ K,
RE: Базис	Знач	Операции			

Базис — семантическая основа, поддерживающая начальное ознакомление с идеями вычислений на ЯП — стартовый диалект ЯП. Полное определение ЯП можно получать как консервативное расширение базиса. Обычно базис приспособлен и к неконсервативному расширению пополнением набора библиотечных функций, реализуемых на уровне аппаратуры или СП без нарушения идей ЯП. Это позволяет в реализации СП для любого ЯП поддерживать разные ПП, необходимые для поддержки ПЖЦП, чтобы достигать результат независимо от исходных возможностей ЯП.

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

Операции — минимальный комплект функций для обработки значений.

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

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

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

Вторая строка Таблицы 2.11 представляет уровень макрорасширений, что вместе с первой строкой позволяет характеризовать средства языков низкого уровня.

Таблица 2.11

Семантическая декомпозиция макрорасширения ядра ЯП

Горизонтальная шкала: функции	**V**	E	M	C	S,
Вертикальная шкала: диалекты	V	FE: V* → V+	FM: (T : A → V)	FC: {F → {0!1}}	FS: A ↔ K,
RE: Базис		Операции			
RM: Макро			Задание		

Макро — пополнение ядра средствами обработки представлений, используемых с целью укрупнения любых конструкций, что позволяет выполнять консервативное расширение ЯП. Кроме того, оно способствует лаконизму текстов программ. Макротехника позволяет наследовать отлаженность фрагментов программ. Бывает важным исключать дубли частей текста, кода и структур данных. Простейший механизм макрогенерации обычно присутствует в СП как препроцессор. Также бывает устроена техника кодогенерации и обработки шаблонов при компиляции программ. Реализация укрупнений может быть функционально эквивалентна вызову подпрограмм и обработке иерархии структур данных. Взаимозаменяемость макроподстановки и вызова подпрограмм нередко используется при оптимизации программ. Этот уровень можно рассматривать как диалект ЯП для перехода к процедурному или функциональному программированию.

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

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

Задание — хранимое именованное выражение с возможностью многократного выполнения.

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

Стек — схема организации данных, с определённой дисциплиной доступа для поддержки иерархии, возможно с защитой независимых блоков. Повышение уровня ЯП обеспечивается не только особым вниманием к средствам укрупнения данных на базе понятия «иерархия» и оперирование блоками программы. Дальнейшее наращивание объёмов разрабатываемых программ отчасти достигается автоматизацией контроля некоторых условий корректности применения операций и функций к их операндам в определённых границах. Становятся важными понятия «предикат» и «тип переменных», удобно проверяемые при компиляции, что представлено третьей строкой Таблицы 2.12.

Таблица 2.12

Семантическая декомпозиция диагностического дополнения ядра ЯП

Горизонтальная шкала: функции	**V**	**E**	**M**	**C**	**S,**
Вертикальная шкала: диалекты	V	FE: V* → V+	FM: (T : A → V)	FC: {F → {0!1}} FS: A ↔ K,	
RE: Базис	Значение	Операции	Память	Управление	Вектор,
RM: Макро	Данное	Функции	Задание	Блоки	Стек,
RC: Границы	Исключения	Предикаты	Типы переменных	Логики	Варианты,

представлено третьей строкой Таблицы 2.12.

Границы — методы проверки вычислимости функций и выполнимости заданий. Цель представления границ — снижение трудоёмкости отладки программ упрощением поиска ошибок. При отсутствии ошибок проверка воспринимается как накладные расходы. Встречаются механизмы установки ловушек на непредусмотренные ситуации и программирования обработки исключений с возможностью продолжения вычислений. Этот уровень можно рассматривать как диалект ЯП для представления частичных функций и программирования в ограничениях.

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

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

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

Логики — проверка соответствия типа данных или значений операциям обработки значений или доступа к памяти, возможно с учётом условий вычислимости и обработчиков прерываний или ловушек исключительных ситуаций..

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

По мере расширения границ программа может стать достаточно универсальной, способной к разумному поведению на любых входных данных, что выражено в Таблице 2.13, в которой добавлена слева колонка с обозначениями строк полученной понятийной матрицы, соответствующей основным диалектам ЯП. \

равноправных элементов или блоков. Полезно при отладке программ как механизм удостоверения принципиальной выполнимости вычислений при частичной постановке задачи.

По мере расширения границ программа может стать достаточно универсальной, способной к разумному поведению на любых входных данных, что выражено в Таблице 2.13, в которой добавлена слева колонка с обозначениями строк полученной понятийной матрицы, соответствующей основным диалектам ЯП. \

Таблица 2.13

Понятийная матрица: Семантическая декомпозиция практичного обобщения ядра ЯП

№	Горизонтальная шкала	V	E	M	C	S
правила	Вертикальная шкала: диалекты	V	FE: V* → V+	FM: (T : A → V)	FC: {F → {0,1}}	FS: A ↔ K
B = E	RE: Базис		рации			
X = M	RM: Макро			Задание		
D = C	RC: Границы				Логики	
P = S	RS: Общность					Ввод-вывод

Общность — дополнительные средства обеспечения отладки и применения программ, поддерживающие возможность разумного продолжения вычислений при любых исходных данных и аварийных ситуациях. Так выстраивается диалект ЯП, достаточный для отладки программ и поддержки ПЖЦП.

Неопределённость — вводятся специальные дополнительные значения и ловушки (⊥, Error, Future, откладывание, обещания,ожидания). Такое расширение множества значений позволяет учитывать в текстах программ некоторые отдельные особенности процесса разработки, отладки и схемы ПЖЦП.

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

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

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

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

Таким образом, разным видам функций относительно схемы применения функций к аргументам на уровне ЯП при реализации СП соответствуют определённые позиции специальной понятийной матрицы, полученной как двумерная шкала. Возможен кумулятивный эффект по второму измерению получается совмещением результатов ячейки [E,S] с [M,V], [M,S] с [C,V], [C,S] с [S,V] (или [B,S] с [X,V], [X,S] с [D,V], [D,S] с [P,V] в альтернативных обозначения). Более подробные пояснения даны в таблице 2.14, показывающей условность разделения реализационных понятий.

Таблица 2.14

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

Индекс	***Пояснение***
EV BV	Само-определяемые скаляры, их смысл не требует интерпретации, текстовое представление понятно без комментариев.
EE BE	Операции над скалярами, они должны вырабатывать скаляры. При выходе за границы V могут его пополнять или вырабатывать сигнал неуспеха. Множество значений может быть пополнено представлениями формул.

Индекс	*Пояснение*
EM **BM**	Обработка памяти над таблицей адресов с соответствующими им значениями. Адреса и таблица могут быть значениями или не рассматриваться как значения и использоваться неявно как сущности другой природы. Появляется именование значений и формул, что расширяет класс допустимых формул, обеспечивает многократное использование хранимых результатов и приводит к понятию «данное».
EC **BC**	Безусловное и условное (без «else») управление процессом вычислений, приоритеты в формулах вычисления и переход к очередному действию или по метке.
ES **BS**	Конструирование последовательностей по принципу соседства и перебора слева направо дополняются возможностью возвратов или выбора любого элемента ради обратимости. Как правило это вектора или строки.
MV **XV**	Хранимые данные могут иметь имена, что позволяет решать проблемы укрупнения используемых единиц, воспринимаемых равноправно со скалярами и операциями. Возникают имена логических значений для представления условий выбора хода процесса.
ME **XE**	Композиции операций рассматриваются как безымянные функции, равноправные базовым операциям, они могут обрабатывать и вырабатывать не только скаляры, но любые определённые данные, используя упаковку ряда значений в последовательность.
MM **XM**	Именование функций упрощает их многократное использование в формулах, включая представление рекурсии.
MC **XC**	Композиции операций и функций можно рассматривать как одно действие, что приводит к понятию «блок», выделенный скобками, влияющими на порядок вычислений и задающими области видимости имён — иерархия. Появляются системные процедуры, что может приводить к неконсервативным расширениям.
MS **XS**	Структуры однородных данных и процессов сопровождаются дисциплиной доступа к элементам — FIFO, FILO, взаимоисключение, одновременность или др.
CV **DV**	Множество значений пополняется специальными представлениями сигналов «успех-провал» процесса независимо от существования логических значений, а также текстами диагностических сообщений.
CE **DE**	Появляется понятие «предикат» и специальная функция ERROR для выбора обработчиков диагностических ситуаций и продолжения недоопределённых вычислений.
CM **DM**	Появляется типизация значений и данных, а также сигнатур операций и функций, используемая для профилактики неудачных вычислений.
CC **DC**	Вводятся специальные схемы управления вычислениями на основе проверки условий соответствия данных и действий для их обработки.
CS **DS**	Появляются структуры с равноправным доступом к разнородным элементам, возможно с их разметкой и указанием кратности вхождения.
SV **PV**	Вводятся мультизначения и специальные значения для представления разного рода неопределённостей, раскрытие или игнорирование которых может быть полезно для продолжения вычислений.
SE **PE**	Появляются операции над произвольным числом параметров, их распространение на любые структуры данных, проекции формул, отложенные вычисления, формулы доопределения и продолжения вычислений.
SM **PM**	Понятие присваивания распространяется на обмен данными с периферийными устройствами. Возникает идентификация устройств, абстрактные и конкретные имена, протоклы взаимодействий, сигналы готовности действий, время ожидания отклика от устройства, копии и многое другое, отражающее специфику разного оборудования.
SC **PC**	Возникают отображения, средства ввода-вывода, сетевое управление, потоки, эстафета обслуживания, итерирование, условия срабатывания, актив-пассив, администрирование, сервер, ОС.
SS **PS**	одновременность, синхронизация, взаимоисключения, сигналы и сообщения, пакеты, натройки, сервисы, конфигурации.

2.7. Пример представления результатов сравнения ЯП

Результаты анализа языка Pure Lisp представлены в Таблице 2.15, а языка Pascal — в Таблице 2.16. В клетках таблиц расположены фрагменты текстов программ на анализируемых языках или обозначения фрагментов или понятий языка.

Понятийная матрица позволяет охарактеризовать средства языка Pure Lisp в рамках семантической системы вида < V, F, R >, отображаемой в абстрактную машину secd, обладающую независимыми регистрами. Функции такой системы могут принадлежать основному множеству, правило применения функций может входить в набор функций (см. Таблицу 2.15).

Понятийная матрица, характеризующая средства языка Pascal, обладает меньшей кумулятивностью. Pascal определён в рамках семантической системы вида <V; F; R>, отображаемой в P-код , равномощный secm, и в отличие от secd, регистры sec размещены в общей памяти M. Составляющие такой системы строго разделены. Множество V фиксировано по составу, причём, обычно конечно, неявное определение R не допускает модификаций, функции могут выполнять роль данных в терминах указателей, связь с внешним миром определяется вне языка через библиотечные модули (см. Таблицу 2.16).

Таблица 2.15

Понятийная матрица определения языка Pure Lisp

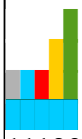
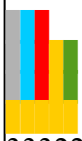
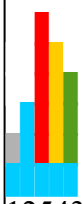
№ столбца		Категория функции	V	E
строки	Схема	Виды функций	V	F: V* → V
E=B	 11123	**Ядро**	Атомы	eq: VxV → {NIL,T}
M=X	22222	**Специальные функции**	S-выражения: точечная пара, список	`(lambda (x1 x2 ...) form)` `(label FN F)`
C=D	 33322	**Границы** **Диагностика**	NIL «Нет переменной» «Нет функции»	atom: V → {NIL,T} apply: F x V* x AL → V `ERROR`
S=P	23232	**Общность** **Практичность**	Числа Строки	evlis : (form*) → V* evcon : ((form form)*) → V **+ - * /**

Таблица 2.16

Понятийная матрица определения языка Pascal

№ столбца		Категория функции	V	E
строки	Схема	Виды функций	V	F: V* → V

E=B	 12543	**Ядро**	`Type = (a, b, ...)`	`= <>`
				`pred`
				`succ`

M=X		**Расширение**	`false` `true`	`+ - * /` `div mod`

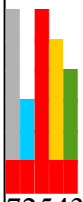
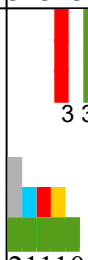
			maxint integer (1,3,6,9, ...) const A=1; B=2; type ta = 1..10	F(...)
C=D		**Границы** **Диагностика**	«Нет определения» “Несоответсвие ТД” “Выход из диапазона”	= < > >= <= in not or and Булевозначные функции
S=P		**Практичность**	Указатель: Nil X^	Функция-параметр

Таблица 2.15

Понятийная матрица определения языка Pure Lisp

M	**C**	**S**
AL : Atom → V	(F → **{NIL, T})***	A ↔ K
AL = ((NIL . NIL) (T . T))	eval: Sexpr x AL → V quote	cons: V1 V2 → (V1 . V2) car: (V1 . V2) → V1 cdr: (V1 . V2) → V2
pairlis: X xY → AL assoc: X x AL → V	((lambda (x1 x2 ...) form) v1 v2 …) ((label FN F) …)	list Список (F v1 v2 ...) Форма
null Свободные переменные Типы значений	cond FUNCTION	Строки Clozure = Funarg
Псевдо-функции Свойства	map, reduce raplca, raplcd lazy	READ PRINT

Таблица 2.16

Понятийная матрица определения языка Pascal

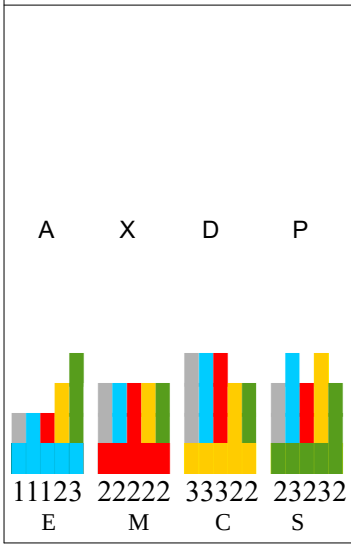
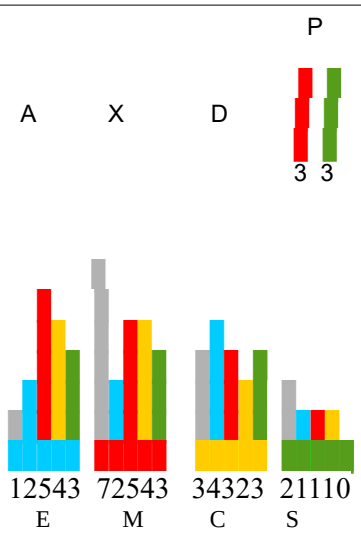
M	**C**	**S**
M : Id → V + F	(F → {True, False})*	V N ↔ Array [N] of V
Id var X : int; X := 1; label L; L: a := 1;	(1*(2+X)) S1 ; S2 Ключевые слова goto L	s: array [1..9] of int; s[5] := x; x := s[5]

M	C	S
var va : ta; begin ... begin ... end ... end func F subr S end.	if X then S1 else S2 begin S1 ; S2 ; ... end S(X, Y) forward	s: array [1..9] of array [1..3] of int; s[5,2] := x; x := s[5,2]
Типы данных: int char array ...	case Соответствие ТД	set record record … case
Процедура-параметр библиотечные модули: **new** **dispose**	while for until	библиотечные модули: **read** **write**

Такие понятийные матрицы достаточны для представления результатов сравнительного анализа ЯП и оценки сложности их применения, например, по числу различимых конструкций в каждой клетке понятийной матрицы, что можно представить как в Таблицах 2.15-2.17.

.Таблица 2.17

Визуальная форма представления результатов сравнения сложности ЯП.[80]

Lisp — 40+	Pascal — 82+
По столбцам: (8 9 8 9 9) (8 9 8 9 9)	*По столбцам:* (13 9 14+3 11 9+3) (13 9 14+3 11 9+3)
По строкам: (8 13 12 10) (8 13 12 10)	*По строкам:* (15 21 15 5+6) (15 21 15 5+6)
	

.

По таблице 2.17 сразу видно, что понятийная сложность языка Pascal выше понятийной сложности языка Lisp, в языке Pascal большое разнообразие средств работы с памятью наряду с бедностью средств взаимодействия с внешним миром. Поэтому применение языка Pascal требует привлечения системных библиотек, визуально изображённых справа сверху. Кроме того, сразу видно, что понятийная нагрузка по категориям понятий в языке Lisp распределена достаточно равномерно, в то время как в языке Pascal резкий крен в базовые средства организации данных в памяти и управление вычислениями.

[80] Нагляднее вместо букв использовать цветные прямоугольники.

2.8. Реализационная прагматика

Для оценки сложности реализации СП требуется рассмотреть более мощное пространство решений, обусловленное с одной стороны развитием аппаратуры, особенно периферийных устройств, с другой стороны проблемами выбора практичных реализационных решений из некоторого, не всегда заранее определённого, множества [6]. Кроме того, не исключены разные ПП для внешне одинаковых конструкций, что можно пронаблюдать на внешнем сходстве присваиваний в языках Pascal, Val и Sisal. Учитывая, что за время реализации ЯП его определение обычно претерпевает развитие, возникает необходимость системного обобщения, позволяющего сглаживать рост трудоёмкости варьирования реализации СП, что схематично представлено в Таблице 2.18 как реализационный куб, состоящий из четырёх слоёв, представленных понятийными матрицами с номерами **A, L, K, U** (Abstraction, Language, Kernel, Universe)**.**

Таблица 2.18

Реализационный куб понятий СП

Название слоя			**№ слоя**	**Спецификация семантических систем слоя**			
Универсальный слой 4			**U**	{РП} ↔ {КЯ↔АС↔АМ↔КМ}			
Операционный слой 3		**K**	АМ → КМ				
Изобразительный слой 2		**L**	КЯ ↔ АС				
Абстрактный слой 1	**A**	АС → АМ					
№ строки	**№ столбца**	**V**	. E	. M	. C	. S,	
	Виды функций	V	V* → V+	T: N → V	(F → {0, 1})*	A ↔ K	
B	Вычисления	Элементарные данные	Операции	Память Переменные	Управление	Структуры	
X	Укрупнения	Составные данные	Определения	Параметры	Блоки	Иерархия Стек	
D	Границы	Диагностика Исключения	Предикаты Прерывания	Типы пере-менных	Логика	Комплекты	
P	Практичность	Внешние данные	Библиотеки функций/процедур	Устройства	Интерфейсы	Ввод-вывод	

Следует отметить, что раскладка понятий на разных слоях может отличаться, это требует определения соответствий между ними. Такие реализационные кубы достаточны для представления результатов парадигмального анализа ЯиСП, а также оценки сложности их разработки и реализации, например, по числу различимых средств на каждом слое куба в отличие от сложности применения, что суммарно соответствует мере в виде кубика Рубика 4*4*4 [6]. Это значит, что от рассмотрения куба возможен переход к матрице, а от матриц можно перейти к векторам, подобно решениям по обработке многомерных структур в языках Fortran, APL, БАРС и Sisal.

Варианты инструментальной поддержки работ производственными системами реализации ЯП, схематически представленные в Приложении 2, показывают тенденцию к сосредоточению активности на уровне изобразительного слоя, обычно поддержанной средствами автоматизации синтаксически управляемого конструирования анализаторов, что делает их весьма продуктивными.

Можно обратить внимание, что изменение изобразительных средств или пользовательских интерфейсов без изменения семантики и прагматики ЯП не приводит к повышению производительности или эффективности программ и не всегда даёт повышение продуктивности программирования. Учитывая, что первичное обучение в 3-5 раз эффективнее повторного, можно сделать вывод, что эта тенденция даёт фору новичкам и тем самым решает проблему смены поколений программистов. Более заметно такой ход развития ЯП объясняется двумя причинами. Технологии программирования обязаны гарантировать продуктивность производства при выполнении определённых требований к оборудованию и квалификации исполнителей, что обеспечивается наследованием семантики и прагматики ЯП. При этом происходит смена поколений программистов, каждое поколение формирует свой лексикон, который для технологий важно связывать с готовыми решениями уровня семантики и прагматики. Продуктивность слабо зависит от синтаксиса, его изменение безопасно для сроков выполнения работ.

Дальнейшее развитие пространства решений при реализации ЯиСП направлены на инструментальную поддержку процесса разработки СП и схем, используемых при проектно-модельной подготовке этого процесса, выборе основных технологий программирования и расширении сферы приложения. Нередко такие работы имеют статус внутреннего заказа.

Технологическая поддержка (преобразования программ, факторизация, масштабирование относительно аппаратуры, вывод моделей и верификация, реорганизация и настраиваемость).

Человеческий фактор (индивидуальная разработка, группы, команды, фирмы-проекты).

Практический каркас (изученность задачи, полный жизненный цикл программ, отладка, работоспособность программы, тесты, улучшаемость, уточнение, утончение).

Эксплуатационная карта СП — уровень пользователя (интерфейс, образцы сценариев и данных, учебные упражнения и задачи с решениями в виде отлаженных программ, лексикон и руководства).

Инструментальное окружение (рабочее место системного программиста — БНФ, навигация-декомпозиция, интеграция-библиотеки компонент, систематизированных по критериям парадигмальной декомпозиции).

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

Современные работы по проверке доказательств теорем используют понятие «λ-куб», близкое понятию «семантическая система» вида $< V; F; R >$ без кумулятивного эффекта по горизонтали, дополненному учётом полиморфизма основных множеств. В таких работах декларируется эквивалентность понятий «доказательство» и «программа», но в рамках чисто функционального программирования, как в языке Haskell, дающем приоритет константным вычислениям.

Предложенную формализацию при оценке сложности и трудоёмкости программирования можно дополнить разделением требований к постановкам задач по сферам применения на академические и производственные, а по уровню изученности на чёткие, развиваемые и усложнённо трудоёмкие.

При классификации ЯиСП ключевое значение имеет семантика, но для уяснения преимуществ ПП, поддерживаемой ЯП, требуется понимание не только эксплуатационной, но и реализационной прагматики (РП), которая отчасти 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 поддерживается по разному в конкретных ПП:

ФП — списки, «сборка мусора», списки свойств атома;

ИП — побочные эффекты, векторы, ТД (переменная-значение);

ЛП — разностные списки, последовательный перебор, откатка — бэктрекинг;

ООП — ссылки, виртуальные и абстрактные методы и классы, множественное наследование.

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

Таблица 2.19

Список понятий ЯП, различаемых операционной семантикой при сравнении ПП

Понятие	Пояснение
Атом, Скаляр, Литерал	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,	Встроенная команда или подпрограмма, рассматриваемая как элементарная база

Операция	при организации вычислений.
Условие, Логика	Концепция истинностных значений может требовать как специального типа данных, так и рассматриваться как нагрузка обычных значений (0, NIL).
Функция[81]	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.
Диагностика	Варианта сообщений, стиль и лексикон.
Блок, библиотека, пакет	Место укрупнённых конструкций в механизме сборки программы.
Процесс	Понятия, отражающие динамику исполнения программы.
Ввод-вывод	Возможности управлять видом данных.
Устройство	Разнообразие использования переферийных устройств,включая сети и облака.

При неформальной характеристике стиля программирования отмечаются различия в акцентах при ответе на следующие вопросы:
1. В чем заключается основной метод обработки программы при отладке?
2. Как пронаблюдать результативную активность программы?
3. Когда принимается решение о продолжении незавершённых вычислений?
4. В каких пределах планируется функционирование участков повторяемости?
5. Каким способом гарантирована корректность сложной информационной обработки?

Варианты ответов:
1. В чем заключается основной метод обработки программы при отладке?
— Интерпретация текста программы (КС), приводящая к результату её выполнения.
— Интерпретация структуры программы (АС), приводящая к результату её выполнения.
— Компиляция текста программы (КС), 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.

2. Как пронаблюдать результативную активность программы?
— Изменение состояния отдельных элементов памяти (ИП).
— Вычисление и вывод значения выражения (ФП).
— Протокол обмена данными между программой и пользователем.
— Изображение хода вычислений в виде диаграммы.

3. Когда принимается решение о продолжении незавершённых вычислений?
— Обработка прерываний.
— Получение диагностических сообщений.
— Повторный прогон программы.
— Подготовка обработчиков прерываний.

4. 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.

5. Какие механизмы гарантируют корректность сложной информационной обработки?

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

— 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.

Описание концептуального языка или ядра ЯП, приспособленное для его сравнения с другими языками, содержит следующие части:

список общих понятий показывает уровень сопоставимости сравниваемых ЯП;

АС позволяет оценить глубину проработки используемых понятий;

АМ показывает масштаб переносимости СП;

РП дает оценку трудоёмкости реализации СП;

схема интерпретации или компиляции ЯП даёт показатель организованности процесса вычислений;

примеры программ решения типовых задач для иллюстрации концепций ЯП позволяют продемонстрировать парадигматические вариации в решении типовых задач.

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

2.9. Определитель парадигм

Первые языки программирования обладали машинной ориентированностью и поддерживали принципиально важную, но небольшую по длительности и трудозатратам часть ПЖЦП — от 2-х до 5 %, заключающуюся в кодировании готовых алгоритмов в терминах автоматов изобразительными средствами КЯ. Появление ЯВУ расширило языковое покрытие ПЖЦП примерно до 10 % для хорошо поставленных задач, имеющих алгоритмы решения над типовыми структурами данных, причём алгоритмы приспособлены для нисходящих методик программирования.

Парадигма ФП посягнула на ПЖЦП 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.

Парадигма ЛП распространила 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 ПЖЦП почти до 60 % при восходящей методике разработки.

Появление ООП смягчило итеративность ПЖЦП для задач, связанных с развивающимися областями приложения, снизило необходимость в повторном программировании, что довело языковое покрытие примерно до 80 %.

Определитель парадигмы языка программирования содержит следующие процедуры:
— Парадигмальная декомпозиция ЯП на проекции, удобные для независимого анализа;
— Разложение монопарадигмальных проекций ЯП на фрагменты по уровням или слоям с целью выделения базовых средств ЯП и его реализационного ядра — семантический базис;
— Декомпозиция семантического базиса ЯП 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 ЯП — нормализованное определение;
— Сравнение полученного определения с описаниями известных парадигм и концептуальных языков;
— Обоснование выводов относительно парадигмы, поддержанной исследуемым ЯП;
— Фразеологический словарь ПП, используемых при определении ЯП;
— Определение уровня абстрагирования и степени изученности языка, оценка сложности его понятий, его ниши в ПЖЦП и деятельности программистов (цели и задачи), а также базовых языков и ПП, повлиявших на его определение, использованных при его создании и реализации, как основы для рекомендаций по выбору и применению ЯП и его СП.

Примеры представления результатов парадигмального анализа ЯП можно выразить в табличной форме.

Таблица 2.20

Парадигмальная характеристика Pure Lisp

Параметр	Описание
Стартовая проблема	Расширение границ вычислимости на новые задачи, обладающие исследовательским компонентом.
Эксплуатационная прагматика ЯП[82]	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	Все понятия сведены 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, LABEL/DEFUN.
Семантические расширения[83]	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 абтрактной 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 ветвлений и функций, восстановление контекста, поддержка рекурсии.
Реализационная прагматика СП[84]	Списки из бинарных узлов, содержащих пару тэгированных указателей. Тэг показывает тип данного, адресуемого указателем. Автоматизировано освобождение памяти служебной программой «Сборщик мусора».
Парадигмальная специфика ЯСП [85]	Исторически основополагающий классический язык, поддерживающий функциональное программирование в полном объёме.

Ещё один пример такой же компактной парадигмальной характеристики.

Таблица 2.21

Парадигмальная характеристика ядра языка Pascal

Параметр	Конкретика
Стартовая проблема	Снижение стартового барьера при обучении программированию на задачах, имеющих алгоритмы решения.
Эксплуатационная прагматика ЯП	Язык учебного назначения, созданный для обучения студентов методам программирования решений задач, готовых для эффективной реализации при поддержке автоматным моделированием.
Особенности системы понятий	Программа и данные — 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.

[82] Уровень языка, его ниша в ПЖЦП, соответствующая технология.

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

[84] Структуры данных и традиционно подразумеваемые механизмы реализации

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

Базовые средства ЯП.	TRUE, FALSE, перечислимые значения, :=, a[i], IF, WHILE, PROCEDURE, st1 ; st2 .
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 СП	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.
Парадигмаальная специфика ЯСП	Прецедент строго определения языка программирования, обладающего не слишком высокой сложностью в реализации, поддерживающего методику результативного структурного программирования. Может выполнять роль эталонного монопарадигматического языка при сравнительном анализе языков и определении их парадигматической характеристики.

Средства и методы программирования складывались на фоне быстрого расширения сферы применения компьютерных технологий и стремительного взлёта эффективности элементной базы. Наряду с расслоением ПП в зависимости от глубины и общности технических решений по организации процессов обработки данных происходит их интеграция в рамках новых ЯП, всё более полно поддерживающих ПЖЦП. В результате обнаружились явные сложности классификации ЯП и определения их связи с конкретной ПП. Рассмотрение технических особенностей ЯиСП через их парадигмальную характеристику и понятийные таблицы может дать достаточно лаконичные формы их определения относительно ряда простых концептуальных языков и составления номенклатуры типовых семантических систем, определяющих семантику и прагматику современных ЯиСП. Следует обратить внимание, что большинство парадигм являются производными от базовых парадигм, отличаются определённым изменением их отдельных свойств. Краткая характеристика ПП может быть представлена следующим образом:
- схема постановки задач, успешно решаемых в рамках ПП,
- структура используемых аппаратных моделей,
- последовательность приоритетов в принятии решений по подготовке программ,
- кумулятивные эффекты в определениях семантических систем,
- формат понятийных таблиц, определяющих семантику ЯП, поддерживающих ПП,
- семантические и прагматические принципы реализации ЯиСП, поддерживающих ПП.

Таблица 2.22
Паспорта ПП на примерах языков Pure Lisp и Pascal

	ФП: Pure Lisp	ИП: Pascal	*Примечание*
Причина создания ЯП	Расширение границ вычислимости на новые задачи, обладающие исследовательским компонентом.	Снижение барьера при преподавании программированию на задачах, имеющих алгоритмы решения.	*Стартовая проблема различается по целям*
Типовая задача	Представить и отладить алгоритм решения задачи.	Запрограммировать алгоритмическое решение задачи.	*Отличаются степени изученности*
Формат семантических систем	**< V, F, R >** Функции могут быть значениями, а правила — функциями	**< V; F; R >** Строгое разделение значений, функций и правил	*Накопительные эффекты различаются по полноте решений.*
Особенности правила применения операций к дан-	Кроме обычных функций с пред-вычислением аргументов имеются пост-функции, работающие как	Пред-вычисление аргументов при фиксированных схемах управления	*Полнота или границы возможных моделей вычислений*

ным	программируемые схемы управления или макросы		
Приоритеты	ESMC Вычисления; структуры; имена; управление	MCES Распределение памяти; логика управления; вычисления; структуры	*Последовательность принятия решений в разных направлениях.*
АМ	**secd** Бесконечные регистры с защитой неизменяемых данных	**secm** Общая память ограниченного объёма без защиты данных от изменения	*Архитектурная модель для ФП гибкая, для ИП жёсткая.*
АС	АС = Lisp Программа представлена как структура данных языка, допускающая обработку при выполнении	Pascal → АС Программа переводится на внутренний, промежуточный язык, при выполнении недоступный	*Системная модель разной сложности.*
Диалекты	BXDP Базис; расширения; частичность; устройства	PXDB Устройства; библиотеки; типы данных; базис	*Разные методики взятия стартового барьера.*

Выводы

1. Поддержка ПП при определении ЯП и разработке СП выражается в реализации средств организации вычислений, механизмов обработки параметров, использования СД и размещения СД в памяти, методов контроля за вычислениями и управления ходом вычислений. Успех применения ПП зависит от того, в какой мере используемые ЯиСП 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 диалекты 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.
4. Парадигмальная специфика ЯиСП представляет собой взаимодействие эксплуатационной прагматики ЯП и реализационной прагматики СП, выраженное в особенностях системы понятий, распознаваемых на уровне абстрактного синтаксиса ЯП с выделенными базовыми средствами и семантическими расширениями, дополненными системой команд АМ над её регистрами (инвариант ЯП).
5. Использование промежуточных языков при реализации ЯП следует сопровождать доказательством эквивалентности разных представлений программ.
6. Часто выделяют базовые, фундаментальные и основные ПП. Кроме того, многие ПП бывают производными от базовых, комбинированными, дополнительными, вспомогательными, а также специальными, ориентированными на конкретный класс задач.

Практика 2. Семантический базис знакомого ЯП или его части

Краткое описание знакомого ЯП, опыта работы с ним, его особенностей, поддерживающих технологий, достоинств и недостатков (трудоёмкость 4 часа).

2.1. Опишите привлекательные стороны знакомого ЯП, можно выбрать из списка (CGI, C-lang LLVM D Java, JavaScript, ECMAScript, Kotlin JS++, mpC Ластовецкого Eiffel, Go Golang Scala, **Self**)

2.2. Отметьте особенности знакомого ЯП, иногда провоцирующие трудно обнаруживаемые ошибки.

2.3. Опишите задачи, при решении которых предпочтительно применять знакомый ЯП.

ЛЕКЦИЯ 3. ЯЗЫКИ НИЗКОГО УРОВНЯ.

Методика парадигмальной характеристики ЯП в данной лекции иллюстрируется на материале языков низкого уровня (ЯНУ), обычно поддерживающих одну ПП. Программирование или кодирование на ЯНУ ассоциируется с одноуровневыми структурами данных, обусловленными архитектурой и оборудованием[86]. Монопарадигмальность ЯНУ делает их удобными для изучение специфики ПП.

3.0. Компьютер и внешний мир

Применение ЯНУ интересно тем, что при их использовании в принципе достижима предельная эффективность результата, но его получение осложнено сочетанием «низкий старт — высокий финиш». Иными словами, легко достичь успеха в первых упражнениях, но трудно создать программный продукт и обеспечить его квалифицированное сопровождение. Для ЯНУ характерна однозначность соответствия между программой и процессом, порождаемым при ее реализации. При хранении данных и программ используется общая глобальная память с произвольным доступом. Поэтому анализ операционной семантики ЯНУ можно выполнить на уровне абстрактной машины (АМ), вполне определяющей свойства программ и процессов, подготовленных с помощью ЯНУ. Как правило, при определении абстрактной машины ЯНУ достаточно трех регистров, назначение которых соответствует реализации понятий «результат», «программа», «память» или «результат», «контекст», «программа».

Традиционно к ЯНУ относят машинно-зависимые языки ассемблера, макропроцессоры, машинно-ориентированные языки, языки управления процессами, базами данных и сайтами. Для таких ЯНУ характерно, что все действия в программе выражены явно. Программа — произвольная смесь команд, соседство которых практически не ограничивается. Доступны любые фрагменты данных и программ. Предопределены все базовые средства по представлению значений и структур данных в памяти и схема управления их обработкой, что позволяет четко относить ЯНУ к конкретной парадигме.

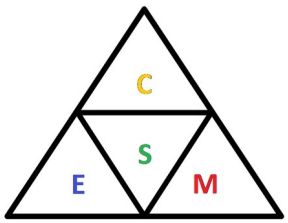
Парадигмальные различия ЯНУ обычно связаны с особенностями используемых компьютерных средств — с архитектурными моделями, что можно символизировать их треугольными диаграммами. Представляют интерес и другие подходы к машинно-ориентированному эффективному программированию. Язык Forth – пример организации вычислений над стеком. Его можно рассматривать как язык-ядро с возможностью практически неограниченного проблемно-ориентированного расширения. Обычно для решения сложных задач или задач с расширяемой постановкой выбирают небольшой семантически базис, такой что из него можно вывести всё остальное потом, после сдачи-приёмки некой базовой части. Планируется график расширения реализации до полной постановки задачи.

Операционная семантика ЯНУ обычно содержит целочисленную арифметику, ограниченную разрядностью адресов или машинных слов, использует работу с общими, глобальными идентификаторами, поддерживает последовательное управление вычислениями и осуществляет организацию структур данных по принципу соседства элементов, расположенных в памяти (вектора, строки, стеки, очереди, файлы). Примерно так определена базовая часть учебного подмножества или диалекта (элементарного уровня) любого ЯП. Другие вспомогательные семантики языков ассемблера, макропроцессора, вычислений над стеком и управления процессами, машинно-зависимых и машинно-независимых машинно-ориентированных ЯП могут рассматриваться как расширения такой базовой части. Это востребовано для СУБД, языков работы с сайтами и рядом других средств создания распределённых систем.

3.1. Продукционная макротехника

Макротехника — технически самый простой метод обработки данных произвольной природы, заключающийся в определении макросов, приспособленных к изменению окрестности их вхождения с подстановкой параметров. Средства макрообработки текстов часто встречаются как препроцессоры в ЯиСП, в языках управления заданиями, в языках переписывания и разметки текстов и в других приложениях, связанных с необходимостью выполнить преобразование данных, не достаточно удобно программируемое на привычных ЯВУ.

Лежащая в основе макротехники архитектурная модель обычно обладает определёнными ограничениями. Механизмы вычислений или обработки данных (**E** — Eval) минимизированы, обычно не выходят за пределы целочисленной арифметики, размещение результатов обработки (**M** — Memory)



[86] Ассемблер «Эльбрус» и автокод «Инженер» — контрпримеры, показывающие недостаточность чисто приаппаратной оценки уровня языка.

происходит непосредственно в памяти обрабатываемых данных, управление вычислениями (**C** — Control) ограничено простейшими условиями, а комплексация данных (**S** — Structures) сводится к формированию строк с помощью открытой подстановки параметров в шаблоны, заданные макросами.

Макротехника дает весьма мощные, но не вполне безопасные средства повышения выразительности ЯП. Отлаживать такой код, содержащий макровызовы, сложнее, кроме того, в момент подстановки параметров обычно нет никакого контроля ни типов, ни видов параметров. Если что-то не так, то это выяснится позже, когда уже трудно проследить что откуда взялось. Иногда пытаются сохранять в комментариях вид макровызова, это слегка облегчает отладку. Бывают макрогенераторы, ставящие пробелы вокруг параметров, а бывает, что сцепляют всё в целостную строку. Встречаются макрогенераторы, допускающие рекурсию и многократный прогон текста для обнаружения вторичных макросов.

Это означает, что семантические системы макрогенераторов имеют формат **< V, F; R >** - макросы и правило их применения разделены, но можно строить новые макросы в процессе макроподстановок. Не всегда анализ данных для макропреобразования происходит последовательно, что позволяет их рассматривать как предшественника логического программирования. В некоторых источниках упоминают продукционное программирование.

Для макропроцессоров семантика ЯНУ обычно сопровождает средства работы со строками в стиле открытых процедур. Программа представляет собой поток макроопределений и макровызовов. Редко представлено описание алгоритма применения макросов, чаще провозглашается, что макросы применяются в произвольном порядке, но всё-таки реально имеется некий определённый порядок их вызова, его приходится учитывать при размещении определений макросов. Это задаёт упорядочение приоритетов как CESM — сначала порядок возможных вызовов макросов и разметка границы между вычисляемыми и неизменяемыми фрагментами определений (C), затем решение о выполнимых вычислениях (E), после этого шаблоны формирования результатов (S) и именование макросов и их параметров (M).

Имена макросов внешне выглядят равноправно с базовыми средствами. При определении и реализации макросов используется понятие позиции и шаблона для подстановки параметров. Встречается стиль нумерации позиций, который позволяет обойтись без их именования как переменных. В результате подстановки макросов формируется текст, замещающий исходный, возможно обновлённый текст содержит вторичные макросы. Макропроцессор часто используется в паре с ассемблером (макроассемблер) и другими ЯП. В качестве иллюстраций здесь рассмотрены макропроцессоры GPM, TRAC, а также два макропроцессора из системы подготовки программ на языке SETL.

Для автономных макропроцессоров характерны специальные механизмы регулярного конструирования различимых текстов:

- управляющие символы;
- счетчики;
- генераторы уникальных значений;
- блоки активности (разметка);
- встроенные функции;
- управление вводом-выводом.

Для макропроцессора программа – это содержащий макровызовы текст, при обработке которого программа преобразуется в новый текст программы, полученный в результате макропреобразований. Большинство макропроцессоров допускают переменные макропериода – глобальные и локальные макропеременные.

Макро = (Прог = Текст): Прог → Прог'

Интерпретатор макропроцессора при последовательном сканировании текста выделяет в нем следующие категории строк:

- макроопределение, которое следует поместить в таблицу макросов (c → e) — контекст;
- макровызов без параметров, определение которого следует скопировать из таблицы в результат (e → c);
- макровызов с параметрами, значения которых следует установить, а затем подставить в буферную копию определения и преобразованное определение разместить в результат (e → p → c);
- простая неизменяемая строка, сохраняемая в результате без преобразований;
- конец текста.

Абстрактный синтаксис макротекстов показывает появление функций с произвольным числом параметров, открыто подставляемых в шаблон определения.

(DEF Name Arg1 Arg2 … ArgN Patern) – разделители и ограничители сняты при переходе к АС.

(Name Txt1 Txt2 … TxtN) – однократная открытая подстановка текстов в место соответствующих аргументов.

Спецификация команд абстрактной машины макропроцессора может быть задана парой <E, C>, в которой C – строка, а E – вектор определений.

Таблица 3.1

Пример системы команд макропроцессора

SCM	*Описание команды*	*Примечание*
LDF	Ввод шаблона макроса в таблицу под заданным именем.	Укрупнение действия.
LDN	Размещение параметра макроса в таблице под заданным номером.	Локальные параметры.
SP	Сцепление строки с текстом = обход строки.	Константа.
ARG	Копирование параметра в текст	Подстановка параметра.
AP	Применение макроса.	Применение макроса.

Таблица 3.2

Спецификация макрокоманд — переходы

RM	RM’	*Примечание*
e (LDF Mac Ptn . c)	→ (e|e[Mac]:=Ptn) c	Размещение макроса в таблица определений.
e (LDN Num Arg . c)	→ (e|e[Num]:=Arg) c	Размещение параметров в таблице определений.
e (SP X . c)	→ e c	Пропуск константы.
e (ARG Num . c)	→ e (e[Num] | c)	Подстановка параметра.
e (AP Mac . c)	→ e (e[Mac] | c)	Подстановка макроса.

Макропреобразования могут использовать локальные или глобальные переменные, вложенность областей действия определений, рекурсию. Макропроцессор может быть встроен в компилятор, может быть автономным инструментом системы программирования, таким как текстовый редактор, оптимизатор или отладчик, или может существовать самостоятельно как универсальный инструмент общего назначения.

Макровызовы могут выполняться за одни просмотр или до исчерпания при итеративном анализе текста. Встречаются средства управления глубиной макроподстановки. Популярно синтаксическое подобие макросов выражениям базового языка, хотя это может вызывать путаницу в понимании реальных механизмов при порождении кода программы.

Техника строковой обработки обычно поддерживается операциями вычисления длины строки, выделения подстроки и конкатенации строк.

В системах программирования макротехника применяется на двух уровнях: препроцессоры обычно формируют входной текст для компилятора, а макроассемблеры выполняют сборку кода на уровне генерации ассемблерной программы или её объектного кода.

Макропроцессор получает логическое завершение, поддерживая динамически формируемые макроопределения, возможно используя макровызовы в аргументах.

Встречается интересное применение вложенности макровызовов и макроопределений, включая рекурсию вида ФАКТ (сч) = если <сч = 0> то [1] иначе [(| сч | +] | ФАКТ [сч - 1] | [ )] .

Макроопределение	*Примечание*
ФАКТ (сч) = | <сч = 0> → [1]	Строка для значения 0
| [(| сч | *]	Строка из «(», значения «сч» и «*»
| ФАКТ [сч - 1]	Строка из макроса от «сч», уменьшенного на 1
| [ )]	Строка из «)»
ФАКТ (6) = (6 * (5 * (4 * (3 * (2 * (1 * 1))))))	Результат макроса на значении 6

Пример 3.1. Рекурсивное макроопределение факториала

Макроопределение	*Примечание*
§A, §Def, A , <D>; §Def, B, <C>;;	Вызов «A», где «A» указывает на «D», а «B» на «C»

Пример 3.2. Пример макропрограммы на GPM. Моделирование « if A=B then C else D » обеспечено побочным эффектом на глобальной таблице имен

Техника выполнения макропреобразований достаточно разнообразна. Так, например, язык GPM[87] всю работу с макросами сводит к макровызову вида:

§ mak, a1, a2, ... aN; – вызов макроса.

Позиции макровызова занумерованы по числу предшествующих запятых, что делает ненужным описание переменных и дает возможность самоприменения определений:

~0 ~1 ~2 ... ~N – описание не нужно.

Кроме того используются скобки, блокирующие подстановки при необходимости:

< S > – блокировка подстановок в S.

Кристофер Стрейчи в своё время показал, что достаточно всего одной встроенной функции DEF, выполняющей введение макроопределений, для полноты по Тьюрингу.

Макроопределение	*Примечание*
§Def, mak, опр;	Команда создания нового макроса

Пример 3.3. Введение новых макроопределений GPM

Макроопределение	*Примечание*
§Def, size, 6; §size; => 6 x (§size, §size) => x (6,6) size§size => size6	Определение макроса Варианты вызовов макроса

Пример 3.4. Использование макроопределений GPM

Макроопределение	*Примечание*
§Def, opp, UN~1; §opp, R; §Def, opp, <UN~1>; §opp, R; => UNR	Параметр «~1» вне подстановки => ОШ – нет определения Параметр может быть подставлен => UNR

Примеры 3.5. Использование блокировок в макроопределениях GPM

Совершенно иначе выглядит макротехника в не менее лаконичном языке макропроцессора TRAC. Все сводится к контекстно зависимым макровызовам функций, встроенных и определяемых:

(F, s1,s2,...,sN)

Встроенные функции F:
ds – определение строки,
cl – вызов определение,
ss – выделить сегменты,
rs – чтение строки.

Макроопределение	*Примечание*
#(ds,ПРИМЕР, собака сидит на ковре) #(ss,ПРИМЕР, собака, ковре) #(cl,ПРИМЕР, кошка, кресле) = кошка сидит на кресле	Исходная строка. Выделены замещаемые сегменты. Задана подстановка = результат.

Пример 3.6. Работа с шаблонами на языке Trac

Два интересных механизма макротехники были реализованы в проекте языка Setl при попытке его эффективной реализации посредством языка Little. Для поддержки переноса программы на разные архитектуры предлагалась специальная разметка текста с помощью флагов, в зависимости от значения которых блоки строк включались во входной текст для компилятора. Значения флагов можно было инициировать, наращивать или редуцировать и обнулять.

+ flag – включить строку;

.flag – завершение блока, сопровождается увеличением или уменьшением счетчика, одноименного с флагом;

- flag – пропустить строку.

[87] 1960 Кристофер Стрейчи

Для автоматизации формирования фрагментов текста, обладающих зависимостью от численных характеристик или кратности вхождения в программу, использовался специальный механизм специальных макропеременных.

Макроопределение	*Примечание*
zxN => N + I zyN = N' => N' (zyN := N') zaN => A(N+i)	В строке размещается значение счетчика. Задание значения спецпеременной. В строке размещается имя “A”, сцепленное со значением счетчика.

Пример 3.7. Представление зависимости от процесса формирования текста

Общеизвестно, что макрос легче определять, чем применять. Внешняя простота введения макросов сопряжена с вероятностью порождения трудно обнаруживаемых ошибок периода исполнения программы, индуцированных случайным сходством с подпрограммами на основном ЯП при существенном различии:

- макрос меняет текст программы, содержащей вызовы макроса,
- подпрограмма меняет данные программы и логику процесса исполнения программы.

Макротехника приносит результаты не только на текстах, но и на геометрических фигурах, графах и кодах. Например, макросами можно описать пентамино, оптимизацию и кодогенерацию программ.

Иногда встречаются более специализированные средства, использующие счётчиковые переменные, конструкторы уникальных имен, моделирующие иерархию модулей или параметризирующие зависимость вариантов программы от целевых архитектур.

Определения	*Примечание*
#DEFINE THEN #DEFINE BEGIN { #DEFINE END ;}	**«Синтаксический сахар»:** Можно определить ключевые слова для рамочных конструкций;

IF (I > 0) THEN
BEGIN
A = 1;
B = 2
END
	затем написать текст, похожий на Pascal-программу.

#DEFINE MAX(A, B) ((A) > (B) ? (A) : (B))	**Макрос:** Такое определение обеспечивает функцию «максимум», которая реализуется как последовательный код, а не вызов функции.
X = MAX(P+Q, R+S);	**Макровызов:** При правильном объявлении аргументов такой макрос будет работать с любыми типами данных.
X = ((P+Q) > (R+S) ? (P+Q) : (R+S));	**Результат макроподстановки:** Нет необходимости в различных видах MAX для данных разных типов, как это было бы с функциями

Пример 3.8. Пример макросов языка C

Определения	*Примечание*
(defun MF (NF AF BF) (list ’DEFUN NF AF BF))	Макрос
(eval (MF MNF (A B) (cons B A)))	Макровызов
(MNF 1 2)	Применение результата

Пример 3.9. Пример макротехники на языке Lisp

Основные отличия АМ макрогенератора от других ЯНУ, таких как АМ ассемблера или стековой машины, связаны с открытой обработке строк произвольной длины или данных произвольного размера и с использованием программируемых определений, расширяющих исходную систему команд и поддерживающих локализацию переменных. Нет акцента на обработку простых значений, размещаемых в словах фиксированного размера.

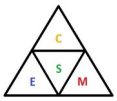
Уровень сложности используемых понятий можно характеризовать одной строкой понятийной матрицы.

Таблица 3.3.
Понятийная матриц а макротехники

V	E	M	C	S
B: Данное	Копирование	§def ИмяМак параметры	Подстановка Цикл	Шаблон

V : константы — Данное (текст, рисунок, код)
E : **V* → V+** вычисления — Копирование констант
M : **T: N → V** память — Имена макросов и параметров
C : **(F → {0,1})*** управление — Цикл и подстановка по шаблону макроса
S : **A → K** структуры — Шаблоны для подстановки

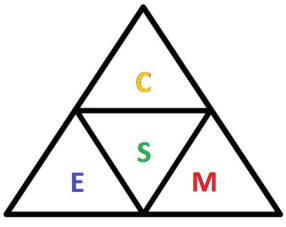
Таблица 3.4
Парадигматическая характеристика макропроцессора

Параметр	*Конкретика*
Постановки задач	Необходимость выполнить преобразование данных, трудно программируемое на привычных ЯВУ.
Ограничения архитектурной модели 	(E — Eval) обычно не выходят за пределы целочисленной арифметики, (M — Memory) непосредственно в памяти обрабатываемых данных, (C — Control) простейшие условиями, (S — Structures) открытая подстановка параметров в шаблонов, заданных макросами.
Семантические системы	**< V, F; R >** - макросы и правило их применения разделены
Упорядочение приоритетов в принятии решений	**C E S M** — порядок вызовов макросов (C), решение о подстановках (E), шаблоны формирования результатов (S) и именование макросов (M)
Понятийные таблицы	**V :** константы — Данное (текст, рисунок, код) E : **V* → V+** — Копирование констант M : **T: N → V** — Имена макросов и параметров C : **(F → {0,1})*** — Цикл и постановка макросов S : **A → K** — Шаблоны для подстановки
Эксплуатационная прагматика ЯП	Главное предназначение макросов в ЯиСП – достижение гибкости и переносимости текстов программ, применяемых в разных условиях.
Реализационная прагматика	Открытая подстановка параметров без контроля границ стыковки фрагментов.
Парадигматическая специфика	Макротехника близка продукционному и логическому стилю программирования, языкам разметки и системам переписывания и разметки текстов. Макротехника характерна для ЯВУ, поддерживающих вызовы по необходимости при организации отложенных (ленивых) вычислений.

3.2. Стековая машина Forth

Язык Forth возник как реакция на появление микропроцессоров. Постановки задач для этого языка нацелены на программирование эффективной обработки чисел фиксированного размера при использовании малого объёма памяти.

Лежащая в основе языка Forth архитектурная модель обладает конкретными ограничениями. Механизмы вычислений используют операции над числами разной величины обрабатываемого слова (16 и 32, возможно, 64) (E — Eval). Основа работы с памятью – словарь именованных операций и функций (M — Memory). Управление вычислениями обогащено средствами блокировки и кодирования программ, что позволяет повышать эффективность информационной обработки и управления вычислениями (C — Control). Комплексация данных устроена как память на основе стека (S — Structures).



В языке Forth принята постфиксная запись программ, удобная для стековой обработки данных, в которой операнды размещаются перед операцией. Используется механизм замкнутых процедур с неявными – стековыми – параметрами. Стек реализован как указатель на текущий элемент в предположении, что перед ним по порядку расположены предшествующие элементы. Программа –

отдельный поток, использующий расширяемый словарь. Поэтому семантические системы языке Forth имеют формат **< V, F; R >** - словарь и правило применения операций к стеку разделены, но можно строить новые функции в процессе вычислений.

Подготовка программ на языке Forth начинается с выбора размера обрабатываемых данных и подходящих операций (E). Затем анализируется порядок размещения обрабатываемых данных в стеке (S). После этого планируется программирование функций и их именование в словаре (M). При необходимости выполняется моделирование традиционных средств управления вычислениями и/или кодированием программы (C). Таким образом получается упорядочение приоритетов как ESMC - . Многие источники рассматривают Forth как простейший язык функционального программирования.

Стек-ориентированная дисциплина обработки данных освобождает от необходимости в понятии «переменная», хотя оно при необходимости моделируется. Программирование на языке Forth сопровождается систематической сверткой понятий, синтаксис применения которых созвучен обратной польской записи[88]. Можно сказать, что хорошая программа на языке Forth – это специализированная виртуальная машина, близкая понятию абстрактный синтаксис (АС), приспособленная к дальнейшему расширению по мере развития постановки задачи.

Интерпретатор языка Forth сортирует слова по принадлежности словарю:

- слова, не найденные в словаре, записываются в стек для предстоящей обработки;
- словарным словам, встроенным в интерпретатор, соответствует правило преобразования стека;
- возможно определение новых слов, запоминаемое в словаре (от «:» до «;»);
- за корректность воздействий на стек отвечает программа, значит — программист;
- результатом считается состояние стека при завершении программы.

Программа на языке Forth строится как последовательность слов, некоторые их которых включены в расширяемый словарь языка:

Forth: текст/словарь → стек: стек → стек'
: слово → словарь

Данные – это тоже слова. Логическое значение «истина» – 0.
Новые слова можно вводить в форме:

: имя *определение* ;

Абстрактный синтаксис языка отличается от АС макропроцессора использованием произвольного числа неявных, безымянных аргументов определяемых команд:

S E C → S' E' C'

S – стек результатов,
E – словарь, представленный как вектор строк,
C – поток слов, образующих программу.

Исполнение программы организовано как диалог над стеком. Каждая команда знает, что взять из стека, во что преобразовать для получения результата программы и какие результаты разместить в стеке.

Таблица 3.5
Команды стековой машины языка Forth

SCS	*Описание команды*	*Примечание*
LDC	Засылка из программы в стек.	Операнды.
DROP	Сбросить из стека верхний элемент.	Манипуляции с фиксированным числом элементов стека.
DUP	Скопировать верхний элемент.	
NIP	Сбросить предпоследний элемент стека.	
OVER	Предпоследний элемент переставить наверх.	
PICK	Выборка из стека элемента с указанным номером.	
SWAP	Обмен местами двух верхних элементов.	

[88] Логик Ян Лукасевич изобрёл эту запись примерно в 1920, чтобы упростить пропозициональную логику.

TUCK	Копирование верхнего элемента под второй с сохранением.	
DEPTH	Глубина стека.	
PICK	Выборка из стека элемента с указанным номером.	Манипуляции с заданным числом элементов стека.
ROLL	Перестановка на заданную глубину элемента из стека.	
?DUP	Скопировать ненулевой верхний элемент.	
LDF	Загрузить в словарь определение.	Укрупнение действий.
AP	Применить словарное определение.	

Укрупнение действий – это новые определения с неявными параметрами в стеке.

Таблица 3.6

Спецификация команд стековой машины языка Forth [89]

RS	RS	*Примечание*
s e (LDC X . c)	(X . s) e c	Подготовка стека к предстоящим операциям.
(X . s) e (DROP . c)	→ s e c	
(X . s) e (DUP . c)	→ (X X . s) e c	
(X2 X1. s) e (NIP . c)	→ (X1 . s) e c	
(X2 X1. s) e (OVER . c)	→ (X2 X1 X2 . s) e c	
(XN ... X0 N . s) e (PICK . c)	→ (XN ... X0 XN . s) e c	
(XN ... X0 N. s) e (ROLL . c)	→ (XN-1 ... X0 XN . s) e c	
(X3 X2 X1. s) e (ROT . c)	→ (X2 X1 X3 . s) e c	
(X2 X1. s) e (SWAP . c)	→ (X1 X2 . s) e c	
(X2 X1. s) e (TUCK . c)	→ (X1 X2 X1 . s) e c	
s (DEPTH . c)	→ (N . s) e c	
(0 . s) e (?DUP . c)	→ (0 . s) e c	
(X . s) e (?DUP. c)	→ (X X . s) e c	
(NF . s) e (LDF F ";" . c)	→ s (e|e[NF]=F) c	Пополнение словаря.
(NF . s) e (AP . c)	→ s e (e[NF] . c)	Применение словарного определения.

Система программирования для языка Forth содержит пару «интерпретатор – компилятор», причем техника компиляции весьма эффективна. Система использует единый порядок представления данных и команд в программе это последовательности слов. Данные располагают перед операциями по их обработке. Операция – это известное системе слово. Данные просто загружаются на стек, из которого операция берет их в соответствии с числом ее параметров. Интерпретирующий автомат для языка Forth сложнее макрогенератора, но немного проще автомата для ассемблера. От макроассемблера Forth отличается использованием такой основной структуры данных как стек, обладающей чёткой дисциплиной доступа к данным. Основные различия с ассемблером таковы:

- словарь языка Forth хранит строки произвольной длины, а таблица меток ассемблера хранит адреса фиксированного размера;
- вместо запоминания адреса возврата при организации подпрограмм/функций определения размещаются расширяемом словаре;
- неявные параметры функций заранее размещаются в стеке, их число известно;
- результаты вычислений сконцентрированы в стеке.

Таким образом, обеспечены базовые функциональные возможности, характерные для систем программирования на языках высокого уровня, обрабатывающих иерархию выражений без привязки к именам и адресам .

Фрагмент программы	*Примечание*
(x y z → x y*z)	Перемножение двух верхних элементов стека
OVER (x y*z → x y*z x)	Копирование второго элемента на верх стека.
- (x y*z x → x y*z - x)	Разность двух верхних элементов
SWAP (x y*z - x → y*z – x x)	Перестановка двух верхних элементов
DUP (y*z – x x → y*z – x x x)	Дубль верхнего элемента

[89] N – длина стека

* (y*z – x x x → y*z – x x*x)	Перемножение двух элементов
+ (y*z – x x*x → y*z – x + x*x)	Сумма

Пример 3.10. Программа на языке Forth для подсчета по формуле
(x y z → x**2 + y*z – x)

Фрагмент программы	*Примечание*
: SQ (вводится слово SQ в словарь)	Объявление имени укрупнённого действия
DUP (A B → A B B)	Дубль верхнего элемента
* SWAP (A B B → A B**2 → B**2 A)	Перемножение и перестановка
DUP * (B**2 A → B**2 A A → B**2 A**2)	Дубль и квадрат
+ (B**2 A**2 → B**2 + A**2)	Сумма – результат в стеке
; (конец определения нового слова)	

Пример 3.11.. Введение нового слова SQ для подсчета суммы квадратов
(A B → A**2 + B**2)

Для удобства программирования определений функций привлекается методика моделирования переменных и констант, а также средств управления процессами с помощью ветвления: условное выражение и переключатель, моделирование циклов.

Язык Forth – пример организации вычислений над стеком. Его можно рассматривать как язык-ядро с возможностью практически неограниченного проблемно-ориентированного расширения машинно-независимых эффективных средств программирования. Язык допускает порождение эффективного кода «хорошо» написанных программ.

Программирование на языке Forth требует вдумчивости и аккуратности. Достижимость лаконичных форм дается ценой нестандартных индивидуальных решений, мало приспособленных к передаче программ в чужие руки. Лозунги «Программируйте все сами!» и «Не бойтесь все переписывать заново!» правильно отражают подход к программированию на языке Forth. Успех достигается максимализмом в тщательной отладке и способностью видеть задачу программирования в развитии. Автор языка Forth Чарльз Маури в 1968 году отметил: «Forth не уравнитель, а усилитель!».

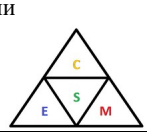
К середине 70-х Forth был третьим по популярности после Бейсика и Паскаля, завоевав свои позиции при освоении микропроцессорных средств. По технике программирования Forth похож на макроассемблер, только вместо системы команд над машинными словами в нем используется система операций на стеком. Краткая характеристика языка Forth может быть сведена к понятийной таблице из двух строк.

Таблица 3.7
Понятийная таблица языка Forth

V	**E**	**M**	**C**	**S**
B: Байт	**Операция**	**Строка**	**Символы**	**Стек**
X: НовОпер	**: ;**	**Имя Строки**	**=0**	**Состояние Стека**

V : константы — Байты и имена операций
E : **V* → V+** вычисления — Встроенные и программируемые операции
M : **T: N → V** память — Исходная строка программы и словарь именованных определений
C : **(F → {0,1})*** управление — Обнаружение операций, проверка на ноль и учёт блокировок.
S : **A → K** структуры — Стек и состояние стека

Таблица 3.8
Парадигматическая характеристика языка Forth

Параметр	*Конкретика*
Постановки задач	Эффективная обработка данных фиксированного размера при использовании малого объёма памяти.
Ограничения архитектурной модели 	(E — Eval) – операции над данными разной величины (16 и 32, возможно, 64 разряда) (M — Memory) – словарь именованных операций и функций. (C — Control) – сравнение с нулём и средства блокировки и кодирования программ. (S — Structures) – комплексация данных в стеке.

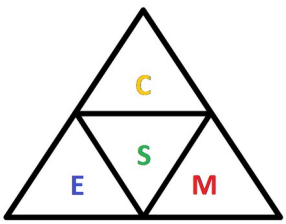
Семантические системы	**< V, F; R >** - словарь и правило применения операций разделены, можно строить свои функции.
Упорядочение приоритетов в принятии решений	**E S M C** - выбор подходящих операций (E), анализ порядка размещения данных в стеке (S), именование функций в словаре (M), управления вычислениями и/или кодированием программы (C).
Понятийные таблицы (две строки)	**V :** — Байты и имена операций E : **V* → V+** — Операции и свои функции M : **T: N → V** — Исходная строка, стек и словарь C : **(F → {0,1})*** — Проверка на ноль и блокировки S : **A → K** — Стек и изменение стека
Эксплуатационная прагматика	Forth – средство разработки специализированных систем с минимизированного ядра и рядом шагов расширения в рамках единой оболочки. Мало приспособлен к передаче программ в чужие руки. Компактный язык-ядро для решения задач с расширяемой постановкой.
Реализационная прагматика	Система программирования использует пару интерпретатор-компилятор.
Парадигматическая специфика	Механизм реализации выражений в большинстве ЯВУ. Forth обладает концептуальным родством с языком Lisp.

3.3. Императивное программирование на ассемблере

Применение ассемблера становится необходимым, когда требуется доступ ко всем возможностям компьютера, особенно за пределами RISK-архитектур, поддерживаемых ЯВУ. Обработка данных с помощью программ на ассемблере сводится к императивной машинно-ориентированной модели управления процессом выполнения операций, порождённых программой. За внешне простым языком ассемблера стоит довольно сложная архитектурная модель компьютера, допускающая детализацию любого компонента. В центре внимания – конфигурация оборудования, состояние памяти, фиксированная система команд, передачи управления, очередность событий, исключения и неожиданности, время реакции устройств и успешность процессов обработки информации, нацеленных на свободный доступ к любым возможностям оборудования. Кодирование алгоритма осуществляется на фоне применения дополнительных средств, таких как блок-схемы и документирование, отчасти компенсирующих отсутствие в языке ассемблера понятий уровня программистской фразеологии, а в естественных языках – понятий, возникающих при такой «сверхточной» детализации программ на языке ассемблера. Результативность представления программ, макротехника или автоматный подход к реализации алгоритмов – методика автоматного программирования, поддерживающая выделение шагов модели программы независимо от базовых средств ЯНУ. Автоматное программирование возникло до появления ЯВУ.

Для языков ассемблера в дополнение к общей семантике элементарного уровня ЯНУ характерно наличие команд над вещественными числами и обработки кодов. Работа с идентификаторами реализуется аппаратными возможностями адресации памяти, обычно обеспечивающей доступ практически к любому хранимому в памяти данному или команде. Время выполнения команд конечно. Управление вычислениями может учитывать результаты промежуточных вычислений (ветвления и переключатели), выполнять итерирование участков повторяемости (циклы) и вызовы подпрограмм, а также обрабатывать внутренние и внешние прерывания. Система команд ассемблера обычно не допускает расширения. Хотя можно строить подпрограммы, они не равноправны с командами. Это значит, что семантические системы ассемблера имеют формат **< V; F; R >** - команды и правило из выполнения полностью разделены. Возможна обработка кода или текста программы, т. к. и то и другое не более чем данное. В 1980-е годы лучшей системой команд считалась архитектурная модель PDP-11. В качестве опорных языков здесь рассмотрены ассемблеры MIX, MSX, P-код, RISC-машина, byte-код (muLisp и Java), MASM и БЭМШ.

Лежащая в основе ассемблера архитектурная модель не имеет ограничений, зато допускает многоуровневую детализацию. Механизмы вычислений используют дополнительные регистры (E — Eval). Память обычно содержит свою быструю адресную арифметику, буферы для обмена данными с внешними устройствами, сверх быстрые регистры, хэши и дополнительные блоки (M — Memory). Управление вычислениями кроме передач управления использует шкалу прерываний и обработчики прерываний для продолжение вычислений (C — Control). Комплексация данных использует пересылки блоков из расположенных по соседству ячеек (S — Structures).



При подготовке программ на ассемблере сначала продумывают распределение памяти (M), затем логику изменения её состояний (C), потом выбирают последовательности команд для обработки данных (E) и, если понадобится, выделение блоков данных типа векторов (S). В результате получается очерёдность приоритетов MCES, соответствующая парадигме императивного программирования.

Традиционно ассемблер реализуют как упрощенный компилятор. Учитывая повышенную нагрузку низкоуровневого программирования на отладку программ, иногда включают в систему программирования дизассемблер и интерпретатор ассемблера, обеспечивающие кроме удобства отладки широкий спектр преобразования программ на ассемблере, их оптимизации и адаптации к развитию аппаратуры. Интерпретирующий автомат для ассемблера устроен проще, чем автомат для абстрактной машины ЯВУ, благодаря встроенной реализации команд – языка КМ и отсутствию локализации имен и их областей действия. Процесс перевода программы с языка ассемблера в язык машинных команд называют ассемблированием.

Язык ассемблера оперирует такими данными, как адреса и значения. Нередко для наглядности в записи операндов команд вводится внешнее различие @адресов и #значений с помощью префиксных символов. Возможны специальные формы записи для блоков данных и литералов. При ассемблировании текст программы отображается в частично адресуемый код программы, исполнение которого обычно сводится к поячеечному изменению состояния памяти:

ассемблер = (Текст → {Код | Адрес}): Пам [ячейка] → Пам

Число слов, отводимое ассемблером под одну символическую команду, зависит не только от собственно кода команды, но и от метода адресации операндов, а возможно, и от других аспектов кодирования программ и данных, обсуждение которых здесь не предполагается. Достаточно констатировать, что программа при ассемблировании распадается на конечные последовательности команд K1 ... Kn, которым сопоставляются конечные интервалы машинных слов W1 ... Wm(a) в зависимости от **a** – системы аспектов кодирования:

[K1 ... Kn] → [W1 ... Wm(a)]

Фактически, операндная часть команды – это адреса, специальные регистры, сумматоры, значения в общей памяти, шкала прерываний и состояний или что-то другое, специфичное для конкретной архитектуры. Парадигма низкоуровневого кодирования на ассемблере нацелена на учет любых особенностей компьютерной архитектуры. Архитектура компьютера часто определяется как множество ресурсов, доступных пользователю. Это система команд, общие регистры, слово состояния процессора и адресное пространство. Процесс ассемблирования заключается в следующем:

- резервирование памяти для последовательности команд, образующих ассемблируемую программу;
- сопоставление используемых в программе идентификаторов с адресами в памяти;
- отображение ассемблерных команд и идентификаторов в их машинные эквиваленты.

Для реализации такого процесса требуется счетчик адресов и таблица идентификаторов. Программист может не знать абсолютных адресов[90] ячеек памяти, занятых для хранения констант, команд и промежуточных результатов вычислений, но обычно предполагается, что последовательно написанные команды будут расположены в последовательных ячейках памяти.

Язык ассемблера обычно различает следующие категории команд:

- вычисления с результатом в сумматоре, для которых следующая команда расположена по соседству;
- изменение части состояния общей памяти, при котором следующая команда расположена по соседству;
- обработка текста программы без генерации нового кода;
- управление со своими правилами выбора следующей команды.

[90] Это зависит от архитектуры и от возможностей ассемблера. Часто это делает не сам ассемблер, а сопутствующий инструментарий.

При записи команд на ассемблере принято структурировать строки на поля, предназначенные для последовательного размещения метки, кода команды, операндов и комментария. Наибольшее разнообразие возможных решений связано с формами адресации операндов на уровне машинного языка. В зависимости от команды используются разные методы адресации операндов.

Обычно система команд ассемблера, кроме команд арифметических вычислений и передач управления, содержит команды манипулирования данными, их ввода-вывода через буфер обмена, возможно, доступный через механизм прерываний. При этом используется код состояния процесса, отражающий свойства текущей выполняемой команды, такие как выработка нулевого или ненулевого кода, отрицательного или положительного числа, переноса разряда за границы машинного слова и др.

Имеются команды условного перехода, возвратный вызов процедуры с использованием регистра возврата и передача управления со счетчиком числа циклов. Встречаются и другие команды, разнообразие которых не влияет на особенности ассемблирования и применения ассемблеров в СП. Именно язык ассемблера традиционно выступает в СП в роли конкретной машины (КМ) при компиляции программ.

Программирование на ассемблере подразумевает знание специфики системы команд процессора, методов обслуживания устройств и обработки прерываний. Система команд может быть расширена микропрограммами и системными вызовами в зависимости от комплектации оборудования и операционной системы. Это влияет на решения по адресации памяти и коммутации комплекта доступных устройств. Но есть и достаточно общие соглашения о представлении и реализации средств обработки информации на уровне машинного кода.

Обычно ассемблер обеспечивает средства управления распределением памяти и управления компиляцией кода независимо от системы команд, что позволяет программисту принимать достаточно точные решения на уровне машинного языка. Существуют средства отладочного исполнения и распечатки листингов. Управление ассемблированием обычно обеспечивает средства авторизации, взаимодействия с отладчиком, текстовым редактором, операционной системой и средствами приаппаратного уровня, доступными через операционную систему и аппаратные средства ввода-вывода (BIOS).

По тексту программы при ассемблировании формируется код программы, выполняющий пошаговое преобразование памяти.

Программирование на ассемблере требует знания особенностей применяемых команд, их операндов и результатов. Большинство команд изменяют один регистр, если не считать счётчик команд. Некоторые команды могут изменять более одного регистра. Например, умножение вещественных чисел может формировать регистр младших разрядов для вычислений с повышенной точностью, деление целых чисел может формировать регистр с остатком от деления, пересылка меняет содержимое целого ряда машинных слов.

Спецификация команд абстрактной машины ассемблера может быть задана над тройкой <S, C, M >, в которой S и C – простые регистры, сумматор и программа, а M – вектор, представляющий общую память.

Приведём пример системы команд.

Таблица 3.9

Пример системы команд ассемблера

Обозначение команды	*Описание команды* [91]	*Примечание*
LDC	Засылка адреса в сумматор.	Подготовка операндов
LDM	Засылка адресуемого слова в сумматор.	
LDS	Косвенная засылка слова в сумматор.	
RV	Пересылка из сумматора в память.[92]	Сохранение результата
RVS	Пересылка слова по адресу из сумматора в память.	
GO	Безусловная передача управления.	
GS0	Условная передача управления по нулевому значению сумматора.	Ветвление
GS1	Условная передача управления по ненулевому значению сумматора.	
GSUB	Передача управления с запоминанием адреса возврата в сумматоре.	Подпрограмма
DEC	Вычитание 1 из содержимого сумматора.	Арифметические операции
INC	Прибавить 1 к содержимому сумматора.	
ADD	Прибавление адреса к содержимому сумматора.	
SUM	Суммирование адресуемого слова и содержимого сумматора.	

[91] Следующая команда расположена по соседству или задана командой передачи управления

[92] Пересылки – это изменение части состояния общей памяти.

Таблица 3.10
Пример спецификации команд ассемблера

RA	RA’	*Примечание*
s (LDC Adr . c) m	→ (s:=Adr) c m	Операнд из команды
s (LDM Adr . c) m	→ (s:=[Adr]) c m	Операнд из памяти
(s:Adr) (LDS . c) m	→ (s:=[Adr]) c m	Операнд по адресу в сумматоре
s (RV Adr . c) m	→ s c (m|m.Adr:=[s])	Результат из сумматора
(s:Adrs) (RVS Adr . c) m	→ s c (m|m.Adr:=[Adrs])	Результат по адресу
s (GO Adr . c) m	→ s Adr m	
(s:0) (GS0 Adr . c) m	→ s Adr m	
(s:1) (GS1 Adr . c) m	→ s Adr m	
s (GSUB Adr . c) m	→ @c Adr m	Укрупнение действий
s (DEC . c) m	→ (s-1) c m	
s (INC . c) m	→ (s+1) c m	
s (ADD Adr . c) m	→ (s+Adr) c m	
s (SUM Adr . c) m	→ (s+[Adr]) c m	

Применение ассемблера для разработки многократно используемых модулей налагает определённые ограничения на структуру кода программы и его свойства:

- перемещаемость. Свойство удобно, когда код программы устроен так, что его можно расположить по любому абсолютному адресу;
- листание страниц памяти. Соотношение между адресуемой и реально доступной памятью может требовать пересмотра в процессе выполнения программы;
- зависимость от данных. Программа должна учитывать готовность данных и вероятность их обновления, особенно в случае обмена информацией через порты или буферы устройств;
- динамика размещения. Программа может размещаться в памяти пошаговым образом, методом раскрутки, с использованием динамической оптимизации, учитывающей статистику реального использования ее составляющих.

Пример программ решения типовых задач средствами ассемблера:

Фрагмент программы	*Примечание*
LD HL, ADR1 откуда LD DE, ADR2 куда LD BC, 2048 сколько LDIR	Установить регистр HL Установить регистр DE Установить регистр BC Переместить данные в соответствии с содержимым регистров HL, DE, BC

Пример 3.12. Пример из ассемблера MSX:
пересылка блока (ТД не имеет значения)

Этот пример показывает зависимость корректности выполнения программы на ассемблере от правильного выбора программистом адресов данных.

Ассемблер отличается от компилятора меньшей сложностью исходного языка, перевод с которого в машинный язык можно выполнить «один-в-один». Ассемблер часто сопровождается возможностью дизассемблирования, что отличает его от большинства других языков программирования. Изучить язык ассемблера проще, чем любой язык высокого уровня, зато продуктивность его применения много ниже[93]. Знание ассемблера помогает понимать код программы, подготовленной на других языках. С языками сверх высокого уровня[94], такими как языки спецификаций или CLU, SETL, Planner положение прямо противоположной — их понимание требует способности к абстрагированию, что означает высокий порог вхождения на этапе изучения с перспективой снижения порога в применении. В результате успешного применения таких языков добиться гораздо сложнее, их применяют много реже. Пример — верификация.

Основной механизм укрупнения действий – передача управления подпрограмме. При необходимости уровень языка может быть повышен с помощью макросов. Императивный стиль программирования

[93] Это связано с разницей цели обучения и цели применения. Одно дело — изучить язык, другое — уметь разумно его применять. Сложность применения средств низкого уровня связана с традицией их реализации, не предоставляющей средства контроля правильности, что не препятствует пониманию таких языков и их успешному применению при необходимости. Высоким является не «порог вхождения», а скорее наоборот - «порог завершения» из-за сложной отладки ассемблерных или макропроцессорных программ, если они не маленькие.

[94] К языкам сверхвысокого уровня можно отнести GNU SETL, Clisp, Cmucl, Clojur Haskell, F#, C#, вообще их немного

наследуется большинством ЯВУ, поддерживающих императивно-процедурное и объектно-ориентированное программирование. Получается понятийная таблица из трёх строк.

Таблица 3.11
Понятийная таблица ассемблера

V	M	C	E	S
X: Команды	Сборка	Старт	Подпрограммы	Соседство
D: Прерывания	Обработчики	Условные передачи управления	Шкала	Номер
B: Коды	Операции	Следование	Команды	Размер Слова

V : константы — Команды, номера прерываний и коды

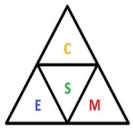
E : **V* → V+** вычисления — Выбор данных в памяти, команды, доступ к шкале прерываний (битовые операции, неявные регистры и др.)

M : **T: N → V** память — Сборка программы и обработчиков прерываний, вызовы процедур (адресная арифметика, регистры, блоки)

C : **(F → {0,1})*** управление — Запуск программы, очередная команда, передачи управления (условия, метки, указатель программы, переходы, обработчики сигналов)

S : **A ↔ K** структуры — Размер слова, соседние данные в памяти, индексация (сигналы, шкала прерываний, шины передачи данных)

Таблица 3.12
Парадигматическая характеристика ассемблера

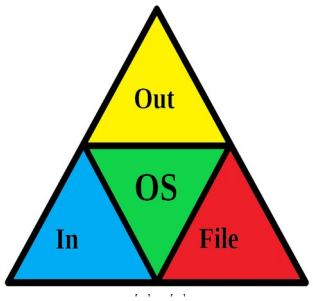
Параметр	*Конкретика*
Постановки задач	Требуется обеспечить доступ ко всем возможностям компьютера, особенно за пределами RISK-архитектур, поддерживаемых ЯВУ.
Ограничения архитектурной модели 	Архитектурная модель без ограничений, допускает детализацию. (E — Eval) вычисления используют дополнительные регистры. (M — Memory) содержит адресную арифметику, сверх быстрые регистры, хэши и др. (C — Control) управление использует шкалу прерываний и обработчики прерываний. (S — Structures) использует пересылки блоков.
Семантические системы	**< V; F; R >** - команды и правило из выполнения полностью разделены.
Упорядочение приоритетов в принятии решений	M C E S: распределение памяти (M), логика изменения состояний памяти (C), последовательности команд вычисления (E), выделение векторов (S).
Понятийная таблица (три строки)	**V :** Команды, номера прерываний и коды E : **V* → V+** размещение в памяти, вычисления, доступ к шкале прерываний (битовые операции, неявные регистры и др.) M : **T: N → V** Сборка кода и обработчиков прерываний, вызовы процедур (адресная арифметика, регистры, блоки) C : **(F → {0,1})*** Запуск программы, очередная команда, передачи управления (условия, метки, указатель, переходы, обработчики) S : **A ↔ K** Размер слова, соседние данные, индексация (сигналы, шкала прерываний, шины передачи данных)
Эксплуатационная прагматика ЯП	Ассемблер – средство эффективного программирования при решении задач доступа к полному спектру возможностей оборудования. Бывает реассемблер, билистинг, прокрутка и отладчик.
Реализационная прагматика	- Код можно расположить по любому абсолютному адресу (перемещаемость); - листание страниц памяти, виртуальность; - динамика размещения; - объектный код.

Парадигматическая специфика	Классический язык императивного машинно-ориентированного программирования.

3.4. Языки управления процессами

Задачи языков управления процессами нацелены на переход от представления программ к выполнению процессов на компьютере. Лежащая в основе таких языков архитектурная модель выходит за пределы процессора, она обобщена на устройства ввода-вывода.

Процессор по существу становится лишь средством коммуникации периферийных устройств (S — OS), вычисления привлекаются по мере необходимости, в значительной мере используются данные с устройства ввода (E — In). Работа с памятью обобщена на файлы — устройства внешнего хранения данных (M — File). Файлы могут использоваться и как унифицированное представление устройств, заданий и процессов. Управление вычислениями дополнено понятиями приоритет процесса, логика «успех-отказ» и вывод данных, включая сообщения об ошибках (C — Out).



In – устрой
Out – устр
File – устройства внешнего хранения данных.

Рассмотрим базовые средства для решения проблем обработки информации на уровне функционирования операционных систем (ОС), исполнения отдельных задач и разработки информационных систем, активно использующих понятие «файл». Нередко представление процессов и файлов может быть унифицировано, что видно по сопоставлению команд по обработке файлов и манипулированию процессами. Учитывая их особенное место, их естественно выделять в особенную парадигму. Критерием является, что приходится думать иначе, чем при обычном программировании.

Таблица 3.13

Параллелизм команд над файлами и процессами

Действие	*Файлы*	*Процессы*
Вывести список	Ls	Ps
Сменить статус	Chmod	Bg Fg
Удалить	Rm	Kill
Сцепить последовательно	Cat	| конвейер
Показать контекст/задания	Env	Jobs
Создать файл/процесс	Cp – в новый файл	Fork
Сменить директорию/процесс	Cd	Exec
Проверить/Ждать	Test	Wait
Переход к результату	Echo	Eval
Формат файла/процесса	Таблица строк/записей	Список команд с параметрами

Описание процесса начинается с определения класса событий, представляющих интерес для участвующих в нем объектов. Множество имен событий, используемых при описании процесса или объекта, обычно предопределено (C). Далее определяется конфигурация физически доступных устройств, что можно рассматривать как расширение понятия «память» (M). Потом решаются вопросы защиты файлов от несанкционированного доступа (S) и выполняются необходимые вычисления (E). Это задаёт упорядочение приоритетов впринятии решений как CMSE — управление, память для файлов, защита данных и вычисления.

Первая абстракция при моделировании процессов – исключение времени, т. е. отказ от ответов на вопрос, происходят ли события строго одно за другим. Это обеспечивается следующими договоренностями:

- элементарные действия исполняются мгновенно;
- протяженное действие: всегда пара событий – начало и конец;
- нет точной привязки действий к моменту времени;
- определены отношения «раньше – позже», «одновременно», «независимо»;
- совместность событий понимается как отношение «синхронизация»;
- одно событие из независимых возникает в любом порядке, без причинно-следственной связи.

На уровне ОС информационная обработка выглядит как семейство взаимодействующих процессов, выполняемых по отдельным программам – заданиям или сценариям, размещенным в файлах, а потому допускающих формирование и изменение в динамике. Это означает, что семантические системы ОС, также как макрогенераторов и языка Forth, имеют формат **< V, F; R >** - задания и правило их применения разделены, но можно строить новые задания, равноправные с командами.

Языки для ОС работают с очередями, которые могут быть представлены как строки или файлы. В памяти хранится контекст задания и его сценарий. Контекст содержит перечень доступных файлов. При управлении процессами выполнения заданий используются условия готовности и вырабатываются сигналы, символизирующие успех выполнения действий. Сигналы также хранятся в контексте. Действия могут быть организованы в конвейеры или последовательности и обусловлены успехом предшествующих действий. Очередь может быть пополнена.

Функционирование ОС обеспечивает следующие явления и критерии:

- порождение новых файлов и процессов по ходу дела;
- время жизни файлов и процессов произвольно – нет гарантий;
- неограниченная динамика событий;
- содержание может быть незавершенным;
- изменение содержания и состава;
- очередь процессов с условиями готовности.

Построение модели языка управления заданиями требует дополнительных операций по работе с очередями, что может быть устроено как «ленивый» список, в конец которого можно встраивать новые элементы. В качестве опорного языка рассмотрен Bash, абстрактная машина которого может быть определена как <E, C, D>, где:

E - контекст_процесса – вектор записей Имя: Данные + stdin stdout;

C - текущий_процесс – строка из команд;

D - очередь_отложенных_процессов – вектор записей Имя: Данные.

Команды интерпретатора ОС выполняют обмен данными между процессами и контекстом, обработку очереди, файлов и контекста, проверку ряда условий над данными, контекстом и очередью, выработку сигналов, включая установку сигнала о завершении процесса.

Таблица 3.14

Типичный набор команд языка управления процессами

SCQ	*Описание команды*	*Примечание*
ECHO	Вывод аргументов	Визуализация
PWD	Выводит название текущего рабочего каталога	
LS	Список файлов в текущей директории	
CAT | MORE	Просмотр содержимого текстового файла	
CD	Сменить директорию	Манипуляции с файлами
CP	Копировать файлы	
MV	Переместить или переименовать файл	
RM	Удалить файлы	
LN	Создать символическую ссылку	
EVAL	Конструирование команды на лету и ее выполнение	Вычисления
EXEC	Вызов другого процесса	
LET	Вычисление выражений	
READ	Ввод значения переменной	Установка значений
SET	Изменяет значения внутренних переменных скрипта	
TEST	Проверка условия	
TRUE	Возвращает код успешного завершения = ноль	
FALSE	Возвращает код завершения, свидетельствующий о неудаче	
Контроль процессов		
PS	(print status) список текущих процессов с их IDs (PID)	Визуализация
FG	Активизировать фоновый или приостановленный процесс	

BG	Сделать процесс фоновым. Обратная функция от fg.	Управление активностью процессов.
WAIT	Ждет выхода из дочернего процесса	
KILL	«Убить» процесс. PID «убиваемого» процесса даёт PS	

Обозначения:

stdin – стандартный ввод. То, что набирает пользователь в консоли.
stdout – стандартный вывод программы.
stderr – стандартный вывод ошибок.
(Expr) – результат вычисления выражения или успех выполнения процесса.
$ – переменная для кода успеха/результата процесса.
$* – все аргументы переданные скрипту(выводятся в строку).
$! – PID последнего запущенного в фоне процесса.
$$ – PID самого скрипта.
NN(d) – список номеров и имён элементов очереди
[<Num, Name, Text>, …] – очереди процессов.
NULL – пустой файл.
H(d) – голова очереди, точнее – процесс с наивысшим приоритетом.
T(d) – хвост очереди, остаток после удаления головы. d = H(d) • T(d).
PN – имя текущего процесса.

Таблица 3.15
Спецификация команд управления процессами

RQ	RQ’	*Примечание*
e (ECHO String . c) d	→ (e[stdout] | String) e c d	Вывод на стандартное устройство
(e[PWD]=DName) (PWD . c) d	→ (e[stdout] | DName) c d	
(e[PWD]=DName) (LS . c) d	→ (e[stdout] | [DName]) c d	
e (CAT Fname . c) d	→ (e[stdout] | [Fname]) c d	
e (CD New. c) d	→ (e[PWD] := New) c d	Установка переменной
(e[F1]=Datum) (cp F1 F2. c) d	→ (e[F1]=Datum; e[F2]=Datum) c d	Ссылка на копию файла
(e[Fname]=Datum) (RM Fname . c) d	→ (e[Fname]=NULL) c d	Ссылка на пустой файл
e (EVAL T1 ... TK . c) d	→ e (T1 | ... | TK . c) d = e ($* . c) d	Выполнение скрипта
e (EXEC Fname . c) d	→ e ([Fname] c) d	Выполнение файла
e (TEST Expr . c) d	→ (e[$] := (Expr)) c d	Проверка успешности
e (TRUE . c) d	→ (e[$] := 0) c d	Установка признака успешности
e (FALSE . c) d	→ (e[$] := 1) c d	
e[stdin=Text] (READ X . c) d	→ (e[X] := Text) c d	Прием текста
e (PS . c) d	→ (e[stdout] | NN(d)) c d	Номера процессов
e (BG . c) d	→ e H(d) (T(d) | <$$, e[PN], c >)	Смена статуса процесса
e (FG Num . c) (d[Num]=Text)	→ e (Text | c) (d[Num]:=NULL)	
e (KILL Num . c) (d[Num]=Text)	→ e c (d[Num]:=NULL)	

Поддержаны рамочные конструкции для построения многоярусных условий и циклов. Более подробно со средствами управления процессами на уровне ОС можно ознакомиться в книгах по ОС.

Фрагменты	*Примечание*
$ ls doc[c-d]	Перечень файлов с именами, соответствующими маске
$ set 0 noclobber	Установка системной переменной
$ cat newletter1 newletter2 >! Oldletters	Управление потоком данных
$ at 8:15 jobs	Назначение времени запуска работ

Пример 3.13. Пример программы управления заданиями

Внешне языки управления процессами выглядят как нечто среднее между макроассемблерами и языками высокого уровня. Различие проявляется в понимании данных, подвергаемых обработке, и командах, к которым сводятся процессы обработки:

– роль данных выполняют файлы – объекты, обладающие собственным поведением и подверженные влиянию внешнего мира. Существование файлов в период обработки не всегда очевидно. Файлы могут участвовать одновременно в разных процессах[95];
– выполнение команды рассматривается как событие, которое может быть как успешным, так и неудачным. Кроме того, существуют внешние события[96]
– реакция на событие программируется как обработчик события, выполняемый независимо от других обработчиков (это отдельный процесс);
– программа процесса может быть нацелена не на получение результата за конечное время, а на обеспечение непрерывного обслуживания заданий на обработку объектов;
- процесс может быть активным или отложенным;
- процессы могут конкурировать за общие объекты;
- возможна синхронизация процессов и порождение подчиненных процессов;
– программа процесса выглядит как объект и создается как элемент данных, а потом может применяться равноправно с командами;
– последовательное расположение команд в программе не считается основанием для их выполнения в точно том же порядке. Выполнение команды может занимать ряд интервалов времени, между которыми выполняются другие команды.

На уровне ОС основная работа сводится к управлению заданиями, нацеленными на эффективную загрузку общего оборудования и других ресурсов. Доступ к общим ресурсам обычно регулируется с помощью очередей запросов на обслуживание имеющихся устройств и ресурсов, не только процессора. Обслуживание носит асинхронный характер[97]. Основной критерий качества – возможность продолжить выполнение заданий без принципиальных потерь информации.

Любая программа при разработке и отладке выполняется на фоне ОС, управляющей процессами ввода-вывода данных ради демонстрации хода обработки данных. Поэтому минимальный контекст отлаживаемой программы – стандартный ввод-вывод, доступный по умолчанию. На уровне языка управления процессами активно используются умолчания, раскрываемые в терминах текущих значений или системных переменных. Основное отличие – укрупнение данных, переход от ячеек и строк к долгоживущим файлам. Кроме того, смягчается зависимость от последовательности вызова процессов, времени их инициирования. Переход к проблемам управления процессами влечёт радикальное изменение понятия «результат». Это не более чем код успеха/провала завершённого процесса. Возможен учёт приоритетов отложенных процессов. Появляются имена, локализованные внутри скриптов.

Поверхностная визуальная характеристика понятийной сложности средств ОС может быть выражена как понятийная таблица и 4-х строк.

Таблица 3.16.
Понятийная таблица языка заданий

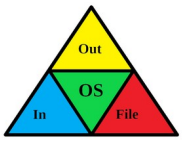
V	**E**	**M**	**C**	**S**
B: устройства.	драйвер	имя:	событие	комплекс
X: файл	файл	имя.*	?Э ±	директории
D: задание	процесс	PS	приоритет	библиотеки
P: UID	настройки	среда	прерывание	Сеть

V : константы — Устройства, файлы, задания и сетевые адреса
E : **V*** → **V+** вычисления — Доступ к устройствам, ввод данных, насройки и процессы
M : **T: N** → **V** память — Имена, обстановка и протоколы

[95] Бывает возможность одновременного чтения (много читателей) и встречается возможность распределённых воздействий на БД.
[96] Подразумевается механизм сигналов (unix), не считая слома самого компьютера.
[97] Обычно указатели на предстоящие процессы ОС размещает в очереди. Некоторые процессы могут иметь приоритеты. Выполнимость других может зависеть от доступа к устройствам или датчику времени и от наличия памяти. ОС по мере возможности запустить очередной процесс выбирает из неё тот, у которого наивысший приоритет с учётом исполнимости. Поэтому реально процессы запускаются не поочерёдно, а в том или ином порядке в зависимости от совокупности разных причин в обстановке ОС. Кроме того, по мере прохождения процессов, оператор может смотреть очередь процессов и по своему желанию повышать или понижать приоритеты отдельных процессов.

C : **(F → {0,1})*** управление — Проверка существования файлов, события, приоритеты и прерывания
S : **A ↔ K** структуры — Комплекты, директории, библиотеки

Таблица 3.17
Парадигматическая характеристика языков управления процессами

Параметр	*Конкретика*
Постановки задач	Преодоление границы между программами и процессами их выполнения на компьютере.
Обобщение архитектурной м 	Процессор становится средством коммуникации устройств (S — OS), вычисления используют данные с устройства ввода (E — In), память обобщена на файлы (M — File), управление учитывает приоритеты процессов, логику «успех-отказ» и вывод данных, включая сообщения об ошибках (C — Out).
Семантические системы	Семантические системы ОС имеют формат **< V, F; R >** - задания и правило их применения разделены, задания в файлах, равноправны с командами.
Упорядочение приоритетов в принятии решений	**C M S E** - Предопределено множество событий (C), определена конфигурация физически доступных устройств (M). Потом организована защита файлов (S) и необходимые вычисления (E).
Понятийные таблицы (4 строки)	V : файлы, устройства, задания и сетевые адреса E : V* → V+ ввод данных, настройки, процессы, результаты вычислений M : T: N → V имена, каталоги, обстановка и протоколы C : (F → {0,1})* существование файлов, события, приоритеты и прерывания, «успех-отказ» S : A ↔ K комплекты файлов, директории, библиотеки
Эксплуатационная прагматика ЯП	Обеспечение бесперебойной эксплуатации и функционирования оборудования. Абстрагирование от аппаратуры, обслуживание запросов от программных инструментов и устройств,
Реализационная прагматика	Чаще всего используются две модели ОС – супервизор, контролирующий взаимодействие процессов, или автомат, способный тиражировать себя при ветвлении процессов. И в том и в другом случае возникает бесконечный цикл анализа происходящих событий с включением обработчиков.
Парадигматическая специфика	Любая программа при разработке и отладке выполняется на фоне ОС, управляющей процессами ввода-вывода данных ради демонстрации хода обработки данных. Поэтому минимальный контекст отлаживаемой программы – стандартный ввод-вывод, доступный по умолчанию.

Формальные модели ОС достаточно просты. По уровню сложности они кажутся проще АМ, они не гарантируют защиту контекста. Усложнением является поддержка процессов, допускающих независимое выполнение, что требует перехода к комплекту взаимодействующих АМ, Реализационная семантика ОС, как правило, требует введения большого числа дополнительных понятий (очередь, логика, словарь, точка возврата, позиция в стеке, шкала прерываний и т. д.) и требований к качеству, надёжности и безопасности, возникающих на уровне эксплуатационной прагматики, а также схем программ и программистской терминологии, что делает реальные ОС весьма сложными.

3.5. Базы данных и массовое обслуживание

Применение и разработка баз данных (БД) по сложности используемых понятий характеризуется тремя слоями понятийных таблиц из 4-х строк. Три слоя отражают многоуровневое моделирование реальных объектов при переходе к записям в БД и пользовательским интерфейсам, причём раскладки понятий на слоях могут отличаться, хотя они представляют одно и то же содержание.

Таблица 3.18.
Реализационная модель БД

Название слоя	**№ слоя**	**Спецификация семантических систем слоя**

Реализационная модель 3		**K**	AM → KM			
Пользовательская модель 2		**L**	AM → KM			
Концептуальная модель 1		**A**	AC → AM			
№	**столбца**	**V**	**E**	**M**	**C**	**S**
строки	**Виды функций**	V	V* → V-	T: N → V	(F → {0, 1})*	A ↔ K
B	Вычисления	Числа, даты, время, строки	Операции	Клетка Транзакции	Запросы	Записи
X	Укрупнения	Таблицы	Функции	Индексы	Столбцы	Файлы
D	Границы	Диагностика Исключения	Предикаты	Типы полей записи	Метаданные Схема данных	Клиент Интерфейс
P	Практичность	Нормализованные формы: НФ[1-12]	Отношения Шаблоны	Распределённая информационная система (РИС)	Система управления базами данных (СУБД)	Ввод-вывод in/out сервер

Важнейший механизм БД, заслуживающий наследования современными ЯП - это транзакции, вносящие определённость в организацию процессов из действий, возможно подверженных сбоям. Многие возлагают надежды на такую организацию памяти для языков параллельного программирования. Переход на уровень систем управления базами данных (СУБД) приводит к дополнению схемы четвёртым, универсальным слоем.

3.6. Сети и сайты, распределённые системы и Интернет

Устройство информационных сетей по существу является функционированием гетерогенных графов, подверженных непредсказуемой модификации. Принципиальная разница проявляется в существенном обобщении архитектурной модели, заключающемся в учёте сетевых конфигураций и многопроцессорных комплексов. Вместо АМ приходится рассматривать абстрактный комплекс (АК), команды которого представляют возможности взаимодействия процессов. Обобщение на возможности сетевого доступа и многопроцессорности можно рассматривать как основание рассматривать ПВ равноправно с базовыми и фундаментальными парадигмами.

Механизмы вычислений определены возможностями комплекта доступных процессоров (E — Complex). Работа с памятью дополнена дополнительными методами адресации компьютеров и серверов (M — Cloud). Управление обогащено разными ограничения доступа к узлам сети (C — Net). Комплексация данных расширена возможностями серверов, сетевых сервисов и облачного хранения данных (S — Host).

Complex – многопроцессорная конфигурация
Cloud – облачное хранилище
Net – дистанционное управление
Host – сервер, платформа

Сложность системы понятий оценивается понятийной таблицей из четырёх строк.

V		**E**	**M**	**C**	**S**
B	html	компьютер	URL	ping	сеть
X	сайт	маршрут	сервер	навигация	Шлюз (gate)
D	браузер	окно	буфер	выбор	DNS
P	UID	SSH	протокол	±	сервисы

Таблица 3.19
Наследование методов ЯНУ парадигмами ЯВУ

Механизм	Парадигма	*Примечание*
Ассемблер	ИП ООП	Императивный стиль программирования, как на ассемблере, унаследован большинством процедурно-императивнвых или объектно-ориентированных языков. Обычно он дополнен более сложными выражениями и иерархией структур данных и классов объектов
Стековая машина	ЯВУ	Стековые машины служат в большинстве ЯВУ основным механизмом поддержки выражений, функций и процедур с вычисляемыми параметрами
Макро-техника	ЛП ФП препроцес-соры	Продукционное программирование в стиле макротехники унаследовано логическим программированием. Функциональное программирование переносит её на уровень работы со структурами данных. Многие системы программирования на ЯВУ используют такую технику в препроцессорах и как внутренний инструмент кодогенерации.
Управление процессами	ООП, ПВ	Механизм управления заданиями частично наследуется парадигмой ООП, использующей взаимодействия объектов с помощью сообщений. Он существенно используется и парадигмой параллельных вычислений для описания асинхронных процессов.

Механизмы представления и обработки данных, накопленные в ЯНУ, в значительной мере унаследованы методами реализации ЯВУ, что позволяет локализовать изучение таких механизмов. Практика программирования на ЯНУ имеет образовательное значение. Ценящие подготовку высококвалифицированных программистов вузы, готовящие победителей международных чемпионатов по программированию, включают в начальное обучение программирование на ассемблере и управление процессами на Linux.

Выводы:
1. Монопарадигмальность ЯНУ делает их удобными для изучения отдельных парадигм.
2. Базовые средства основных парадигм можно представить на материале языков ассемблера, макротехники, стековой машины и управления заданиями.
3. Разнообразие ЯНУ показывает спектр возможных реализационных решений при переходе от ЯП к ЯиСП.
4. Средства СУБД и работы с сетями показывают практичность расширений архитектурных моделей на работу с файлами, сетями и комплексами процессоров.

Практика-3: Маленькие упражнения на разных ЯНУ или их аналогах из знакомых ЯП.

3.1. Средствами препроцессора разрешите программам на Си вместо фигурных скобок использовать слова «begin» и «end».

3.2. Напишите на языке Forth функцию от четырёх параметров, вычисляющую произведение суммы первых двух параметров и разность третьего и четвёртого.

3.3. Сделайте на ассемблере или с помощью команд абстрактной машины функцию Фибоначчи методом динамического программирования.

3.4. На языке Bash напишите скрипт, позволяющий при наличии одного данного файла его первый символ записать во второй заданный файл.

3.5. Подготовьте краткую справку о возможностях какой-либо архитектуре или языка низкого уровня.

(ЭВМ БЭСМ-6 JVM P-код RISC-архитектуры MIX MMIX MPI Ассемблер «Эльбрус» LLVM)
(bash Сигма Unix Postscript Basic Forth PHP Sml XML SQL Tcl* TRAC **Tcl** GPM html JCL Эпсилон)

ЧАСТЬ 2. ФУНДАМЕНТАЛЬНЫЕ ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Вторая часть посвящена рассмотрению фундаментальных парадигм программирования, сложившихся в практике создания и применения языков высокого уровня (ЯВУ). Эти парадигмы направлены на преодоление кризиса технологий программирования в производственной, академической и образовательной сферах, в которых по разному рассматривается важность статики и динамики организации процессов вычислений, а также критериев правильности, эффективности, производительности, надёжности и безопасности программ. Обычно фундаментальными называют парадигмы функционального, параллельного, императивного программирования, а также мета-программирования, логического программирования и ООП.

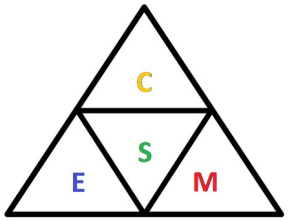
Лекция 4. Функциональное программирование

Обычно функциональное программирование привлекает внимание при смене электронной базы и обнаружении новых исследовательских задач. В последние десятилетия такой интерес связан с исследованием проблем параллельного программирования.

4.0. Особенности языков высокого уровня

Переход к ЯВУ привел к потере разнообразия архитектурных моделей.

Лежащая в основе ЯВУ архитектурная модель обычно соответствует ограничениям RISC-архитектур[98]. Механизмы вычислений ограничены целочисленным сложением, вычитанием, простейшими логическими операциями и сдвигами (E — Eval). Работа с памятью сводится к чтению-записи в регистры или слова, имеются сверх быстрые регистры и хэши (M — Memory). Поддержаны условные передачи управления, организация подпрограмм и установка статусного слова (C — Control). Комплексация данных использует стек, указатели и соседство слов (S — Structures). Детализация при необходимости выполняется ассемблерными вставками, а обобщение — библиотечными модулями.



Стоит отметить, что система команд БЭСМ-6 содержала подмножество, эквивалентное RISC-архитектуре.

Граница между ЯНУ и ЯВУ определяется по переходу от одноуровневых конструкций к активному использованию иерархии определений и структур данных. Такой переход, нацеленный на защиту обрабатываемых данных, в разных ЯП выполнен по-разному, в разной мере, что влечёт определённые парадигмальные различия.

В сравнении с ЯНУ семантика ЯВУ содержит арифметику, разделенную на ряд вспомогательных систем обработки целых и вещественных чисел в разных диапазонах, обычно зависящих от разрядности адресов и машинных слов. Имеются дополнительные системы работы с указателями, символами и строками и средства конструирования и контроля типов данных. Поддерживается набор стандартных операторов управления вычислениями: ветвления, циклы и вызовы процедур/функций. Кроме работы с глобальными объектами в общей памяти, используются разные схемы локализации хранимых в памяти объектов, что обеспечено организацией структур данных по принципу иерархии описаний или динамики исполнения.

При анализе парадигм ЯВУ необходимо учитывать следующие их особенности:

– практикуются неявные формы представления отдельных понятий ради лаконизма записи программ;
– выражения чаще всего рассчитаны на схему предвычисления над скалярами конкретной длины или сложными значениями (сначала вычисляются операнды, затем вычисляется результат операции);
– различаются виды ветвлений и циклов, категории функций и процедур;
– типы данных конструируются по фиксированным в ЯП правилам и реализуются по принятым в СП шаблонам;
– схемы управления вычислениями нередко фиксированы в ЯП и конкретно реализованы в СП;
– взаимодействие и соответствие средств и методов, относящихся к разным семантическим системам, при их реализации в СП определено по традиции и прецедентам, причем в коде оно скрыто или рассредоточено, а не структурировано;

[98] https://www2.eecs.berkeley.edu/Pubs/TechRpts/1982/CSD-82-106.pdf

– эффективность программирования базируется на знании методов реализации значений и обработчиков структур данных в памяти;

- результат программы, как правило, рассредоточен по разным глобальным переменным, но во многих ЯВУ есть выражения и функции, формирующие единственный результат.

Инструментальное ядро ЯВУ на начальных этапах реализации или изучения можно ограничить одной арифметической системой. Элементарные уровни опорных ЯВУ разных парадигм можно описать как расширения или конкретизации такого ядра.

История парадигм ЯВУ позволяет в конструкциях ЯП и методах их поддержки в СП отметить вехи совершенствования системной программотехники:

– первый ЯВУ Fortran ввёл в практику раздельную компиляцию, радикально снизившую трудозатраты на отладку программ благодаря выделению техники сборки программ из независимо создаваемых модулей, что дало взлёт продуктивности программирования алгоритмов;

– универсальный язык Lisp дал жизнь машинно-независимому стилю символьной обработки данных, что позволило исследовать многие классы новых задач и привело к парадигме функционального программирования (ФП), смягчающей переходы к новым поколениям элементной базы и смене ИТ, повысило продуктивность исследований средств и методов в новых сферах приложения компьютерной обработки данных;

– разработка языка Си как средства реализации операционных систем привела к технологии машинно-зависимого переноса программ, что определило популярность языка Си и одновременно поддержало технологию переноса ЯП на множество архитектур и повысило продуктивность создания Си-подобных ЯП;

– практика применения учебного языка Pascal позволила сформулировать принципы структурного программирования, выполняющего роль не только доминирующей методологии учебного программирования, но и роль производственного ЯП на протяжении почти десятилетия;

– программирование на языке Prolog позволило работать с недоопределѐнными постановками задач, повышая степень их изученности в рамках парадигмы логического программирования (ЛП) и реализации средств логического вывода на базе частичного определения решений, достаточных для практики, что способствовало повышению продуктивности освоения новых классов задач;

– 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, неискажающее ранее отлаженные определения, что позволило повышать продуктивность программирования наследованием готовых решений;

- методика ООП и появление C++ расширили сферы приложения информационных систем, при конструировании которых необходимо повторное программирование.

Представление специфических деталей парадигм при сравнении ЯВУ можно выразить в форме нормализованной предикатной формы интерпретатора, различающего сквозные понятия сравниваемых языков на уровне абстрактного синтаксиса (АС) и абстрактной машины (АМ), дополненной описанием реализационной прагматики и онтологической спецификации опорных ЯП, поддерживающих разные ПП.

При сравнении императивного и функционального подходов к программированию, П. Лэндин (P.J. Landin) предложил специальную абстрактную машину secd в качестве АМ для ЯП функционального программирования, подробное описанную в книге П Хедерсона. Машина secd работает над четырьмя независимыми регистрами неограниченного размера: стек для результатов, контекст для именованных значений, управляющая программа, резервная память (Stack, Environment, Control list, Dump). Там же для ЯП логического программирования описана машина secdr, дополняющая определение secd регистром для хранения неопробованных вариантов. Аналогичная конструкция пи-кода, предложенного Н. Виртом для реализации компиляторов, принципиально отличается опорой на общую память фиксированного размера, что может быть представлено как абстрактная машина secpw для ЯП императивного программирования (ИП). Она работает над пятью регистрами в общей памяти: стек для значений, контекст для аргументов функций, локальных переменных и регистра возврата из функций, управляющая программа, общая память и внешний мир, совмещающий возможности внешней памяти и иерархии данных (Stack, Environment, Control list, Partition, World). Переход от императивного программирования к ООП можно проиллюстрировать на примере байт-кода Java-машины, представленного здесь как машина secdpw, построенная включением резервной памяти в машину secpw.

В практике современного программирования популярны графические интерфейсы и компонентные технологии, поддерживающие перенос отлаженных результатов в разные системы. В центре внимания – интеграция с встраиваемыми библиотеками процедур, эффективная компиляция программ, контроль типов данных, соответствие стандартам области применения программ и технологиям быстрой разработки удобно сопровождаемых программ. Ряд проблем эффективности решается включением в ЯВУ

низкоуровневых средств. Практически исчезает необходимость в блок-схемах, а методика само-документирования и реализации справочных подсистем смягчает роль документирования. Программе на ЯВУ обычно соответствует семейство допустимых процессов, определение которого представлено формальной семантикой и прагматикой ЯП. Система программирования (СП), поддерживающая ЯВУ, как правило, порождает один из процессов этого семейства. Такое сужение диктуется не только реализационной прагматикой ЯВУ, но и необходимостью воспроизведения процессов при отладке программ.

Текст программы на ЯВУ обычно обретает бипланарность – императивное представление процесса обработки данных в нем совмещено с декларативным описанием типов обрабатываемых данных, спецификаций, прагм и пр. Возникает нечто вроде пространственной аппроксимации процесса вычислений, используемой при проверке корректности программ - статический или динамический контроль типов данных. Проработка понятий ЯВУ характеризуется пропорциями между чёткой аппликативностью и недетерминизмом, пространствами константных и переменных значений, элементарных и составных данных, открытыми и замкнутыми процедурами, средствами обработки строк и файлов, возможностями активного и «ленивого» вычисления, использованием последовательных и разветвляющихся схем управления вычислениями. Все ЯВУ используют стек при реализации укрупненных конструкций и защиты локализуемых данных.

4.1. Общее представление о функциональном программировании (ФП)

Внимание к идеям ФП привлёк в 1977 году Джон Бэкус в своей Тьюринговской лекции, рассматривая его как универсальную методику преодоления трудностей в императивно-процедурном программировании (ИП), связанным с избыточной детализацией обработки данных в терминах традиционного присваивания скалярных значений. Руководитель разработки языка Fortran и соавтор формул Бэкуса-Наура (БНФ) призвал к преодолению узости «бутылочного горлышка» императивно-процедурного стиля программирования и привёл в качестве примера проект функционального языка программирования FP, поддерживающего лаконичные формы представления обработки многомерных векторов, содержательно напоминающие отдельные конструкции языков Lisp и APL.

Функциональный стиль программирования сложился в практике решения задач символьной обработки данных в предположении, что любая информация для компьютерной обработки может быть сведена к символьной (существование аналоговых методов принципиально не противоречит этой гипотезе). Слово «символ» здесь близко понятию «знак» в знаковых системах. Информация представляется символами, смысл которых может быть восстановлен по заранее известным правилам.

Функциональное программирование рассматривает процесс обработки данных как композицию их отображений с помощью универсальных функций. Программа при таком подходе – не более чем одна из разновидностей данных. ФП сумело преодолеть синтаксический разнобой независимо развиваемых средств и методов организации информационных процессов. Это удалось благодаря нацеленности на проявление и ортогонализацию семантических подсистем в организации программируемых процессов. Не менее важна развиваемость представления унифицируемых структур данных и комплектов функциональных объектов. Все это позволяет языки ФП рассматривать как средство сопряжения разнородных конструкций. На их основе можно осуществлять любые интегрированные построения, представимые в машинно-независимом стиле.

Языки функционального программирования (ЯФП) используют гибкие списки и абстрактные атомы. ЯФП нацелены на полный контроль типов значений при исполнении программ, допускающих динамический анализ и управление с откладыванием вычислений - «ленивые вычисления», автоматизацию повторного использования памяти – «сборка мусора» (garbage collector - GC).

Методы ФП основаны на формальном математическом языке представления и преобразования формул, поэтому можно дать точное, достаточно полное описание основ ФП и специфицировать СП для поддержки и разработки разных ПП, моделируемых с помощью функционального подхода к организации деятельности.

Функциональное программирование отличается от большинства подходов к программированию тремя важными принципами.

1. Природа данных.

Все данные представляются в форме символьных выражений. Данные реализуются как древообразные структуры, что позволяет локализовывать любые важные подвыражения. Система программирования с такими структурами обычно использует для их хранения всю доступную память, поэтому программист может быть освобожден от распределения памяти под отдельные блоки данных.

2. Само-описание обработки символьных выражений.

Важная особенность ФП состоит в том, что описание способов обработки данных представляется программами, рассматриваемыми как символьные данные. Программы строятся из рекурсивных функций. Определения и вызовы этих функций, как и любая информация, могут обрабатываться как обычные данные, получаться в процессе вычислений и преобразовываться как значения.

3. Подобие машинным языкам.

Система ФП допускает, что программа может интерпретировать и/или компилировать программы, представленные в виде структур данных. Это сближает методы ФП с методами низкоуровневого программирования и отличает от традиционной методики применения ЯВУ. Не все языки ФП в полной мере сохранили эту возможность, но для языка Lisp она характерна. Такая возможность достижима на любом стандартном языке, просто так делать не принято.

Примерно с середины 1990-х кристаллизовалась идея «чистого функционального программирования» (ЧФП) как раздела дискретной математики, исследующего однозначные функции в дискретном пространстве над значениями типа целых чисел, символьных строк или графов. ЧФП можно рассматривать как математическую основу более общего производственного ФП. Постепенно это «чистое функциональное программирование» стали называть просто «функциональным программированием». Такое разделение смыслов примерно соответствует разнице в стандартах на академические и производственные ЯП. Теперь общее определение ФП нередко начинают с утверждения, что характеристическая его особенность заключается в том, что **одинаковые формулы в одинаковом контексте имеют одинаковое значение**. Из этого выводят исключение присваиваний, глобальных переменных[99], передач управления, побочных эффектов, работы с внешними устройствами. Такая формула уязвима из-за отсутствия единого точного определения термина «контекст».

Понятие «контекст» в программистском лексиконе обладает некоторой двойственностью. Это одновременно и фрагмент программы, иногда называемый областью существования или видимости, и контекстная таблица соответствия используемых в этом фрагменте символов и их значений. Семантика ЯП может по-разному ограничивать правила воздействия на таблицу соответствия. В одном ЯП могут сосуществовать две и более контекстных таблицы для одного фрагмента или синтаксической позиции — статика и динамика, а кроме того, глобальный и локальный контекст. Системы программирования (СП) для одного ЯП могут по-разному решать вопрос о порядке перебора таблиц при определении значения формулы, более того, такой порядок может быть изменён по опциям из программы или задания на её компиляцию.

Можно обратить внимание, что при оптимизирующей компиляции нередко выделяют линейные участки между двумя соседними присваиваниями. В таком случае можно считать, что на внутреннем языке компилятора происходит приведение программы к структуре из участков однократного присваивания, удобной для обнаружения невычисляемых, константных или дублирующихся формул. Компилятор заменяет константные формулы на вычисленную при компиляции константу, а дублирующиеся формулы — на переменную, значение которой будет вычислено при выполнении программы. Не исключено, что возможность при решении сложных задач явно и безопасно использовать при подготовке программ подобные оптимизации является сильным мотивом использования ЧФП. Эта же причина позволяет ФП быть полезным дополнением для любой ПП.

Если контекстом считать участок однократного присваивания, то приведённая выше характеристика не отличает парадигму ФП от других парадигм, допускающих запись программ в таком стиле. Кроме того, ещё в начале 1960-х Маккарти отмечал, что роль глобальных переменных могут выполнять самые внешние локальные переменные. Современные чисто функциональные языки программирования, такие как Haskell, для работы с побочными эффектами и внешними устройствами расширяют язык специальными «монадами», другие – просто библиотечными модулями или пакетами. Отсутствие передач управления во многих ФЯП компенсируется введением понятия «продолжение» (continuation), позволяющего через параметры передавать имя следующей функции. Можно вспомнить, что ещё при создании языка Algol-68 была замечена равносильность понятий «вызов процедуры» и «передача управления», а также присваиваний и передачи параметров. Таким образом, при переходе к ЧФП происходит не «исключение» вышеперечисленных понятий программирования, а «изменение» их формы вхождения в программы [100].

---

[99] «Global variables are harmful» Тони Хоар (Tony Hoare) при декларации принципов структурного программирования.

[100] «Гони природу в дверь, она влетит в окно» Н.М. Карамзин «ЧУВСТВИТЕЛЬНЫЙ И ХОЛОДНЫЙ. ДВА ХАРАКТЕРА»

Обычно ФП подразумевает поддержку ряда семантических и прагматических принципов, которые способствуют созданию функциональных моделей на этапе компьютерных экспериментов, полезных при решении новых и исследовательских задач. При подготовке программы программист может следовать семантическим принципам, отчасти выраженным или провозглашённым в определении семантики ЯП. Прагматические принципы предоставляются СП, освобождая программиста от решений, не связанных с характером постановки задачи. Описания таких принципов при определении языка обычно выглядят как примечания. Большинство семантических и прагматических принципов были заложены Дж. Маккарти в первых реализациях языка Lisp. Некоторые сложились позднее в практике разработки новых систем ФП при поиске более эффективных и производительных системных решений.

Функциональное программирование поддерживает семантические принципы представления функций, такие как всеобщность (универсальность), само-применение, равноправие параметров.

Всеобщность/Универсальность. Понятия «функция» и «значение» представлены теми же символами, что и любые данные для компьютерной обработки. Каждая универсальная функция, применяемая к любым данным, за конечное время выдаёт результат или диагностическое сообщение. Этот принцип позволяет строить представления функций из их частей - символов и вычислять части представления по мере поступления данных. Они могут формироваться даже в процессе вычислений и обработки информации о них.

Само-применение. Представления функций могут использовать сами себя прямо или косвенно, что позволяет создавать четкие и краткие рекурсивные символьные формы. Представление как значений, так и функций может быть рекурсивным. Примеры само-применения дают многие математические функции, особенно рекурсивные, такие как факториал, числа Фибоначчи, суммирование рядов и многие другие, определение которых использует математическую индукцию.

Равноправие параметров. Порядок и метод вычисления параметров значения не имеют. Параметры функции не зависят друг от друга. Можно заметить, что параметры при вызове функции вычисляются на одном уровне иерархии, в одном и том же контексте. Поэтому некоторые параметры могут вычисляться перед вызовом функции, а другие могут быть вычислены позже, но в том же контексте. Любая символьная форма в определении функции может быть извлечена из неё, превращена в параметр и, наоборот, подставлена в неё.

Повторное использование данных обеспечивается именованием или идентификацией. Параметры имеют имена, часто называемые переменными, хотя в ФП они не меняют значения в одном и том же контексте. Чисто на уровне понятий переменная — это именованная часть памяти, предназначенная для многократного доступа к изменяющимся данным, а константа - к неизменяемым данным. Изменение отношения между именем и значением в ФП возможно только путем перехода в другой контекст, что эквивалентно изменению имени. Функциональные переменные допустимы и равноправны обычным постоянным функциям и могут быть значениями аргументов или формироваться как результаты других функций. Редкость навыков работы с функциональными переменными означает лишь то, что пора освоить такую возможность, потенциал которой может превзойти ожидания сейчас, когда программирование становится более компонентно-ориентированным.

Функциональное программирование поддерживает прагматические принципы организации вычислений, такие как гибкость ограничений, неизменяемость данных и строгость результата. Прагматические принципы поддерживаются системой программирования, точнее, разработчиками СП. Методы поддержки таких принципов обычно не определены в языке ФП, их выбирает разработчик системы ФП.

Гибкость ограничений. Для предотвращения необоснованных простоев памяти поддерживается оперативный анализ достижимости данных, освобождение и повторное использование памяти, хранящей недостижимые данные. Бывает, что памяти не хватает не на всю задачу, а только на отдельные блоки данных, возможно, не так уж важные для ее решения. Возникают ситуации, когда одни из этих частей исчерпаны, а другие не используют выделенное им пространство. В системах ФП эту проблему решает специальная функция - «сборщик мусора» (garbage collector - GC), которая пытается автоматизировать перераспределение или освобождение памяти, когда какой-либо области памяти недостаточно. Это означает, что данные могут быть любого размера. Новые реализации сборщика мусора эффективно учитывают преимущества восходящих процессов на больших объемах памяти. Многие новые ЯиСП теперь включают такие механизмы независимо от парадигмы.

Неизменяемость данных. Представление каждого результата функции помещается в новую часть свободной памяти без искажения аргументов этой функции, они могут быть полезны для других функций. В любое время возможен доступ к данным, ранее полученным в результате вычислений. Таким образом, значительно упрощается отладка программ и обеспечивается обратимость любых действий.

Строгость результата. Любое количество результатов функции может быть представлено в виде единой символьной формы, из которой при необходимости может быть выбран желаемый результат. Часто этот принцип интерпретируется как требование единственного или даже исключительно скалярного результата функции, что приводит к сомнениям в законности использования функций целочисленного деления, извлечения корня, обратных тригонометрических функций и многих других категорий математических функций. В этом плане примечательны соображения Г.М. Фихтенгольца, приведённые в предисловии к учебнику по матанализу. Он отмечает, что если прямые функции часто бывают однозначными, то обратные, как правило, таким свойством не обладают.

Представление алгоритмов в виде функциональных программ дает практически важные следствия. Мета-программирование, гомоиконность и анонимность определений вытекают из семантических принципов. Прагматические принципы приводят к моделям непрерывности процессов, обратимости действий и унарных функций. Эти следствия служат основой для интуитивного оперирования функциональными моделями, которые позволяют ставить и понимать прямой компьютерный эксперимент.

Мета-программирование является следствием принципа универсальности, который позволяет обрабатывать и создавать представления программ так же, как любые данные. Данные, представляющие значение или функцию, могут быть составлены из частей вплоть до букв. Любые части программ могут быть вычисляемыми фрагментами. Возможны смешанные и частичные вычисления, оптимизирующие преобразования, макрогенерация и многое другое, необходимое при создании операционных систем и оптимизирующих СП. Всё это можно делать практически на базе любой СП, но обычно нет ни такой традиции, ни примеров.

Гомоиконность означает внешнее подобие представлений функций и обрабатываемых данных. Любую структуру данных можно рассматривать как схему выбора ветвей функции, что позволяет придавать программам удобочитаемый и легко отлаживаемый вид. В частности, можно выбрать списочное представление БНФ и синтаксически конструировать анализаторы, компиляторы и другие обработчики данных.

Анонимность определений поддержана взаимозаменяемостью в программе имён и их значений. Это позволяет не загружать текст программы именами однократно используемых определений, особенно передаваемых через параметры при определении отображений. Такая возможность полезна для выполнения факторизации программ, что напрямую следует из принципа равноправия параметров с учетом принципа универсальности. Части данных, любые подформулы могут быть значениями параметров функции. Это позволяет использовать концепцию выбора проекций или слоёв программы, а также частичного вычисления. Благодаря факторизации можно выделять из программы проекции, аналогичные схеме программы, допускающей верификацию.

Следствия поддержки прагматических принципов в системах ФП формируют и поддерживают интуитивные образы, такие как непрерывность процесса (бесконечность), обратимость действий и унарные функции, которые обеспечивают основу для построения функциональных моделей. Кроме того, комплекс семантических и прагматических принципов обеспечивает поддержку подготовки функциональных моделей программ для организации параллельных вычислений, которые могут быть сведены к комплексам независимых, недетерминированных потоков.

Непрерывность процессов интуитивно следует из прагматической поддержки принципа гибкости ограничений.

Обратимость действий основана на иллюзии неизменяемости данных, механизмы которой скрыты в системе программирования.

Унарные функции связаны с принципом строгого результата. Для любой функции с произвольным количеством параметров можно построить ее эквивалент с одним параметром, являющимся структурой из первоначальных параметров. Кроме того, возможность перейти от списка параметров или результатов к одному аргументу или строгому результату и обратно позволяет отойти от обычной схемы операций, которая сопоставляет два операнда одному результату, к операциям, сопоставляющим один набор операндов другому набору результатов.

Параллелизм опирается на конкретизацию общего комплекса семантических и прагматических принципов, который позволяет рассматривать любой объем представленных данных и при необходимости реорганизовывать пространство потоков. Такой комплекс делает ФП удобным для работы с программами, направленными на организацию параллельных процессов. Прежде всего, это принцип равноправия, который гарантирует одинаковый контекст при вычислении параметров функции. Становится возможным представлять независимые потоки и объединять много-поточные или многопроцессорные программы в общий программный комплекс. Кроме того, в параллелизме полезны принципы строгого результата и универсальности, что позволяет при необходимости учитывать любое количество представленных потоков

и реорганизовывать пространство потоков. Происходит конкретизация принципов для нужд параллелизма, прецеденты уже встречаются в конкретных ЯП.

Оттеснение маловероятных вычислений направлено на предотвращение затрат на выполнение чрезмерного количества потоков, соответствующих слишком редким ситуациям. При решении задач параллельных вычислений (ПВ) важен принцип равноправия программ с данными, традиционно востребованный в задачах операционных систем и управления процессами. Любой фрагмент, выполнение которого маловероятно или невозможно, может быть перемещён из представления функции в отложенное действие — «ленивые» вычисления.

Балансировка нагрузки сокращает реальное время выполнения много-поточной программы, что можно рассматривать как распространение принципа гибкости на временные ограничения. Ленивые или преждевременные вычисления дают возможность быстро и оперативно перераспределять нагрузку. Пример поддержки такого принципа даёт язык mpC/

Само-применение в виде рекурсивных функций обычно рассматривается как осложнение, которое приводит к опасному росту стека. Здесь следует отметить, что многие системы ФП предлагают ряд практичных решений этой проблемы. Кроме того, в ФП существуют отложенные действия, мемоизация, восходящая рекурсия, методы динамического программирования и оптимизация рекурсий за счет сведения к циклам, что во многих случаях позволяет практически устранить чрезмерное разбухание стека. А поддержку работы со стеком в рамках принципа гибкости ограничений удаётся поддерживать более эффективно, чем в большинстве ЯиСП.

Пространство итераций используется в качестве основного параметра для управления распараллеливанием циклов. Любое конечное множество может работать как пространство итераций для определенной на нем функции. Техника и концепция пространств итераций убедительно поддержаны в языке Sisal, в котором такие пространства строятся над перечислимыми множествами с использованием операций скалярного и декартового произведения.

Несколько сложнее с прагматическими принципами, требующими пересмотра системных решений на уровне разработки СП. Здесь важную роль играет автоматическое распараллеливание, реже встречаются идентичность повторных прогонов и представление много-контактных фрагментов.

Автоматическое распараллеливание заключается в извлечении из программы автономных частей, допускающих независимое выполнение.

Идентичность повторных прогонов для целей отладки программ и измерения их производительности оказались проблемой при переходе к суперкомпьютерам. Обнаружилось, что при слишком большом числе процессоров исчезает нужная для отладки и измерений возможность пронаблюдать повторное исполнение программы. При очередном прогоне могут быть сбои на других процессорах.

Много-контактные фрагменты, такие как схемы управления или операции, имеющие несколько операндов и дающие более одного результата, на первый взгляд, противоречат принципу строгого результата. Однако такая возможность позволяет строить много-поточные функции, которые принимают параметры от ряда одно-ранговых потоков и генерируют ряд результатов в терминах числа потоков. Таким образом, можно, как и в функциональном языке параллельного программирования Sisal, переключиться на операции, которые отображают одну строку операндов в другую строку результатов. Это может соответствовать структуре некоторых особо эффективных аппаратных узлов и, таким образом, допускать представление более эффективных решений.

При переходе к многократно применяемым программам и ПВ **успех приложения и производительность программ становятся важнее, чем их формальная корректность и эффективность.**

В дополнение к принципам и следствиям в реальных ЯиСП производственное ФП обычно включает механизмы практичного компромисса, которые внешне в ЯП выглядят как специальные функции. Например, Lisp 1.5, Clisp, Cmucl, Clojure и другие члены семейства Lisp обычно предоставляют следующие механизмы такого рода:

- контроль типов данных смягчает *принцип универсальности* функциями статического и динамического анализа типов данных;

- схемы циклов, моделирующие несколько расширенное множество знакомых механизмов управления вычислениями, позволяют преодолеть типичные опасения по поводу сложности реализации *принципа само-применения*;

- возможность восстановления данных при использовании деструктивных функций, имеющих безопасные аналоги, позволяет исключить чрезмерное потребление памяти, частично отклоняясь от *принципа неизменности данных*;

- программируемые прогнозы объема памяти и скорости выполнения позволяют по мере необходимости выполнять распределение памяти, нейтрализуя грубоватые механизмы *принципа гибкости ограничений*;

- псевдо-функции, помимо генерации результата, выполняют нагрузку в виде воздействия на внешнюю память или взаимодействия с устройствами, включая операции ввода-вывода и файловые операции, что несколько влияет на *принципы равноправия параметров и неизменяемости данных*;

- мемоизация позволяет радикально снижать сложность многократно повторяемых вычислений, делая понятную уступку успешному опыту против *принципов строгого результата и верификации*.

Общее взаимодействие принципов, следствий, приложений и практичных компромиссов в системах ФП представлено в Таблице 4/1.

Таблица *4.*1
Ключевые взаимосвязи принципов ФП

	Семантика	Прагматика
Принципы	универсальность само-применение равноправие параметров	гибкость ограничений неизменяемость данных строгость результата
Следствия	метапрограммирование гомоиконность анонимность определений	непрерывность процессов обратимость действий унарные функции
Приложения к ПВ	оттеснение мало вероятных вычислений балансировка нагрузки пространства итераций	автоматическое распараллеливание идентичность повторных прогонов много-контактные фрагменты
Практичный компромисс	контроль типа данных схемы циклов восстановление данных	программируемый учет прогнозов псевдо-функции доступа к устройствам мемоизация

Парадигмальные ошибки, обнаруженные при работе знакомых СП на современном многопроцессорном и сетевом оборудовании, показывают, что некоторые из них были просто невидимы до появления сетей, мобильных устройств и суперкомпьютеров. Следствия семантических и прагматических принципов ФП и высокая моделирующая сила аппарата функций, дополненная специальными функциями практичных компромиссов, позволяют дополнять основные парадигмы механизмами параллельных вычислений для практической работы по повышению производительности программы.

Функциональное программирование является основным полигоном исследования и разработки средств и методов верификация программ, вытекающих из принципа само-применения с методами рекурсии, математической индукции и логики. Большинство систем верификации программного обеспечения создаются в рамках ФП. Это увеличивает надежность и безопасность программ, хотя и не позволяет решить проблему корректности в полной мере. Трудности связаны с недостаточностью классической логики по отношению к неклассическим логикам программирования, а также с некоторыми интуитивными ошибками при оценке истинности логических и вероятностных формул. ФП активно применяется для генерации программ и выполнения динамически конструируемых прототипов программ, а также для систем, применяемых в областях с низкой кратностью повторения отлаженных решений (например, в учебе, проектировании, творчестве и научных исследованиях), ориентированных на решение новых задач, оперативные изменения, уточнения, улучшения, адаптацию и т. п.

Как уже отмечалось, общая характеристика постановок задач ФП может кратко выражена следующим образом:

- Исследуется предметная область, обладающая задачами с недостаточной степенью изученности для производственного программирования.

- Следует выбрать символьное представление данных для этой области и отладить систему универсальных функций, пригодных для создания прототипов решения актуальных задач из этой области.

- Программирование начинается с принятия решений относительно необходимых вычислений (**E**) и структур данных (**S**) — иерархия списков. Потом может быть выполнено именование многократно используемых неизменяемых данных (**M**) и при необходимости приняты решения по порядку вычислений с однозначным строгим результатом (**C**).

Полученная последовательность приоритетов принятия решений **ESMC** может рассматриваться как характеристический признак функционального программирования (ФП) и языков, поддерживающих эту парадигму.

4.2. Основы

Сформулированная Джоном Маккарти (1958) концепция символьной обработки информации восходит к идеям Черча и других математиков, известным как лямбда-исчисление с конца 20-х годов прошлого века. Выбирая лямбда-исчисление как теоретическую модель, положенную в основу языка Lisp, Маккарти предложил рассматривать функции как общее базовое понятие, к представлению и реализации которого достаточно естественно могут быть сведены все другие понятия, возникающие в практике программирования. Такое сведение вовсе не означает, что все понятия сваливаются в одну кучу, что исчезают границы между понятиями. Сведение выполнено так, что при сохранении всех понятийных границ выстроено более общее пространство, в рамках которого эти понятия упорядочены и могут взаимодействовать согласно формальным определениям разных категорий функций. Унификация понятий концептуального минимума (Pure Lisp) для безмашинного обучения методам символьной обработки представлений функций, включая отображения, отложенные действия, и другие функции высших порядков, использующие исключительно чистые функции без побочных эффектов представлена в таблице 4.2.

Таблица *4*.2

Унификация понятий языка Lisp.

Конструкция	*Примеры представления*	*Трактовка*	*Пояснение*
Встроенная константа	Nil	Представление функции без параметров, результатом которой является пустой список.	Результаты таких функций хранятся непосредственно в памяти. Их получение не требует вычислений.
Элементарное значение	List X A	Представление неопределённой функции, которая может быть в дальнейшем определена как значение переменной или представление конкретной функции.	
Идентификатор	A X ATOMIC IDENT	Представление функции, арность, категория и определение которой задаются разными средствами связывания атомов с их смыслом. (Связывание аргументов с параметрами при вызове функции или именование определения.)	
Именованная константа	A	Представление функции без параметров, в любой позиции программы выдающей ранее заданное значение.	
Переменная	X	Представление функции без параметров, результат которой может зависеть от контекста, например, от области видимости при вызове других функций.	
Составное значение	‘(A B C) ‘(A (B C) D) ‘(A . B)	Результат унарной функции QUOTE, препятствующей вычислению своего аргумента: (QUOTE (A B C)) (QUOTE (A (B C) D)) (QUOTE (A . B))	Блокировка вычислений, в частности для организации отложенных действий.
Вызов функции	(FN LIST-FRM)	Представление выражения в виде списка из представления функции и представлений её параметров в виде выражений. При	Общая схема вычислений. Локальное связывание

		необходимости результат функции вычисляется с помощью универсальной функции APPLY.	аргументов со значениями параметров, сохраняемое в стеке.
Выражение (форма)	X FNAME (CAR ‘(a b c))	Представление аргумента универсальной функцией EVAL, пригодное для вычисления значения этого аргумента.	Вывод значений по представлению выражений.
Ветвление	(COND ((EQ X A) Y) ((ATOM X) D) -- -- (T Z))	Специальная мультифункция над произвольным числом аргументов, каждый из которых является списком из представлений предиката и соответствующей ему ветви. При пустом списке – результат Nil.	Метод организации частичных вычислений.
Определение анонимной функции	(LAMBDA (x y) (expr x y))	Результат специальной бинарной функции LAMBDA, первый аргумент которой – список аргументов определяемой функции, а второй представляет выражение, задающее её тело.	Конструирование представления функции.
Именование локального определения	(LABEL NAME DEF)	Результат специальной бинарной функции LABEL, связывающей новое имя, заданное первым аргументом, с представлением функции, заданной вторым аргументом, в котором это связывание локализовано.	Поддержка многократного применения функции.
Вычисление	(EVAL expr)	Результат универсальной функции EVAL, анализирующей структуру представления выражения и выбирающей метод вычисления значения выражения.	Поддержка управления ходом вычислений, включая возобновление отложенных действий.

Практическое программирование обычно выходит за круг константно-определённых программ, поддерживаемых чистыми функциями. Для целей решения новых задач, отладки и оптимизации программ на компьютере возникает расширение пространства используемых категорий функций, обладающих разными побочными эффектами. Трактовка основных понятий программирования, унифицированных как функции, для практичного расширения средств отладки и оптимизации программ и решения новых задач приведена в таблице 4.3.

Таблица *4*.3

Трактовка основных понятий программирования.

Конструкция	*Примеры представления*	*Трактовка*	*Пояснение*
Вывод данных	(PRINT X)	Представление тождественной псевдо-функции PRINT с побочным эффектом, заключающемся в размещении эквивалентного её аргументу текста на экране или в	(PRINT X) = X Результат совпадает с аргументом, поэтому размещение в

		заданном файле.	программе вывода данных может не влиять на ход выполнения программы.
Ввод данных	(READ)	Представление псевдо-функции без параметров READ, побочным эффектом которой является конструирование представления данного, эквивалентного строке, набранной или размещённой в заданном файле.	Включение в программу средств ввода данных позволяет управлять ходом вычислений без редактирования текста программы, что позволяет повысить надёжность отладки.
Обработка прерываний, ошибок, исключений, неожиданностей и т.п.	(ERROR N “message” continuation)	Представление псевдо-функции, обеспечивающей для заданного номера события поясняющее сообщение и возможное продолжение процесса вычислений. По умолчанию – восстановление начального контекста.	Поддержка вычислений в стиле бэктреккинга.
Элементарное значение	123 3.14 4/5 «строка»	Представление само-определимой функции без аргументов, результатом которой является её собственное представление.	Результаты такой функции хранятся непосредственно в памяти. Их получение не требует вычислений.
Операции над числами	(+ 1 2 3 4) (* 1 2 3 4) (- 1 2 3 4) (/ 1 2 3 4)	Представление мультифункций над произвольным числом аргументов. Аддитивные операции при пустом списке аргументов вырабатывают число 0. Мультипликативные – число 1.	(+ 1 2 3 4) = 1+2+3+4 (* 1 2 3 4) = 1*2*3*4 (- 1 2 3 4) = 1-2-3-4 (/ 1 2 3 4) = 1/2/3/4
Арифметические выражения	(+ (* 3 5 D) (- A 8) (/ 1 2 X))	Представление результатов вычислений не требует определения типов реализуемых чисел, зависящих от формата кода или длины машинного слова.	(/ 1 2) = ½ Длина целых чисел не ограничена, результат целочисленного деления может быть представлен как дробь без потери точности, точность вещественных может быть задана.
Именование глобальной функции	(DEFINE NAME DEF)	Результат специальной бинарной функции DEFINE, связывающей новое имя, заданное первым аргументом, с глобальным определением функции, представленным вторым аргументом.	Поддержка комбинаторики независимых определений функций.
Конструирование представления именованной глобальной функции с параметрами	(DEFUN NAME ARGS DEF)	Результат специальной функции DEFUN, конструирующей по списку аргументов и определяющему выражению новое глобальное определение функции и связывает его с именем.	

Область видимости переменных	(LET LIST EXPR)	Специальная функция LET строит область видимости, в которой определены значения переменных согласно списку LIST, используемых при вычислении выражения EXPR.	Поддержка иерархии наследуемых определений.
Область видимости вспомогательных функций	(FLET LIST REXPR)	Специальная функция FLET строит область видимости, в которой определены вспомогательные функции согласно списку LIST, используемых при вычислении выражения REXPR.	
Конструирование специальных функций	(MACRO)	Специальная функция MACRO создает определения новых специальных функций, обрабатывающих представления своих параметров без их предварительного вычисления.	Поддержка альтернативных методов вычислений.
Цикл	(LOOP …)	Специальная функция, один из параметров которой является предназначенным для многократного вычисления телом цикла, а остальные используются при управлении кратностью вычисления тела цикла.	Поддержка традиционных схем представления программ.
Императивные вычисления	(PROG (V ..) ….)[101]	Специальная функция, выполняющая последовательное вычисление выражений, рассматриваемых как операторы. Результат выделен функцией RETURN или определён последним оператором.	
Параллельные вычисления	(MULTIPLY …)	Специальная функция, выполняющая вычисление своих параметров в произвольном порядке, не заданном заранее. Значения всех параметров доступны объемлющим функциям.	Подготовка параллельных вычислений.
Компиляция функций	(COMPILE …)	Специальная функция, побочный эффект которой заключается в создании кода функции, оттесняющего её символьное определение.	Оптимизация отлаженных функций по скорости выполнения.
Структуро-разрушающие функции	(RPLACA X Y) (RPLACD X Y) (CONC AL BL) (MAPCON FN AL)	Представление функций с побочными эффектами, вызванными использованием памяти аргументов при конструировании результата, при наличии их чисто функциональных эквивалентов: (RPLACA X Y) = (CONS Y (CDR X)) (RPLACD X Y) = (CONS (CAR X) Y) (CONC AL BL) = (APPEND AL BL) (MAPCON FN AL) = (MAPLIST FN AL)	Поддержка синхронизации независимых участков программы, возникающих при вычислении рекурсивных функций, независимых параметров, выполнении итераций и т. п.

Управление обработкой информации в лямбда-исчислении осуществляется в рамках иерархии свободных и связанных переменных, реализуемых с помощью ассоциативной таблицы соответствия

[101] Функция PROG позволяет использовать императивный стиль программирования без нарушения функционального стиля, Она допускает присваивания своим рабочим переменным, которые возникают при её вызове и исчезают при выходе из неё. Это не создаёт изменений внешнего контекста её вызова. Она решает вопрос «ЧТО?», императивным методом «КАК?», что не противоречит идее декларативного программирования.

символов и их смысла. Обработка представляется посредством правил интерпретации выражений, построенных из всюду определенных функций, аргументы которых могут быть упорядочены. По степени общности такое построение сравнимо с аксиоматической теорией множеств.

Следует отметить некоторую разницу в понимании принципов ФП, сложившуюся в теории и практике программирования. Эта разница чётко проявилась при стандартизации языка Lisp в 1980-е годы в виде принятия двух стандартов: LISP1 – академический и LISP2 – производственный. Теоретически достаточно исследовать чисто функциональные лаконичные представления программ, поведение которых не зависит от побочных эффектов. Результат программы может быть получен системой редукций её представления. Процессы применения редукций можно выбирать в зависимости от стратегии вычислений. На практике основная трудоёмкость связана с отладкой программы на базе конкретной СП, поддерживающей определённую стратегию вычислений или дающей возможность явного управления ходом выполнения программы в зависимости от данных. Чисто функциональная программа при оптимизирующей компиляции[102] может быть сведена к её формальному результату без генерации исполнимого кода[103].

По отношению к проблемам определения ЯиСП основные идеи ФП сложились при реализации языка Lisp и в работах Венской лаборатории IBM в начале 1960-ых годов. Эти идеи оказались трудными для восприятия в пионерскую эпоху программирования, но в настоящее время их популярность растет, что и обуславливает целесообразность фундаментальных исследований в сфере ФП, проектирования и моделирования.

С конца 70-х годов появились Lisp-процессоры, доказавшие, что неэффективность ФП обусловлена характеристиками оборудования, а не стилем программирования[104]. Функциональные мини-языки хорошо показали себя и при решении задач аппаратного уровня. Все это превращает ФП в практичный и перспективный инструментарий. Такая схема подтверждается самой историей развития диалектов языка Lisp и родственных ему ЯП.

Изучение ФП начинается с овладения техникой работы с так называемыми «чистыми», как бы математическими, идеальными функциями в области константных вычислений. Для реализации таких функций характерен отказ от необоснованного использования присваиваний и низкоуровневого управления вычислениями в терминах передачи управления. Такие функции удобны при отладке и тестировании благодаря независимости от контекста описания и предпочтения явно выделенного чистого результата. Трудоемкость отладки композиций из хорошо определенных функций растет аддитивно, а не мультипликативно. Кроме того, системы из таких функций могут развиваться в любом направлении: сверху вниз и снизу-вверх (а также расширяясь и сужаясь, если понадобится). Можно быстро продвинуться по сложности решаемой задачи, не отвлекаясь на синтаксическое разнообразие и коллизии при обработке общих данных. Для обучения такому стилю программирования на языке Lisp был создан его диалект Pure Lisp и определён его интерпретатор.[105] Концептуально близкие идеи «структурного программирования» были сформулированы более чем через десять лет.

Если в качестве данных допускать не только значения, но и символьные формы для вычисления этих значений, то вопрос о времени вычисления аргументов можно решать не столь категорично. Кроме обычных функций, аргументы которых вычисляются предварительно, в ряде случаев можно рассматривать и реализовывать специальные функции, способные обрабатывать аргументы нестандартным способом по любой заданной схеме, отложенные действия (lazy evaluation).

Функции могут быть частично определёнными, отображающими часть множества, и многозначными, когда хотя бы одному значению аргумента соответствует два или более значений функции. Прежде всего, понятие «функция» обогащается представлением о псевдо-функциях, которые используются с целью предоставления аппаратных, зависимых от устройств действий (ввод/вывод, сообщения, рисование и т. п.).

---

[102] Время на преобразования или компиляцию программ не учитываются при замерах эффективности, а функциональные программы вполне можно компилировать. Стиль программирования влияет на продуктивность программирования и на жизнеспособность программ, на длину их жизненного цикла.

[103] Обычно в составе оптимизаций компиляторы содержат анализ на константные вычисления и их выполнение на этапе компиляции. Если в программе «3+5», то в исполнимом коде будет «8» без суммирования чисел. Чисто функциональная программа не содержит ввода данных и подчинена принципу неизменяемости данных,

[104] В те времена малый объём памяти, сложность работы с символьными данными — ещё не было понимания пользы от такой обработки, отсутствие поддержки рекурсии, низкая надёжность работы внешней памяти. В современных архитектурах эти характеристики существенно улучшены.

[105] К языку Lisp нет смысла сводить все языки программирования, но их можно моделировать, как и на любом другом универсальном языке. Вопрос в эффективности и целесообразности. Теоретически все полные по Тьюрингу языки взаимосводимы. Их принципиальная разница лежит в сфере практики — отличается продуктивность языка и производительность систем на разных классах задач.

Они фактически осуществляют известный побочный эффект[106] в результате работы конкретного оборудования и ОС – минимального контекста исполнения любой практически полезной программы.

Формально все псевдо-функции обязательно выполняют и отображение аргументов в результаты, что позволяет им равноправно участвовать в любой позиции формулы, задающей вычислительный процесс. Формальный результат сопровождается дополнительными эффектами. Этот переход обеспечивает, при необходимости, корректное моделирование всей традиционной программотехники, включая присваивания, передачи управления, системные вызовы, обработку файлов и доступ к любым устройствам. Все эти непредсказуемо сложные машинно-зависимые реалии при функциональном стиле программирования локализованы, наращиваются на ранее отлаженный каркас функционирования программы, их представления могут быть четко отделены от сущности решаемой задачи. ФП повышает продуктивность отладки программ благодаря созданию коллекций абстрактных лаконичных универсальных функций, приспособленных к многократному применению в разных программах.

Здание ФП получает логическое завершение на уровне определения функций высших порядков, удобных для синтаксически управляемого конструирования программ на основе спецификаций, типов данных, визуальных диаграмм, формул и т. п. Функциональные программы могут играть роль спецификации обычных императивно-процедурных программ. Иногда такой переход не вызывает затруднений. Факториал можно определить рекурсивно как сведение к значению функционала от предыдущего числа, но столь же понятно и определение в виде цикла от одного до N. На языке Sisal для этого не требуется даже цикла, достаточно задать границы области, элементы которой перемножаются (* 1, ,N). Конечно, числа Фибоначчи легко порождать с помощью рекурсивного восходящего процесса, но и цикл с заданной границей заработает вполне практично. Однако встречаются несложные задачи, для которых такой переход не столь прост. Отнюдь не любая обработка произвольной последовательности легко излагается в терминах векторов, и многие задачи на больших графах могут весьма сложно приводиться к итеративной форме. Заметные трудности в процесс сведения *рекурсии* к итерации создает динамика данных и конструируемые функции. Даже реализация равенства для произвольных структур данных при неизвестной размерности и числе элементов является не простым делом. Известно, что лаконичность рекурсии может скрывать нелегкий путь. А. П. Ершов в предисловии к книге П. Хендерсона привел поучительный пример не поддавшегося А. Чёрчу решения задачи о рекурсивной формуле, сводящей вычитание единицы из натурального числа к прибавлению единицы {1 –1 = 0 | (n +1) -1 = n}, полученного С. Клини лишь в 1932 году, когда многие математики уже готовы были смириться с неразрешимостью этой задачи.

Алгоритм	*Примечание*
n –1 = F(n, 0, 0)	Сведение к вызову вспомогательной функции.
где	
F(x, y, z) =	Вспомогательная функция.
если (x = 1) то 0	Объявление значения от 1.
иначе если ((y +1) = x) то z	Проверка достижения предыдущего числа.
иначе F(x, y +1, z +1)	Шаг рекурсии с наращиванием параметров.

Пример. 4.0. Выражение «-1» через «+1»

Решение получилось через введение формально усложненной функции F со вспомогательными параметрами, которое противоречит интуитивному стремлению к монотонности при движении от простого к сложному.

Возможны ограничения на типы данных, допускаемых в качестве аргументов, в таком случае речь идет о частичных функциях. Эти функции должны выяснять допустимость фактических параметров и сообщать о несоответствии. Удобно, если часть такой работы берет на себя компилятор в классической традиции статического контроля правильности типов данных, но динамический контроль типов данных в условиях, характерных для современных информационных сетей, может быть надежнее, чем традиционный статический анализ, сложившийся на замкнутых, защищенных от несанкционированного доступа конфигурациях, обеспечивающий гарантии сохранения скомпилированного кода программы при его использовании. (Имеется в виду вероятность искажения скомпилированного кода при его эксплуатации на компью-

[106] **Ввод** данных включает в программу ранее не известное значение, не подвластное статическому анализу, что мешает логическим выводам относительно правильности программы. **Цикл** обычно меняет значение параметра цикла или использует предикат, зависящий от изменяемых данных, что противоречит принципам чистого ФП

тере в сетях.) Это приводит к компромиссу в виде объектно-ориентированного программирования, допускающего динамический контроль типов данных. [107]

Важно отметить, что преимущества ФП обусловлены не только семантикой используемых ЯП, но и особенностями его поддержки в СП:

– представления данных (чисел, строк, имён, списков) не ограничены по длине;

– полностью автоматизировано первичное и повторное распределение памяти – «сборка мусора»;

– выделен комплекс базовых средств для программирования без побочных эффектов в памяти программы, достаточный для представления ленивых вычислений, отображений[108] и других функций высших порядков;

– встроены средства управления вычислениями, расширяющие возможности СП в направлении основных ПП.

4.3. Функциональные ЯП

Идеи ФП достаточно полно поддержаны в проекте Lisp 1.5, выполненном Дж. Маккарти и его коллегами. В этом исключительно мощном языке не только реализованы основные средства, обеспечившие практичность и результативность ФП, но и впервые опробован целый ряд поразительно точных построений, ценных как концептуально, так и методически и конструктивно, понимание и осмысление которых слишком отстает от практики применения. Понятийно функциональный потенциал языка Lisp 1.5 в значительной мере сохранён стандартом Common Lisp и других диалектах, многое унаследовано ЯП, поддерживающими другие ПП. Краткая характеристика языка Lisp и его диалектов и приемников приведена в Приложении 3.

Языки ФП достаточно разнообразны. Существует и активно применяется более трехсот диалектов языка Lisp и родственных ему языков на базе лямбда-исчисления: Interlisp, muLisp, Clisp, Scheme, ML, Cmucl, Logo, Hope, Erlang, Clean, Sisal, Haskell, Miranda, Clojure, Racket и др.[109] При сравнении языков и парадигм программирования часто классифицируют функциональные языки по следующим критериям: «ленивый» или аппликативный, последовательный и параллельный. Например, ML является аппликативным и последовательным, Erlang – аппликативным и параллельным, Haskell – «ленивым» и последовательным, а Clean – параллельным и «ленивым». Scheme, ML, Hope, Haskell – типичные представители академического стандарта LISP1, а Common Lisp, lisp, Sisal, Cmucl – производственного стандарта LISP2[110]

В рамках проекта .Net выполнено большое число реализаций различных ЯП, в числе которых Haskell, Sml, Scheme, Mondrian, Mercury, Perl, Oberon, Component Pascal, Разработан F# – новый язык ФП. Еще большее разнообразие предлагает проект DotGNU**,** пытающийся отстоять приоритет в области разработки переносимого ПО. Развиваются линии учебного и любительского программирования и методично осваивается техника выстраивания иерархии абстрактных машин при определении ЯП.

Разработка языков функционального программирования (ЯФП) и приспособленность средств ФП к быстрой отладке, верификации, их лаконизм, гибкость, конструктивность и моделирующая сила позволяют рассматривать ФП как основу информационной среды обучения современному программированию на всех уровнях проблематики от алгоритмизации до включения в социальный контекст приложений разрабатываемых ИС. Согласно международным рекомендациям обучение программированию в университетах следует начинать именно с ФП.

4.4. Отображения и функционалы

Выразительная сила ЯФП наиболее результативно проявляется в программировании отображений. Отображения обычно используются при анализе и обработке данных, представляющих информацию

---

[107] На вопрос студента: «Что может дать функциональный стиль и традиция реализации функциональных систем программирования?» можно спрогнозировать, что скорее всего постепенно сформируется практика профессиональной разработки прототипов сложных систем подобно проектно конструкторским бюро в производстве приборов, машиностроении и строительной индустрии.

[108] Отображающая функция должна быть применима к элементам отображаемого множества, т. е. нужно согласование типов или видов данных.

[109] https://en.wikipedia.org/wiki/Comparison_of_functional_programming_languages – 30 языков
https://en.wikipedia.org/wiki/List_of_Lisp-family_programming_languages — 40 языков
http://ru.progopedia.com/paradigm/functional/ - 52 языка

[110] Этот стандарт нарушает принцип "функциональной чистоты". Стандартом попытались охватить основные сферы активного применения языка Lisp, принцип функциональной чистоты тогда ещё не доминировал. Впрочем, этот стандарт меньше по объёму, чем для C++. Следует отметить, что стандарты вообще изменяют редко. Это трудоёмкий и дорогостоящий процесс, требующий для утверждения примерно пары лет календарного времени, не считая собственно разработки. Кроме того, всегда имеют место опасения, что возникнет смена стандарта.

разной природы. Вычисление, кодирование, трансляция, распознавание – каждый из таких процессов использует исходное множество цифр, шаблонов, текстов, идентификаторов, по которым конкретная отображающая функция находит пронумерованный объект, строит закодированный текст, выделяет идентифицированный фрагмент, получает зашифрованное сообщение. Так работает любое введение обозначений – от знака переход к его смыслу.

Определение отображений – ключевая задача информатики. Построение любой информационной системы сопровождается реализацией большого числа отображений. Сначала выбираются данные, с помощью которых представляется информация. В результате по данным можно восстановить представленную ими информацию – извлечь информацию из данных (по записи числа восстановить его величину). Потом конструируется набор структур, достаточный для размещения и обработки данных и программ в памяти компьютера (по коду команды можно выбрать хранимую в памяти подпрограмму, которая построит новые коды чисел или структур данных).

Говорят, что отображение существует, если задана пара множеств и отображающая функция, для которой первое множество – область определения, а второе – область значения. При определении отображений, прежде всего, должны быть ясны ответы на следующие вопросы:

– что представляет собой отображающая функция;
– как организовано данное, представляющее отображаемое множество;
– каким способом выделяются элементы отображаемого множества, передаваемые в качестве аргументов отображающей функции.

Это позволяет задать порядок перебора множества и метод передачи аргументов для вычисления отображающей функции. При обходе структуры, представляющей множество, отображающая функция будет применена к каждому элементу множества.

Проще всего выработать структуру множества результатов, подобную исходной структуре. Но возможно, что не все полученные результаты нужны, или требуется собрать их в иную структуру, поэтому целесообразно заранее прояснить еще ряд вопросов:

– где размещается множество полученных результатов;
– чем отличаются нужные результаты от полученных попутно;
– как строятся итоговые данные из отобранных результатов.

При функциональном стиле программирования ответ на каждый из таких вопросов может быть дан в виде отдельной функции, причем роль каждой функции в схеме реализации отображения четко фиксирована. Схема реализации отображения может быть представлена в виде определения, формальными параметрами которого являются обозначения функций, выполняющих эти роли. Такое определение называют функционалом. Говоря более точно, функционал может оперировать представлениями функций в качестве аргументов или результатов.

Функции, выполняющие конкретные роли, могут быть достаточно общими, полезными при определении разных отображений, – они получают имена для многократного использования в разных системах определений. Но они могут быть и разовыми, нужными лишь в данном конкретном случае, – тогда можно обойтись без их имен и использовать анонимное определение непосредственно в точке вызова функции.

Таким образом, определение отображения может быть разбито на части (функции и функционалы) разного назначения, типичного для многих схем информационной обработки. Это позволяет упрощать отладку систем определений, повышать коэффициент повторного использования отлаженных функций. Можно сказать, что отображения – эффективный механизм абстрагирования, моделирования, проектирования и формализации крупномасштабной обработки информации. Возможности отображений в информатике значительно шире, чем освоено практическим программированием, но их применение требует дополнительных пояснений.

Отказ от барьера между представлениями функций и значений дает возможность ФП использовать символьные выражения как для изображения заданных значений, включая любые структуры над числами и строками, так и для представления функций, обрабатывающих любые данные. (Напоминаем, что определение функции представляется как данное.) Таким образом, функционалы в ЯФП – это функции, которые используют в качестве аргументов или результатов представления других функций, а не собственно функции как принято говорить. При построении функционалов переменные могут играть роль имен функций, представления которых находятся во внешних формулах, использующих функционалы. Пока в работах по большеобъёмным данным получила признание методик map-reduce.

Рассмотрим технику использования функционалов на упражнениях с числами и покажем, как от простых задач перейти к более сложным.

Определение на Clisp	*Примечание*
(DEFUN map-el(fn xl) (COND (xl (CONS (FUNCALL fn (CAR xl)) (map-el fn (CDR xl))))))	Поэлементное преобразование XL с помощью функции FN. Пока XL не пуст Присоединяем результат FN от головы XL к списку преобразованных остальных элементов.
(map-el #'1+ xl) [111]	Следующие числа
(map-el #'CAR xl)	«головы» элементов = CAR
(map-el #'length xl)	Длины элементов

Пример 4.1. Отображение элементов списка с помощью заданной функции

Числа представляются в языке Lisp как специальный тип атома, называемый псевдо-атом. Атом такого типа состоит из указателя с тэгом, специфицирующим слово как число, тип этого числа и адрес собственно числа произвольной длины. В отличие от обычного атома, одинаковые числа при хранении могут не совмещаться.

Определить функцию покомпонентной обработки двух списков с помощью заданной функции FN.

Определение на Clisp	*Примечание*
(DEFUN map-comp (fn al vl) (COND (al (CONS (FUNCALL fn (CAR al) (CAR vl)) (map-comp (CDR al) (CDR vl))))))	fn покомпонентно применить к соответственным элементам al и vl Пока AL не пуст Присоединяем результат FN от голов AL и VL к списку преобразованных остальных элементов
(map-comp #'+ '(1 2 3) '(4 6 9))	= (5 8 12) Суммы
(map-comp #'* '(1 2 3) '(4 6 9))	= (4 12 27) Произведения
(map-comp #'CONS '(1 2 3) '(4 6 9))	= ((1 . 4) (2 . 6) (3 . 9)) Пары
(map-comp #'EQ '(4 2 3) '(4 6 9))	= (T NIL NIL) Сравнения

Пример 4.2. Покомпонентные действия над векторами, представленными с помощью списков

Определение на Clisp	*Примечание*
(DEFUN mapf (fl el) (COND (fl (CONS (FUNCALL (CAR fl) el) (mapf (CDR fl) el)))))	 Пока FL не пуст, присоединяем результат очередной функции от EL к списку результатов остальных функций
(mapf '(length CAR CDR) '(a b c d))	= (4 a (b c d))

Пример 4.3. Применение списка функций к общему аргументу

Определение на Clisp	*Примечание*
(DEFUN map-f-e (fl el) (COND (fl (CONS (FUNCALL (CAR fl) (CAR el)) (map-f-e (CDR fl) (CDR el))))))	*Пока список функций не пуст,* присоединяем результат очередной функции от очередного элемента EL к списку результатов остальных функций
(map-f-e '(length CAR CDR) '((a b c d) (c d) (3 4 5)))	= (4 c (4 5))

Пример 4.4. Ряд функций над рядом списков

Определение на Clisp	*Примечание*
(defun map-ap (fn x) (cond (x (append (funcall fn (car x)) (map-ap fn (cdr x))))))	Пока список функций не пуст, цепляем результат очередной функции от очередного элемента EL к списку результатов остальных функций
(map-ap 'cdr '((1 2 3 4) (2 4 6 8)	= (2 3 4 4 6 8 6 9 12)

[111] #'x – эквивалент (FUNCTION x), что является представлением функции в качестве аргумента.

(3 6 9 12)))	

Пример 4.5. Сцепление результатов функции

Определение на Clisp	*Примечание*
(defun DECART (x y) (map-ap #'(lambda (i) (map-el #'(lambda (j) (list i j)) y)) x))	Однородный список пар, без лишних скобок
(DECART '(a s d) '(e r t))	= ((a e) (a r) (a t) (s e) (s r) (s t) (d e) (d r) (d t))

Пример 4.6. Декартово произведение

Определение на Clisp	*Примечание*
(defun conv-el (**fn** xl) (cond ((null xl) (**fn** Nil)) (xl (funcall **fn** (car xl) (conv-el **fn** (cdr xl))))))	Бинарная функция
(conv-el #'+ '(1 2 3 4))	= 10
(conv-el #'* '(1 2 3 4))	24

Пример 4.7. Свёртки

Определение на Clisp	*Примечание*
(defun DECART (x y) (map-ap #'(lambda (i) (map-el #'(lambda (j) (list i j)) y)) x))	Однородный список пар, без лишних скобок
(DECART '(a s d) '(e r t))	= ((a e) (a r) (a t) (s e) (s r) (s t) (d e) (d r) (d t))

Пример 4.8. Редукция

Композициями таких функционалов можно применять серии функций к списку общих аргументов или к параллельно заданной последовательности списков их аргументов. Естественно, и серии, и последовательности представляются списками.

Такие формулы удобны при моделировании множеств, графов и металингвистических формул, а к их обработке сводится широкий класс задач не только в информатике.

Показанные построения достаточно разнообразны, чтобы можно было сформулировать, в чем преимущества применения техники функционального программирования:

– отображающие функционалы позволяют строить программы из крупных действий;
– функционалы обеспечивают гибкость отображений;
– определение функции может совсем не зависеть от конкретных имен;
– с помощью функционалов можно управлять выбором формы результатов;
– параметром функционала может быть представление любой функции, преобразующей элементы структуры;
– функционалы позволяют формировать серии функций от общих данных;
– встроенные в Clisp функционалы приспособлены к покомпонентной обработке произвольного числа параметров;
– любую систему взаимосвязанных функций можно преобразовать к одной функции, используя вызовы безымянных функций.

4.5. Отложенные действия

Императивная организация вычислений по принципу немедленного и обязательного выполнения каждой очередной команды не всегда результативна. Существует много не императивных моделей управления процессами, позволяющих прерывать и откладывать процессы, а потом восстанавливать их и запускать или отменять. Организация такого управления, достаточного для оптимизации и программирования параллельных процессов, реализуется с помощью так называемых «замедленных»[112] или «ленивых» вычислений (lazy evaluation)[113]. Основная идея таких вычислений заключается в сведении

[112] Замедленные - значит они не выполняются сразу, немедленно. Синоним ленивых, задержанных и отложенных действий.

[113] Теперь включается во многие ЯП, например, Java не просто ООП, он мультипарадигмальный. Его авторы были под обаянием изученного в университете языка Scheme и сразу кое-что от него в свой язык включили. ООП не совпадает с

вызовов функций к представлению рецептов их вычисления, содержащих замыкания функций в определенном контексте:

(λ () fn) – заблокировать вычисление «fn», превратив его в тело функции без аргументов;

(fn) – разблокировать выражение «fn» в форме вызова функции без параметров.

Непосредственное применение таких формул влечёт многократное вычисление «fn», поэтому в инструментальном ядре используется реализационная структура данных, названная «рецепт», хранящая варианты представления выражения:

{ [F (fn . e)] | [T x]}

Сначала рецепт представлен как «(F (fn . e))», где «(fn . e)» – это замыкание выражения «fn» в пределах контекста «e». Попытка вычисления рецепта приводит к замене его результатом. По прежнему адресу размещается структура «(T x)», в которой «x» равен результату вычисления «fn» в контексте «e» (Табл. 4.4).

Таблица *4*.4

Дополнительные команды АМ для эффективной поддержки отложенных выражений

Команда	*Пояснение*
LDE	Загрузка отложенного выражения с созданием рецепта
RPL	Замена рецепта его результатом
APN	Возобновление отложенного выражения

Вычисляется такими командами рецепт не более чем один раз, и то если его результат действительно нужен. (Табл. 4.5)

Таблица *4*.5

Дополнение АМ для эффективного выполнения отложенных действий

Команда	*Результат*
s e (LDE f . c) d	→ ([F (f . e)] . s) e c d
(x) e (RPL) ([F (F . e)] ee c . d)	→ (x . s) ee c d при этом ([F (F . e)] → [T x] [114]
([T x] . s) e (APN . c) d	→ (x .s) e c d
([F (f . e)] . s) ee (APN . c) d	→ Nil e f ([F (F . e)] ee c . d)

Эффект отложенных вычислений можно продемонстрировать на обработке бесконечных структур данных, в которых продолжение структуры представлено как вычисление следующих элементов.

Фрагмент	*Примечание*
(def beg (LAMBDA (k lst) (cond ((EQ k Nil)()) (T (cons (car lst) (beg (cdr k) (eval (cons (cdr lst) Nil)))))))) (def endless (LAMBDA (m) (cons m (lambda () (endless (cons m m))))) (beg '(a b c) (endless 'A)) ; = (A (A . A) ((A . A) A . A))	Вырезка начального отрезка

Пример 4.9. Конструирование последовательности произвольной длины из бесконечного списка

императивностью, хотя в некоторой мере наследует её.

[114] Включение в квадратные скобки «[ ]» здесь символизирует размещение данного по тому же адресу.

4.6. Свойства атомов

Методы расширения функциональных построений могут быть применены для моделирования привычного императивно-процедурного стиля программирования и техники работы с глобальными определениями. Здесь демонстрируется расширение базовой схемы обработки символьных выражений и представленных с их помощью функциональных форм на примере механизма списков свойств атомов. В результате можно собирать функционально полное определение гибкой и расширяемой реализации ЯП, что и показано на примере Lisp-интерпретатора, написанном на языке Lisp.

До сих пор атом рассматривался только как уникальный указатель, обеспечивающий быстрое выяснение различимости имен, названий или символов. Теперь описываются списки свойств, которые начинаются или находятся в указанных ячейках.

Каждый атом имеет список свойств. Когда атом читается (вводится) впервые, тогда для него создается пустой список свойств, который потом можно заполнять. Список свойств устроен как специальная структура, подобная записям в языке Pascal, но указатели в такой записи сопровождаются тэгами, символизирующими тип хранимой информации. Первый элемент этой структуры расположен по адресу, который задан в указателе. Остальные элементы доступны по этому же указателю с помощью ряда специальных функций. Элементы структуры содержат различные свойства атома. Каждое свойство помечается атомом, называемым индикатором, или расположено в фиксированном поле структуры[115].

Здесь достаточно принять к сведению, что реализация атомарных объектов – это сложная структура данных, в свою очередь представленная списками.

С помощью функции GET в форме (GET x i) можно найти для атома x свойство, индикатор которого равен i.

Значением (GET 'FF 'EXPR) будет (LAMBDA (X) (COND ...)), если определение FF было предварительно задано с помощью (DEFUN FF (X)(COND ...)).

Свойство с его индикатором может быть вычеркнуто – удалено из списка функцией REMPROP в форме (REMPROP x i).

С середины 70-х годов возникла тенденция повышать эффективность разработкой специальных структур, отличающихся в разных реализациях. Существуют реализации, например, muLisp, допускающие работу с представлениями атома как с обычными списками посредством функций CAR, CDR.

4.7. Гибкий интерпретатор

В качестве примера повышения гибкости определений приведено упрощенное определение Lisp-интерпретатора на языке Lisp, полученное из М-выражения, приведенного Дж. Маккарти в описании Lisp 1.5.

Ради удобочитаемости здесь уменьшена диагностичность, нет пост-вычислений и формы PROG. Язык Lisp хорошо приспособлен к оптимизации программ. Любые совпадающие подвыражения можно локализовать и вынести за скобки, как можно заметить по передаче значения.

Определения функций хранятся в ассоциативном списке, как и значения переменных.

Функция SUBR – вызывает примитивы, реализованные другими, обычно низкоуровневыми, средствами.

ERROR – выдает сообщения об ошибках и сведения о контексте вычислений, способствующие поиску источника ошибки. Уточнена работа с функциональными аргументами.

Определение	*Примечание*
(DEFUN EVAL (e al) (COND ((EQ e NIL) NIL)	

[115] В Си и большинстве систем программирования память выделяется блоками разной длины, предназначенными для ранения векторов с доступом по индексу. В результате возникают ситуации, когда заранее неизвестно сколько элементов нужно или число элементов изменилось так, что заметная часть блока ничем полезным не занята. Конечно, можно заводить меньший блок, но так редко кто делает, при этом не всегда старый вектор освобождает память или программист должен явно объявлять освобождение памяти, что требует дополнительного внимания. Существует ряд не так уж простых методов уплотнения памяти в таких случаях, начиная с метода «близнецов», опубликованного в книге Кнута, и разные методы сжатия памяти, используемые в операционных системах при эксплуатации магнитных носителей. В системах программирования чаще всего используют модели со счётчиками, что требует дополнения в виде уплотнения памяти, иначе есть риск фрагментации памяти на мелкие кусочки так, что память вообще-то имеется, но нет непрерывных интервалов для удобного доступа к элементам вектора по индексу. В начале 1960-ых годов эти проблемы были математически проанализированы и был сделан вывод, что в таких случаях, если блок не полностью хранит данные, то часть памяти как бы не используется, неизбежно простаивает при довольно жёстком дефиците памяти. Бинарные деревья из небольших узлов свободны от таких простоев, хотя на первый взгляд иногда кажется, что происходит избыточная затрата памяти на хранения адресов таких узлов. Зато появляется возможность автоматического освобождения памяти, хотя здесь не сразу сложились достаточно удобные методы.

```
  ((ATOM e )((LAMBDA (v )                                  
       (COND (v (CDR v ) )                                 
         (T (ERROR 'undefvalue )) )          Диагностика   
    ) (ASSOC e al ) )                        Однократность вычисления
  )                                          значения      
  ((EQ (CAR e) 'QUOTE ) (CAR (CDR e )) )                   
  ((EQ (CAR e) 'FUNCTION )                                 
    (LIST 'CLOSURE (CADR fn ) al ) )         Замыкание     
  ((EQ (CAR e) 'COND ) (EVCON (CDR e ) al ) )  функционального
  (T (apply (CAR e)(evlis (CDR e) al ) al ) )  аргумента   
) )
```

Пример 4.5. Диагностика отсутствия определений при вычислении форм

Определение	*Примечание*
(DEFUN APPLY (fn args al)	
(COND	
((EQ e NIL) NIL)	
((ATOM fn)	
(COND	
((MEMBER fn '(CAR CDR CONS ATOM EQ))	Локализация списка встроенных подпрограмм
(SUBR fn agrs al))	
(T (APPLY (EVAL fn al) args al))	Выполнение подпрограмм
))	
((EQ (CAR fn) 'LABEL)	
(APPLY (CADDR fn)	
args	Именование локальных функций
(CONS (CONS (CADR fn)(CADDR fn))	
al)))	
((EQ (CAR fn) ' CLOSURE)	
(APPLY (CDR fn) args (CADDR fn)))	
((EQ (CAR fn) 'LAMBDA)	
(EVAL (CADDR fn)	Применение функционального аргумента
(APPEND (PAIR (CADR fn) args) al))	
(T (APPLY (EVAL fn al) args al))	
))	

Пример 4.6. Применение подпрограмм и функциональных параметров

Определения ASSOC, APPEND, PAIR, LIST – стандартны.

Определение	*Примечание*
(DEFUN evcon (c a) (COND ((null c) Nil) ((evel (car c) a) (evel (cadr c) a)) (T (evel (caddr c) a))))	Возможно отсутствие ветви

Пример 4.7. Выбор ветви

Определение	*Примечание*
(DEFUN evlis (m a) (COND (m (cons (evel (car m) a) (evlis (cdr m) a)))))	Пока «M» не пуст, присоединяем значение его первого элемента к списку значений остальных элементов

Пример 4.8. Вычисление параметров

4.8. Взаимодействие монад

Примерно то же самое обеспечивают EVAL-P и APPLY-P, рассчитанные на использование списков свойств атома для хранения постоянных значений и функциональных определений. Индикатор MONAD указывает в списке свойств атома на правило интерпретации функций, относящихся к отдельной монаде, MACRO – на частный метод построения представления функции. Функция VALUE реализует методы поиска текущего значения переменной в зависимости от контекста и свойств атомов.

Определение	*Примечание*
(DEFUN EVAL-P (E C) (COND ((ATOM E) (VALUE E C)) ((ATOM (CAR E)) (COND ((GET (CAR E) 'MONAD)	До применения

((GET (CAR E) ' MONAD) (CDR E) C)) (T (APPLY-P (CAR E) (EVLIS (CDR E) C) C)))) (T (APPLY-P (CAR E) (EVLIS (CDR E) C) C))))	функции выясняем её категорию

Пример 4.9. Функции разных категорий

Определение	*Примечание*
(DEFUN APPLY-P (F ARGS C) (COND ((ATOM F)(APPLY-P (FUNCTION F C) ARGS C)) ((ATOM (CAR F))(COND ((GET (CAR F) 'MACRO) (APPLY-P ((GET (CAR F) 'MACRO) (CDR F) C) ARGS C)) (T (APPLY-P (EVAL F E) ARGS C)))) (T (APPLY-P (EVAL F E) ARGS C))))	Для макрофункций нет необходимости в вычислении аргументов

Пример 4.10. Специальные макрофункции поддерживают пост-вычисления параметров

Или то же самое с вынесением общих подвыражений во вспомогательные параметры.

Определение	*Примечание*
(DEFUN EVAL-P (E C) (COND ((ATOM E) (VALUE E C)) ((ATOM (CAR E)) ((LAMBDA (V) (COND (V (V (CDR E) C)) (T (APPLY-P (CAR E)(EVLIS (CDR E) C) C)))) (GET (CAR E) ' MONAD))) (T (APPLY-P (CAR E)(EVLIS (CDR E) C) C))))	Однократный поиск категории

Пример 4.11. Исключение лишнего поиска свойства MONAD

Определение	*Примечание*
(DEFUN APPLY-P (F ARGS C) (COND ((ATOM F) (APPLY-P (FUNCTION F C) ARGS C)) ((ATOM (CAR F)) ((LAMBDA (V) (COND (V (APPLY-P (V (CDR F) C) ARGS C)) (T (APPLY-P (EVAL F E) ARGS C))))(GET (CAR F) 'MACRO))) (T (APPLY-P (EVAL F E) ARGS C))))	Однократный поиск свойства

Пример 4.12. Исключение лишнего поиска свойства MACRO

Расширение СП при таком определении интерпретации осуществляется простым введением/удалением соответствующих свойств атомов и функций.

Полученная схема интерпретации допускает разнообразные категории функций, реализуемые как отдельные монады, и макросредства конструирования функций, позволяет задавать различные механизмы передачи параметров функциям. Так, в языке Clisp различают кроме обычных, обязательных и позиционных ещё и необязательные (факультативные), ключевые и серийные (многократные, с переменным числом значений) параметры.

При разработке и отладке программ происходит развитие средств и методов обработки данных, что приводит к изменению типов представления объектов. Объекты из неопределенных становятся константами или переменными, из обработки скаляров формируется обработка структур данных, от структур данных вероятен переход к функциям и файлам и т. п.

Не менее существенное развитие происходит и на уровне постановки задачи, меняется степень её изученности. Задача из новой, решение которой имеет ранг пробы или демонстрации, становится направлением исследования, в котором созревают практичные или точные подзадачи, возможен и переход к поиску решений на пределе возможностей оборудования. Требования к средствам и методам решения задач на столь различных уровнях изученности привели к гибкости, присущей парадигме функционального программирования.

Определение	*Примечание*
(DEFUN eval(e a) (COND ((atom e) (cdr (assoc e a))) ((eq (car e) 'QUOTE) (cadr e)) ((eq(car e) 'COND) (evcon (cdr e) a))	

```
                    ( T         (apply (car e) (evlis (cdr e) a) a) ))
                                                                          Переменная
                                                                          Константа
(DEFUN apply (fn x a) (COND                                               Ветвление
                ((atom fn) (cond                                          Применение
                                ((eq fn 'CAR)    (caar x))                функции.
                                ((eq fn 'CDR)    (cdar x))
                           ((eq fn 'CONS)  (cons (car x) (cadr x)) )      Элементарные
                           ((eq fn 'ATOM) (atom (car x)) )                функции
                           ((eq fn 'EQ)       (eq (car x) (cadr x)) )
                           (T              (apply (eval fn a) x a)) ) )
                )
((eq (car fn) 'LAMBDA) (eval (caddr fn)
                             (pairlis (cadr fn) x a) ))
((eq (car fn) 'LABEL) (apply (caddr fn) x                                 Программируе
                      (cons (cons (cadr fn) (caddr fn)) a)))))            мые функции
                                                                          Безымянная
(DEFUN evcon (c a) (COND                                                  функция
                          ((eval (car  c) a) (eval (cadr c) a) )          Именованная
                          ( T  (eval (caddr c) a)     ) ))                функция

(DEFUN evlis (m a) (COND                                                  Выбор ветви
                          ((null m) Nil )
                          ( T       (cons (eval (car m) a)
                                          (evlis (cdr m) a)    ))
)
                                                                          Вычисление
                                                                          параметров
```

Пример 4.13. Само-описание Lisp-интерпретатора, предложенное Дж. Маккарти в начале 1960-х годов

Наиболее очевидные практические выводы из выбранных принципов ФП:
– процесс разработки программ разбивается на фазы: построение базиса и его пошаговое расширение;
– рассмотрение программы как реализации алгоритма естественно дополняется табличной реализацией функций, т. е. допустимо использование хранимых графиков функций в виде структур из аргументов и соответствующих результатов наряду с процедурами;
– прозрачность ссылок[116] обеспечена совпадением значений одинаково выглядящих формул, вычисляемых в одинаковом контексте;
– предпочтение универсальных функций и функционально полных систем, трудоемкость первичной реализации которых компенсируется надежностью определений и простотой применения.

Язык программирования Lisp и сложившееся на его основе ФП реально показали свои сильные стороны как инструментарий исследования и освоения новых областей применения вычислительной техники. Это аргумент в пользу функционального подхода к решению новых сложных задач:
– доступны преобразования программ и процессов;
– осуществима лингвистическая обработка информации;
– поддерживается гибкое управление базами данных;
– возможна оптимизация информационных систем, вплоть до полного исключения потерь информации;
– моделирование общения;
– и многое другое.

В настоящее время число новых ЯП стремительно возрастает, что требует специальной разработки методов лаконичного представления их особенностей для практичной оценки их возможностей. Предложенную визуализацию при оценке сложности и трудоёмкости программирования на базе ЯиСП можно дополнить разделением требований к постановкам задач по сферам применения на академические и производственные, а также, по уровню изученности на чёткие, развиваемые, усложнённо трудоёмкие и новые.

4.10. Абстрактная машина и специфика ФП

При сравнении императивного и функционального подходов к программированию, П. Лэндин (P.J. Landin) предложил специальную абстрактную машину secd (подробное описание можно найти в [145]) в качестве АМ для ЯФП. Машина secd работает над четырьмя независимыми регистрами: стек для результатов, контекст для именованных значений, управляющая программа, резервная память (Stack, Environment,

[116] Прозрачность ссылок означает, что формулу можно заменить на её значение или функционально эквивалентную формулу, рассчитывая, что поведение программы не изменится. График — один из вариантов функционального эквивалента.

Control list, Dump). Для характеристики встроенных команд интерпретатора и результата компиляции программ базового языка Lisp понадобятся следующие команды:

LD — ввод данного из контекста в стек;
LDC — ввод константы из программы в стек;
LDF — ввод определения функции в стек;
ATOM — неделимость (атомарность) верхнего элемента стека;
EQ — равенство двух верхних значений стека;
SEL — ветвление в зависимости от активного (верхнего) значения стека;
JOIN — переход к общей точке после ветвления;
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.
CAR — первый элемент из активного значения стека;
CDR — без первого элемента активное значение стека;
CONS — формирование узла по двум верхним значениям стека;
STOP — останов.

Следуя П. 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 ядра включает в себя структуро-разрушающие - деструктивные 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 4.19

Реализационное дополнение Pure Lisp

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

Таблица *4.20*

Парадигмальная характеристика языков, поддерживающих функциональное программирование

Параметр	*Конкретика*
Стартовая проблема	Исследование потенциального максимума возможностей решения сравнительно новой задачи.
Эксплуатационная прагматика ЯП	Применение символьных вычислений для расширения границ вычислимости.
Регистры абстрактной машины	S E C D S – стек операндов и чистого результата. E – стек значений локальных переменных C – стек исполняемой программы. D – стек защиты контекста от случайных искажений.
Категории команд абстрактной машины	Как в Pure Lisp.
Реализационная прагматика	Данные строятся из тэгированных указателей с автоматизацией повторного использования памяти. Система программирования использует пару интерпретатор-компилятор функций. При связывании переменных используются стек, ассоциативный список и список свойств атомов.
Парадигмальная специфика	Система программирования обычно поддерживает механизмы других парадигм, возможно выделенные в отдельные подсистемы (монады в языке Haskell).

Таблица 4.21

Паспорт ЯФП на примерах языка Pure Lisp

	ФП: Pure Lisp	*Примечание*

Типовая задача	Ознакомление с идеями и принципами функционального программирования.	*Выделен стартовый диалект, освобождённый от сложностей массового производственного программирования.*
Формат семантических систем	**< V, F, R >** — Функции могут быть значениями, а правила — функциями	*Накопительные эффекты при программировании способствуют созданию пакетов и диалектов.*
Особенности правила применения операций (R)	Функции разделены на две категории — обычные и конструкторы. Обычные применяются к вычисленным значениям аргументов, а конструкторы обрабатывают представления аргументов без их предварительного вычисления.	*Поддержана возможность программирования произвольных моделей вычислений, включая ленивые и параллельные.*
Приоритеты	ESMC Вычисления; структуры; имена; управление	*Порядок принятия решений* от вычислений к структурам данных, потом имена и минимальное управление.
AM	**secd** — Бесконечные регистры с защитой неизменяемых данных	*Архитектурная модель обладает гибкостью.*
AC	**AC = Lisp** Программа представлена как структура данных языка, допускающая обработку при выполнении	*Системная модель не требует дополнительных средств, достаточно самого ЯП.*
Диалекты	**BXDP** Базис; расширения; частичность; устройства	*Восходящая методика освоения, стартовый барьер через обработку атомов и списков с переходом к списочному представлению выражений и функций.*

Выводы:

1. Языки ФП нацелены на исследование полноты методов решения задачи, что обычно поддерживается удобными средствами отладки программ.
2. Возможно выделение средств чистого функционального программирования, смягчающих проблему отладки побочных эффектов на начальных этапах решения задач, что методически похоже на выделение структурного программирования из парадигмы ИП.
3. Расширение сферы применения ФП нередко достигается построением функциональных моделей других парадигм, что позволяет интегрировать разные подходы к решению задач и осваивать новые механизмы ИТ.
4. При необходимости недостаток эффективности вычислений на базе систем ФП может быть преодолён приведением функциональных программ к императивно-процедурной форме.
5. Результат ФП сводится к построению программы, декомпозированной на функции, допускающие верификацию ряда трудно удостоверяемых свойств, таких как безопасность, надежность живучесть и т. п.

Практика-4:

Отображения и редукции на ФП. СУ-анализ. Гомоморфизм и шаблоны. Переложение любых ПП в ФП

4.1. Напишите отображение списка атомов в список одно-элементных списков из них.

4.2. Из списка произвольных чисел постройте список натуральных чисел, делящихся на 3.

4.3. Выделите из мультипарадигмального языка его функциональную проекцию

4.4. По ранее написанной программе постройте её функциональный эквивалент.

ЯФП: Clisp, Clojure, Cmucl, Erlang, F# Haskell. Oz. Perl, Prolog Python (Rebol) Ruby Racket Scheme, Sisal Smalltalk, Snobol и др.

ЛЕКЦИЯ 5. ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

В предыдущей лекции при описании ФП отмечено, что его принципы допускают конкретизацию, удобную для определения параллельных вычислений в виде много-поточных программ. Семантические принципы представления функций — всеобщность (универсальность), само-применение, равноправие параметров и прагматические принципы организации вычислений — гибкость ограничений, неизменяемость данных и строгость результата, дополняются мета-программированием, гомоиконностью и анонимностью определений, а также приводят к моделям непрерывности процессов, обратимости действий и представлению унарных функций. На этом фоне параллелизм опирается на конкретизацию этого комплекса принципов и следствий, позволяющего рассматривать любой объем представленных данных и при необходимости переходить к многопроцессорным конфигурациям. Равноправие параметров гарантирует возможность представлять независимые потоки и объединять много-поточные или многопроцессорные программы в общий программный комплекс. Кроме того, в параллелизме полезны принципы строгого результата и универсальности, что позволяет при необходимости учитывать любое количество представленных потоков и реорганизовывать пространство потоков. Происходит конкретизация принципов для нужд параллелизма, прецеденты уже встречаются в конкретных ЯП. Такая конкретизация позволяет выполнять оттеснение маловероятных вычислений, балансировку нагрузки на одновременно работающие процессоры, формировать пространства параллельных итераций, а кроме того, привлекать автоматическое распараллеливание, поддерживать идентичность повторных прогонов для целей отладки программ, использовать много-контактные фрагменты с целью эффективной эксплуатации аппаратуры.

Прогресс производства аппаратуры намного опередил развитие технологий программирования. Параллельные архитектуры теперь позволяют принципиально повышать производительность программ решения многих задач. Методы автоматического распараллеливания программ способны обеспечить значительное ускорение вычислений, явно сводимых к комплексу независимых процессов обработки элементов векторов, но отступают перед программами более широкого класса. Перспектива выражения языковыми средствами параллелизма на уровне постановки новой задачи приторможена проблемами оптимизирующей верифицирующей компиляции, обеспечивающей аккуратный выбор эффективной и надежной схемы параллелизма.

Мультипарадигмальность долго живущих ЯП и тенденция XXI-го века по созданию новых мультипарадигмальных ЯП говорят о созревании отдельной ПП, объединяющей выверенные в практике механизмы параллельного программирования, особенно на уровне создания учебных языков и систем

программирования (ЯиСП), поддерживающих раннее обучение программированию как параллельному программированию, что и предсказывал Кеннет Айверсон, автор языка APL.

Ниша параллельного программирования обременена резким повышением сложности отладки программ[117], вызванным комбинаторикой выполнения фрагментов асинхронных процессов. Если первый барьер к успеху в параллельном программировании обусловлен последовательным стилем мышления, то второй барьер зависит от не преодолённой трудоёмкости отладки программ. Хотя основная трудоёмкость жизненного цикла программ связана именно с тестированием и отладкой программ[118], это направление более чем за полвека профессионального программирования прогрессирует преимущественно за счёт «силовых» характеристик оборудования, концептуально почти не выходя за пределы возможностей экстракодов отладки программ в доязыковую эпоху.

Кроме того, трудоёмкость повторного программирования и отладки параллельных алгоритмов обуславливает рассмотрение готовых последовательных программ в качестве эталонов при оценке результатов параллельных вычислений (ПВ), что сдерживает развитие языков параллельного программирования и методов их реализации[119].

5.1. Парадигма параллельных вычислений

В настоящее время осознана актуальность формирования парадигмы параллельного программирования, вызванная расширением и развитием системы базовых понятий, необходимых для рациональной разработки систем управления процессами на современной аппаратуре[120]. Парадигмы в программировании характеризуются стилем мышления при решении задачи, системой используемых понятий и особенностями их реализации. Можно констатировать, что стиль мышления и система понятий для организации параллельных вычислений (ПВ) уже сложились в процессе эволюции языков программирования (ЯП), но типовая их поддержка в системах программирования (СП) ещё не сформировалась

«Ничем не скроешь фундаментальную трудность параллелизма» – так прозвучало заявление Мартина Одерски (Martin Odersky) в его приглашенном докладе на 20-й Международной конференции «Конструирование компиляторов», состоявшейся в марте 2011 года в Германии под председательством Дженса Кнопа (Jens Knoop) [164]. Несмотря на многочисленные призывы и конкурсы[121], пока не появилось общих идей по новому поколению ЯиСП, в которых параллелизм имел бы самостоятельное значение, а не рассматривался бы (по понятным причинам) как пристройка к традиционному программированию[122].

5.2. Пространство решений

Рост интереса к параллельному программированию в наши дни связан с переходом к массовому производству многоядерных архитектур и активной эксплуатацией в сетях распределённых систем. Ознакомление с понятиями и явлениями параллельного программирования в данном материале нацелено на профилактику прочного привыкания к принципам традиционного последовательного программирования. Такая профилактика необходима для специализации в области разработки

---

[117] Отладка занимает примерно 75% времени подготовки программ. Не встречала, чтобы кто-то сравнивал последовательную и параллельную программу. Обычно применяют автоматическое распараллеливание последовательной программы, чтобы обойтись почти без отладки.

[118] Точного перечня методов отладки не существует, можно лишь описать подходы и решаемые при отладке задачи:
-Экспериментальная проверка соответствия программы, постановки решаемой задачи и тестового материала.
- При обнаружении несоответствия анализ источника несоответствия в вышеупомянутой тройке.
- Оценка срочности устранения ошибки.
- Подбор тестового материала, показывающего феномен неслучайной ошибки.
- Локализация причины ошибки в тесте программе или постановке задачи.
- Выбор метода устранения ошибки или, если это слишком накладно, то поиск путей её нейтрализации. (Например, подготовка инструкции, позволяющей пользователю на эту ошибку не нарываться)

[119] Теперь это поддержано в языке Sisal, ранее было в языке Барс. Кроме того, есть ряд диалектов APL, возникших со времён появления матричных процессоров.

[120] Парадигма параллельного программирования обычно позволяет повысить общность постановки задачи, что нередко способствует разработке алгоритма, допускающего более широкую область применения и обеспечить переносимость относительно разных архитектур.

[121] Эти призывы звучат уже более пятидесяти лет.

[122] В дополнение к указанной причине, хочется отметить ещё и развитие сетей целом, увеличение трафика, количества пакетов и далее по списку. Из-за этого в том числе активно развивается микросервисный подход в разработке приложений. А уже отсюда появляются различные механизмы, для взаимодействия этих микросервисов (помимо REST можно назвать RPC, gRPC), появились Kafka и RabbitMQ которые являются яркими представителями акторной модели вычислений. (Примечание Дмитрия Королева)

распределенных информационных систем, а также приложений для суперкомпьютеров, многопроцессорных конфигураций и графических процессоров.

Современное развитие ЯиСП на практике ориентировано на решение задач параллельного программирования. Как правило, новые ЯиСП включают в себя библиотечные модули, обеспечивающие организацию процессов, или подъязыки, допускающие много-поточное программирование. Это не исключает реальную практику ручного распараллеливания ранее отлаженных обычных программ, приведения их к виду, удобному для применения производственных систем поддержки ПВ. Значительная часть таких работ носит технический характер и заключается в систематической реорганизации структур данных, изменении статуса переменных и включении в программу аннотаций, сообщающих компилятору об информационно-логических взаимосвязях. Существенным ограничением результата ручного распараллеливания является не только опасность повторной отладки алгоритма, но и его жесткая зависимость от характеристик целевой архитектуры[123].

Разнообразие моделей ПВ и расширение спектра доступной архитектуры следует рассматривать как интеллектуальный вызов разработчикам ЯиСП, способным решать проблему создания методов компиляции много-поточных программ для многопроцессорных конфигураций. Язык должен допускать представление любых моделей параллелизма, проявляемого на уровне решаемой задачи или реализуемого с помощью реальной архитектуры, причем такое представление требует лаконичных форм и конструктивных построений, гарантирующих сохранение свойств программ при их реорганизации. Не менее важна расширяемость языка по мере развития средств и методов ПВ, темп которого превышает скорость осознания специалистами их возможностей.

Рассматривая задачу формализации языков параллельного программирования как путь к решению проблемы адаптации программ к различным особенностям используемых многопроцессорных комплексов и многоядерных процессоров, мы видим, что решение этой проблемы требует разработки новых методов компиляции программ, особенно масштабируемой кодогенерации, и развития средств и методов ясного описания семантики языков параллельного программирования.

5.2. Параллельные алгоритмы

Обзор решений, встречающихся при организации параллельных процессов для реализации параллельных алгоритмов, даёт некоторое представление о разнообразии многопроцессорных архитектур и систем:

– процессоров может быть много, и они могут обладать разными системами команд;
– процессоры могут работать автономно или сообща выполнять общую работу;
– отдельные процессоры могут функционировать вхолостую, не выполняя полезной работы;
– процессоры имеют свою локальную память и могут работать в общей памяти;
– распределение действий по процессорам может быть предписано программой, а может осуществляться супервизором, поддерживающим определенную модель вычислений.

Такое пространство допускает естественное развитие в направлении моделирования асинхронных процессов, а также помогает уяснить, что существуют контексты исполнения программ, от которых зависит выполнимость команд. При анализе результатов программ обнаруживаются разные категории ошибок управления и выделяются невыполнимые действия, несмотря на то, что все образующие их команды выполнимы. Проявляется роль частичного упорядочения действий в обеспечении работоспособности программ средствами управления вычислениями, отражающего не императивность последовательного исполнения действий при организации параллельных процессов. Оптимизирующие компиляторы любую программу нередко рассматривают как мультипрограмму.

Переход к параллельным алгоритмам влечёт пересмотр содержания многих понятий и введение новых терминов, отражающих разного рода явления и эффекты, не имевшие особого значения для обычных последовательных алгоритмов. Прошли времена, когда параллельное программирование называли «не алгоритмическим». Теперь термин «параллельный алгоритм» связан с пониманием того, что алгоритм может выполняться как одним, так и группой исполнителей[124]. Такое понимание резко расширяет пространство решений задач, меняет подходы к реализации решений, использующих параллелизм, и в

---

[123] Накладные расходы на параллелизм сильно отличаются в разных архитектурах. Есть и такие, где обмены быстрее арифметики. Конечно, традиция оценивать сложность по отношению к арифметике никуда не исчезла, хотя многие задачи не так уж от неё зависят. Накладные расходы больше связаны с отсутствием в популярных ЯВУ возможности определять и программировать дисциплины работы с памятью, устройствами и управления взаимодействиями. Компиляторы обычно работают в рамках RISC-архитектур, удобных для реализации ЯВУ, сложившихся до признания важности параллелизма.

[124] В теории рассматривают много-ленточные машины Тьюринга.

некоторой мере сказывается собственно на постановке задач и планировании полного жизненного цикла программ (ПЖЦП) решения задач, ориентированных на использование параллелизма.

Прежде всего, следует прояснить следующие вопросы, связанные с формулировкой постановки задачи:

1. Насколько изменится трудоёмкость ПЖЦП решения задачи с помощью параллельного алгоритма?
2. В какой мере при постановке задачи следует учитывать модель параллелизма?
3. Как обосновать и измерить выигрыш от разработки параллельного алгоритма?
4. Насколько изменяется постановка задачи при переходе к параллельным алгоритмам?
5. Что даёт парадигма параллельного программирования на уровне разработки параллельного алгоритма?
6. Какими средствами представляются разрабатываемые параллельные алгоритмы решения задачи на этапе, предшествующем разработке программы[125]?

Более чем за полвека традиционного последовательного программирования отлажено колоссальное количество программ, аккумулированных в системы программирования и стандартные библиотеки. Изменение постановок задач, уже имеющих готовые отлаженные программные решения, ради учёта допустимого параллелизма чревато повторным программированием и, что гораздо более трудоёмко, повторной отладкой.

Основной аргумент – целесообразность учета естественного параллелизма на уровне постановки задачи, утрачиваемого при решении задачи посредством обычных алгоритмов. Число языков параллельного программирования, удобных для реализации параллельных алгоритмов год от года растёт, хотя и их применение решает не все проблемы организации параллельных вычислений.

Итак, параллельный алгоритм может быть реализован по частям на множестве различных устройств с последующим объединением полученных результатов и получением целевого результата. Возникают практические вопросы:

- Каким образом в определении алгоритма выделены части, выполняемые отдельными устройствами?
- Обязана ли реализация алгоритма использовать в точности представленный в его определении набор устройств?
- Можно ли последовательный алгоритм рассматривать как параллельный, исполняемый на одном устройстве?

Следующая обойма вопросов касается категории «время» и связана с проблемами синхронизации:

- Могут ли части параллельного алгоритма обладать своим независимым или централизованным отсчетом времени?
- Может ли синхронизация частей алгоритма противоречить его информационным связям и логике управления?
- Можно ли синхронизацию частей алгоритма рассматривать как частный случай асинхронности?

Ещё большие сложности вызывают вопросы доступа к памяти:

- Каким образом взаимодействующие части параллельного алгоритма обмениваются данными?
- Могут ли части параллельного алгоритма изменять состояние общей памяти и памяти других частей?
- Может ли часть параллельного алгоритма воспрепятствовать использованию своей памяти другими частями?

Кроме того, части алгоритма могут быть определены в разных моделях вычислений и над разными структурами данных. Оценка результата разработки параллельного алгоритма, кроме оценки сложности

---

[125] В разных системах и теориях принимают во внимание разные способы: сообщения, сигналы, каналы передачи данных, буферы обмена данными, общая память, рандеву, «размазанные» переменные, императивная потактовая синхронизация, учёт временных интервалов срабатывания, может и что-то ещё, если припомню, то дам дополнение.

вычислений и объёма данных, осложнена целесообразностью оценивать выполнение разного рода трудно формализуемых критериев, часть которых, однако, поддаётся современным средствам верификации на моделях.

И всё же, сколь ни сложен мир параллелизма, программистам предстоит его понять и освоить!

Интересно отметить появление новых архитектур, обладающих полным набором команд с условным исполнением.

Исследования в области информатики в настоящее время претерпевают изменение системы понятий, используемой в практических ИТ. В этом процессе расширяется пространство решений сложных задач, модернизируются методы развития информационных систем (ИС) на основе компьютерных сетей и многопроцессорных комплексов.

Интересно рассмотреть перспективы развития парадигм программирования, обусловленных изменением условий эксплуатации современных информационных систем, особенно связанных с повсеместным распространением сетевых технологий, меняющих критерии оценки качества программ и методы обеспечения надежности и производительности программирования. Две основные линии такого развития – разработка распределенных информационных систем (РИС) и компонентное программирование (КП).

При практичном решении проблемы разработки языков параллельного программирования и подходов к их реализации в центре внимания вопросы выбора лаконичных форм представления программ, обеспечения конструктивных построений, гарантирующих сохранение правильности программ при их реорганизации, и поддержки расширяемости языка по мере развития средств и методов параллельного программирования.

Описание парадигмы параллельного программирования отражает прагматические различия в условиях реализации и применения изобразительных средств, используемых в ПЖЦП. Парадигмы параллельного программирования занимают нишу, связанную с реализацией программ выполнения вычислений на многопроцессорных системах для организации высокопроизводительных вычислений. Эта ниша обременена резким повышением трудоемкости отладки программ, вызванной комбинаторикой выполнения фрагментов асинхронных процессов.

Варьирование правил функционирования сетей допускает как асинхронную, так и синхронную организацию срабатывания действий, включая дозирование нагрузки и специализацию процессоров и распределение действий по потокам выполнения. Использование иерархических, многоуровневых, структурированных и расширяемых сетей обеспечивает моделирование практически любых, накопленных в языках программирования, техник программирования и представления структур данных.

5.3. Практичные системы программирования

При измерении производительности суперкомпьютеров и в экспериментах с распараллеливанием программ активно используются задачи научных расчётов, преимущественно реализующих алгоритмы векторной обработки данных, удобно распараллеливаемых с помощью MPI. Практические задачи современного параллельного программирования обычно выглядят как приведение больших отлаженных ранее программ на C или Fortran-е к форме, дающей выигрыш от распараллеливания с помощью штатных средств, включаемых в доступные системы программирования. В целом такая работа сводится к следующим видам работы:

– разметка участков программы, допускающих автоматическое распараллеливание;
– анализ участков и причин, препятствующих распараллеливанию программы;
– выбор участков программы, допускающих их техническое приведение к форме, пригодной для распараллеливания;
– изобретение рецептов полуручного преобразования текста программы с целью расширения возможностей распараллеливания;
– приведение текста программы к предельно распараллеливаемой форме;
– прогон распараллеленной версии программы для оценки выигрыша от параллелизма;
– частичное перепрограммирование и отладка фрагментов программы для исключения или смягчения эффектов, препятствующих достижению нужных характеристик производительности;
– установление частичной функциональной эквивалентности исходной программы и результирующей её версии.

Обычно компилятор поддерживает оптимизации, обеспечивающие устранение неиспользуемого кода, чистку циклов, слияние общих подвыражений, перенос участков повторяемости для обеспечения однородности распараллеливаемых ветвей, раскрутку или разбиение цикла, втягивание константных вычислений, уменьшение силы операций, удаление копий агрегатных конструкций и др.

Рассматривается зависимость ускорения вычислений от числа процессоров и объема общей и распределенной памяти. Выполняется систематическая замена рекурсии на циклы. Предпочитается однородное пространство процессоров, общая память, быстрые обмены, соседство, гарантирующие улучшение производительности систем для высокопроизводительных вычислений.

Заметно влияние дисциплины работы с памятью на характеристики параллельных процессов. Используется защищенная и размазанная память[126]. Различны решения, принятые в разных языках программирования, по работе с многоуровневой и разнородной памятью[127] (доступ, побочный эффект, реплики, дубли и копии). Обработка транзакций становится одной из типовых семантик работы с памятью в языках программирования.

Предлагаются средства и методы типизации управления процессами, удобными для подготовки программ, ориентированных на исполнение с помощью Open MP или MPI. Идеи языков программирования

---

126 - защищенная память — локальная память процесса, доступ к которой из других процессов запрещён.
- размазанная память — память процесса, допускающая доступ к её содержимому из других процессов так, что как только один процесс изменил её состояние, оно сразу доступно другим процессам без всяких дополнительных запросов или передач по каналам. Такая память продекларирована в Си-подобном языке mpC Ластовецкого, предложенном для многопроцессорного программирования.

127 Не считая внешней памяти, что уже разнородность, начиная с БИС и микропроцессорных архитектур память разделена на зоны с разной скоростью оперирования и разным объёмом. Может ещё и с разной дисциплиной доступа, особенно в многопроцессорных конфигурациях. Теперь во многих архитектурах, как правило, таких областей три: быстрая, оперативная и кэши разного объёма — чем больше объём, тем медленнее доступ. Разбирается с этим СП, на уровне ЯП на этот счёт нет явных средств. Встречаются спецификаторы как в Си, но они не более чем рекомендации компилятору.

по сетевому представлению типов управления и дисциплины работы с памятью ещё не получили удобной системной поддержки.

Компонентно-ориентированная разработка ПО может следовать единому сетевому определению семантики языков программирования и процесса разработки программ.

Некоторую поддержку работ по подготовке программ, использующих параллельные процессы, обеспечивает созданная фирмой Microsoft технология .NET, позволяющая применять общие системные библиотеки в программах на разных языках программирования.

Лидирующие в области поддержки параллелизма фирмы Intel и Microsoft предлагают разработчикам высокопроизводительных вычислений новые решения, направленные на преобразования программ в процессе их создания и распараллеливания:

– помощник по параллельному программированию (Parallel Advisor XE);
– компиляторы-отладчики и библиотеки (Parallel Composer XE2011);
– анализатор потоков и памяти (Parallel Inspector XE);
– профилировщик производительности (Parallel Amplifier XE).

Помощник по параллельному программированию (Parallel Advisor XE) анализирует исходный текст программы на языках Fortran или C[128] до ее компиляции и отмечает узкие места, препятствующие распараллеливанию.

Анализатор потоков и памяти исследует полученный в результате компиляции объектный код и выявляет неудачное распределение ресурсов, снижающее производительность программы.

Профилировщик производительности позволяет наблюдать экспериментальную проверку достигнутой производительности программы.

Технология применения всех этих инструментов предполагает, что программист по ходу дела многократно принимает решения об изменении исходного кода программы и вручную включает в него необходимые прагмы, указывающие компилятору допустимость оптимизирующих преобразований, что делает работу компилятора по распараллеливанию программы проще и надежнее. Возникает проблема автоматизации преобразования программ с целью приведения их к виду, удобному для распараллеливания компилятором и настройки объектного кода на конкретную многопроцессорную конфигурацию. Часть проблем решает схема подготовки параллельных программ с использованием библиотек преобразований исходного и объектного кода.

При демонстрации новых средств поддержки параллельного программирования представители фирм-разработчиков компиляторов показывают, насколько просто выполняется каждое такое преобразование в отдельной позиции программы. Проблемой является то, что таких позиций в нужной

[128] Интеловцы тогда на другие ЯП внимания не обращали. Да и теперь, хотя Python набирает популярность, общая картина меняется медленно.

программе может быть не десятки и сотни, а тысячи и десятки тысяч. Отыскать их все без специальных инструментов проблематично. Кроме того, при выполнении таких работ возникает необходимость повторной разработки структур данных и схем управления вычислениями, а также обоснованного выбора средств реализации программного конструктива.

Так, например, возникает необходимость устранения или преобразования ряда механизмов работы с глобальными переменными, общими блоками данных, глубокой вложенности процедур, перехода от рекурсивных вызовов к итерациям со статическим управлением циклом и другое. Кроме того, возникает целый спектр аналитических выкладок по установлению фактических областей видимости имён, выбор между общей памятью и памятью конкретного потока. Выполнение таких работ вручную связано с риском порождения типичных для параллельных программ ошибок, таких как гонки памяти, конфликт доступа, взаимоблокировка и т. п.

Реорганизация векторов для нужд эффективного распараллеливания может идти разными путями: разделение на чётные-нечётные элементы, разбиение на половины или четверти, распределение по частям, требующим разной интенсивности вычислений и т. п.

Иногда такие проблемы можно решить на уровне макропреобразований текста программы, выполняемых препроцессорами, – необходимо лишь разработать систему подходящих конкретной программе макроопределений[129]. Такие системы обычно не обладают универсальностью: они отражают индивидуальный стиль программирования, учитывают особенности мнемоники и неявные границы схем управления вычисления. В результате создаётся многопоточная версия программы, строго говоря, не являющаяся эквивалентом исходной программы, а лишь эквивалентно строящая её основные результаты.

Особый круг проблем связан с навыками учёта особенностей многоуровневой памяти в многопроцессорных системах. Обычное последовательное программирование такие проблемы может не замечать, полагаясь на решения компилятора, располагающего статической информацией о типах используемых данных и способного при необходимости выполнить оптимизирующие преобразования программы.

Следует отметить, что использование языков параллельного программирования в качестве языка представления исходной программы не гарантирует её приспособленность к удачному распараллеливанию.

---

[129] Макротехника не преодолевает недетеминизм, не заботится о равноправии вариантов. Здесь речь о преодолении неэффективности. Макротехника весьма разнообразна. Имеются прецеденты поддержки произвольного порядка макровызовов в зависимости от возможности их применить, причем не только слева направо, но и обратно и с возможностью повторного анализа текста с целью выявления вторичных вызовов, появившихся в результате макроподстановок до тех пор, пока таковых больше не окажется. Неэффективность макротехника преодолевает благодаря возможности открытых подстановок кода, поэтому она часто идёт в паре с ассемблером.

При анализе пригодности программы к распараллеливанию анализируются следующие потенциальные зоны риска:

– объявление и использование глобальных переменных;
– области видимости переменных и констант;
– определения распараллеливаемых процедур, содержащие внутренние вызовы других процедур;
– рекурсивные определения[130];
– заголовки циклов, управляющие кратностью выполнения распараллеливаемого тела цикла;
– ветвления и переключатели, дающие развилки потоков;
– обработчики исключений;
– реакции на события и сообщения;
– позиции возможного обмена данными между потоками;
– тиражирование данных в общей памяти и в файлах;
– списки параметров функций, возможно преобразуемые в данные в общей памяти;
– декомпозиция управления циклом и реорганизация соответствующих структур данных;
– границы участков изменения значений переменных;
– переменные для значений промежуточных вычислений;
– многократно выполняемые идентичные вычисления – избыточные;
– инкрементные переменные, динамику изменения которых желательно сохранять для пост-анализа;
– позиции диагностической аранжировки для пост-анализа;
– фактически не задействованные переменные и константные или не используемые вычисления;
параметризация зависимостей частей многопоточной программы от номера потока.

Вытекающие из такого анализа преобразования программы могут быть выполнены в стиле символьной обработки данных с помощью техники лексического и синтаксического анализа, регулярных выражений и макропреобразований. Чисто технические трудности связаны с разнообразием типов преобразований. Практики придерживаются традиционной методики отладки, т. е. разделяют процесс преобразований на шаги, на каждом из которых выполняется ровно один тип преобразований, после которого выполняется прогон программы на имеющихся тестовых данных с целью удостоверения сохранения частичной функциональной эквивалентности.

Таким образом, техника приведения программ к распараллеливаемой форме меняет стиль применения систем программирования, требует программистской работы на уровне грани между разбором программы и кодогенерацией, а также обустройства библиотек преобразований программы и её оптимизирующих преобразований.

Важно учесть изменение критериев применимости оптимизирующих преобразований. Если устранение избыточных вычислений в последовательной программе обычно оценивается как улучшение, то в параллельной программе оно может оказаться пессимизацией из-за появления лишних взаимосвязей между потоками и доступа к общей памяти[131].

Следует принять во внимание, что, хотя основная трудоёмкость жизненного цикла программ связана с тестированием и отладкой программ, именно это направление за полвека профессионального программирования практически не получило должного прогресса. Если первый барьер к успеху в параллельном программировании обусловлен последовательным стилем мышления, то второй очевидно зависит от до сих пор не преодолённой трудоёмкости отладки. Основные результаты здесь получены в

---

[130] Дело не в том, что при распараллеливание нужно избегать рекурсию, а в том что автоматические средства распараллеливания обычно с рекурсией не связываются, им сложно анализировать число витков рекурсии. Избегают и вложенных определений процедур. В лучшем случае они выдают совет, что та или иная конструкция неудобна для распараллеливания. В таких случаях для достаточно важных программ программист придумывает систему макросов для преобразования текста программы в более удобный для распараллеливания вид. Учитывая ненадёжность таких ручных поправок, возникает определённый объём дополнительной отладки программы, что и является риском, т. к. объём возникающей доотладки трудно прогнозировать. Конечно, рекурсии не следует избегать, а следует наладить с ней нормальную работу. По мнению разработчиков компиляторов, такая работа требует некоторых изменений на уровня ядра, что в свою очередь ещё более рисковано.

[131] – Вместо «избыточные» лучше было сказать «дублирующиеся».
– Многие параллельные алгоритмы используют дублирование фрагментов программы ради минимизации приостановок процессов и ожидания обменов данными между ними.
– Для многих сверхбыстрых устройств приостановка — слишком дорогостоящая операция.
– Обмен данными влечёт накладные расходы на синхронизацию или проблемы с общей памятью.
– Убирая из программы дубли, нередко приходится решать проблему загрузки процессора, чтобы избежать его простоя. Простой или холостой ход часто оказывается наиболее практичным, но эстетически неприятно, да и энергопотребление может волновать.

форме парадигм функционального, структурного и объектно-ориентированного программирования и технологий поддержки непрерывной работоспособности разрабатываемой программы, смягчающих сложность тестирования, развития и версифицирования программного продукта.

Отладочные средства современных визуальных оболочек систем программирования, как и в ранние времена последовательного программирования, предоставляют для отладки параллельных программ средства выполнения в пошаговом режиме, отслеживания значений переменных и установления точек приостановок, для работы с которыми программист должен заранее подготовить комплекты тестов для всех участков программы и контроля корректности значений внутренних переменных. Возможно указание условий, при которых отладчик произведёт приостановку программы и обеспечит воспроизводимость повторного прогона программы.

Специфику параллельного программирования отражает лишь возможность переключения потоков, обнаружения тупиков и потерянных сигналов, совместно используемых (разделяемых) данных, реентерабельных процедур, вызываемых повторно из другого потока до их завершения.

Для целей отладки выполняется компиляция специальной отладочной версии программы, что говорит о необходимости дополнительной отладки программы уже без отладочного сервиса. Впрочем, известны средства фоновой отладки, обеспечивающее слежение за отлаживаемой программой из независимого смежного процесса уровня ОС.

Лишь в последние годы в работах ИСП РАН наметилось более серьёзное отношение к проблемам отладки параллельных программ, что представлено в форме появления теорий для обработки информации, подверженной искажениям, и методики сравнительной отладки программ, в которой задействовано понятие эталонной программы и схемы распределённого отладчика.

5.4. Модели параллелизма в языках программирования

История языков программирования накопила целый ряд примеров лаконичных форм представления программ, начиная с умолчаний, неявных циклов и операторов ввода-вывода в языке Fortran. С точностью до реализационной прагматики, при разработке языков параллельного программирования можно унаследовать языковые конструкции и механизмы из привычных парадигм программирования и зарекомендовавших себя языков параллельного программирования. [132]

Прежде всего, это алгебраические механизмы распространения функций и операций относительно структур данных, предложенные в первом языке параллельного программирования APL. Дальнейшее упрощение изобразительных средств управления параллелизмом дает предложенный в языке Sisal подход к неявному распараллеливанию циклов на основе построения пространства итераций по пространству обрабатываемых данных.

Ещё острее становятся проблемы разработки, отладки, тестирования и верификации параллельных программ по сложности весьма превосходящие обычное программирование. Именно здесь сосредоточена исследовательская активность современного языкотворчества и системного программирования, поиск средств и методов интеграции удачного опыта параллельного программирования и успешного обучения методам параллельных вычислений. Акцент на анализ подходящих решений в области разработки переносимых программ, средств визуализации процессов, типизации управления и дисциплины доступа к памяти.

Практически в мире параллелизма все базовые понятия программирования претерпевают изменение или расширение (программа, ветвление, цикл, событие, память, результат). Появляется ряд специфических для параллельного программирования понятий (процессор, поток, ожидание, длительность, фильтр, барьер). Программы становятся многопоточными, циклы – параллельными, память обретает копии и реплики, события происходят одновременно в разных синхронизуемых процессах, вычисление результата может не означать завершение процесса. Такая ревизия понятий влияет не только на стиль программирования, но и изменяет характер компиляции на этапе генерации кода программы. Возрастает

---

[132] Конечно, Erlang и его акторную модель стоило упомянуть, тем более, что была поблизости компания, использующая его как производственный язык. Вроде ни в одну из перечисленных он не входит. Теперь появились и другие языки с такой моделью. В те годы считала это частью субъектно-ориентированного программирования.

роль техники использования многократно используемых компонент схемного уровня, соответствующего средствам типизации управления процессами.

Существуют версии ряда стандартных языков императивного программирования, приспособленные к выражению взаимодействия последовательных процессов в предположении, что в каждый момент времени существует лишь один процесс. При таком подходе в программе выделяются критические интервалы, учет которых полезен при распараллеливании программ. Многие традиционные языки программирования приспособлены к выражению локального параллелизма с помощью специальных расширений или библиотечных функций, обеспечивающих выделение участков с независимыми действиями, пригодными для распараллеливания компилятором.

Большие надежды связаны со строго функциональным подходом к спецификации параллельных программ и типов данных в языке с предпочтением так называемой «ленивой» схемы вычислений. Противопоставление достоинств и недостатков «ленивых» и «энергичных» методов вычислений отчасти смягчается концепцией «монад» в строго функциональном языке программирования Haskell. Особенности определения семантики языковых конструкций по-прежнему не отражают решение проблем обеспечения удобочитаемости программ и их отладки[133]. Табулирование сложных вычислений, называемое «мемоизация», становится популярным практичным инструментом снижения сложности вычислений.

В настоящее время существуют сотни функциональных языков программирования, ориентированных на разные классы задач параллельного программирования. Языки функционального программирования обогатились типовыми средствами практически всех известных подходов к представлению программ и организации вычислительного эксперимента и информационных процессов. Обеспечена организация параллельных процессов. Возможна визуализация данных и программ. Имеются средства стандартного и объектно-ориентированного программирования. Поддержано управление компиляцией и конструирование компиляторов. На сегодняшний день утратили актуальность опасения относительно ресурсно-эксплуатационных трудностей функционального программирования. Функциональное программирования вносит свой вклад техникой рекурсивных определений и отображений, параллелизмом обработки аргументов функций, ленивых вычислений, а также сведением понятия «условия» в ветвлениях к понятиям «страж» или «образец».

В семействе практичных языков функционального программирования заметное место занимают языки организации распределенных и параллельных вычислений. Практики с большой похвалой отзываются о языке функционального программирования *Erlang* фирмы Ericsson.

Fortran - сопрограммы

Средства представления параллельных вычислений доступны, начиная с первых языков высокого уровня. Языки Fortran II и Fortran IV были достаточно универсальны для представления программ организации параллельных процессов. Механизм сопрограмм, допускающий многовходовые (Entry) процедуры, позволял представлять программы взаимодействующих процессов и декомпозировать программу на управляющую и вычисляющую части, выглядящие как независимые компоненты программы, но это не привело к практике параллельного программирования и постепенно сопрограммы превратились модули, обеспечивающие представление иерархии функций подобно иерархии классов в ООП. Синтаксически средства параллельного программирования выглядят в современных Fortran-программах как разметка текста ключевыми словами и вызовы библиотечных функций.

Lisp – отложенные «ленивые» вычисления

Унификация представления данных и программ позволила пополнить передачу параметров по заранее вычисленному значению или указателю возможностью организовывать передачу представления выражений, вычисление которых можно выполнять по мере необходимости. Такая возможность привела к

---

[133] – Чаще всего упоминают неидентичность прогонов, связанную с тем, что при большом числе процессоров нет средств гарантировать, что ни один из них не обломается, причем, каждый раз разный.
– Встречаются эксперименты по ФП инструментальной поддержке протоколирования и организации повторных прогонов с учётом протокола. Сдерживает трудоёмкость таких экспериментов.

концепции «ленивых» вычислений и развитию методов динамической оптимизации процессов в отличие от статической оптимизации программ.

APL – векторные операции, просачивание

Более удачной оказалась идея поэлементной обработки однородных структур данных в языке APL, особенно с появлением векторной архитектуры Cray[134]. В 1962 году был предложен интересный механизм реализации многомерных векторов, приспособленный к расширению и распараллеливанию обработки данных. Сложные данные представляются как пара из последовательности скаляров и паспорта, согласно которому эта последовательность структурируется. Такое решение позволяет любое определение функции над скалярами автоматически распространять на произвольные структуры данных из однотипных скаляров. В настоящее время для большинства специализированных языков параллельного программирования типично, что сложные построения факторизуются с учетом особенностей структуры данных так, что выделяются несложные отображающие функции, "просачиваемые" по структуре данных с помощью функций более высокого порядка – функционалов. В результате можно независимо варьировать структуры данных, функционалы, методы сборки полного результата и набор отображаемых множеств. Целенаправленно выделяются конвейерные процессы, приспособленные к минимизации хранения промежуточных результатов. С 1980-ых годов эта идея наследуется многими языками, поддерживающими параллелизм и в наше время фактически является стандартной.

Algol-68 – критические участки и семафоры

К концу 1960-х годов сложилось значительное разнообразие теоретических моделей параллелизма, при исследовании которых проявилась проблема надежности параллельных вычислений, выразившаяся в неожиданном различии поведения императивной последовательности действий в зависимости от включаемых в нее фрагментов «независимых» процессов, нарушающих императивность. Для профилактики таких эффектов в семантику языка Algol-68 включается идея непрерывно исполняемых критических участков и представления их защиты в терминах семафоров.

Unix и JCL – управление процессами

Независимо идеи явного порождения процессов и организации их взаимодействия через каналы возникают в языках управления заданиями и процессами в операционных системах.

Setl – множества и кванторы

Другой подход к надежности программирования предложен в языке теоретико-множественного программирования Setl, ориентированном на активизацию интуиции грамотных математиков при разработке спецификаций программ в терминах преобразования множеств, естественно подразумевающих

[134] – Не так уж мало языков поддерживает подсчитывание выражений справа налево и отсутствия иерархии предшествования функции. Это просто диктат разработчика, так проще реализация. При необходимости обычно можно расставить скобки.

возможность параллельной обработки элементов множества, причем в реализационно независимом стиле. Наследование решений из универсальных языков сверх высокого уровня, таких как Setl, абстрагирование данных и процессов в которых приспособлено к гибкому и строгому структурированию, удобно для культивирования доказательных построений в практике параллельного программирования. В этом плане представляет особый интерес эксперимент по развитию теоретико-множественной семантики языка Setl, в котором весьма общее построение формул с кванторами над множествами погружено в обычную схему последовательного управления процессами. Реализация языка Setl характеризуется богатым полиморфизмом. Для представления множеств используется около двадцати разных структур данных, выбор которых осуществляется системой программирования в зависимости от динамики операций над множествами. В результате программируемые функции слабо зависят от реализационной структуры данных. В практике управления процессами используется понимание команд как позиций независимого порождения процессов. Такое понимание естественно согласуется с идеями теории множеств о независимости элементов множеств и может служить основой архитектуро независимой семантики языка программирования.

SQL – транзакционная память и нормальные формы

Много более заметная проблема параллельной обработки данных независимыми операторами обнаружилась в практике применения общих баз данных, приведшей к выделению языка запросов (SQL) и концепции транзакционной памяти, теперь рассматриваемой как перспективная основа семантики языков параллельного программирования.

Occam – взаимодействие «процесс-канал»

Середина 1970-х годов характеризуется кризисом технологии программирования, выход из которого тогда виделся в массовом переходе к параллельному программированию. Активные исследования разрешимых классов параллельных схем программ показали ряд неудобных, снижающих эффективность распараллеливания, конструкций, таких как ветвления. Э. Дейкстра опубликовал решение этой проблемы в форме защищенных команд, которая нашла свое место в определении языка Occam,предоставляющем для транспьютерного программирования модель взаимодействия CSP процесс-канал. В эти же годы популяризируются идеи структурного программирования, нацеленные на снижение сложности отладки программ, близкие идеям функционального программирования, которое теперь рассматривается как один из универсальных методов представления удобно распараллеливаемых программ.

Ada – «рандеву»

В проект языка Ada предпочли включить механизм «рандеву», сводящий представление взаимодействия процессов к рассредоточенному обмену сообщениями, подобному сигналам в оборудовании и модели CCS, что можно рассматривать как приаппаратное низкоуровневое средство, несколько диссонирующее с высоким уровнем языка.

БАРС – сетевое управление

В нашей стране разработаны языки БАРС и Поляр с разными концепциями сетевого управления процессами и представления дисциплины доступа к памяти. Программирование на уникальном по уровню средств управления процессами языке БАРС нацелено на обеспечение высокопроизводительных вычислений и организацию асинхронных параллельных процессов. При создании языка БАРС в качестве базового ЯВУ был привлечен популярный язык Pascal, в 1970-е годы перераставший из учебного в производственный язык системного программирования. При сохранении основных принципов семантики вычислений были существенно обобщены средства структуризации данных на основе понятия «мультимножество», приспособленного к именованию элементов структур данных и учета кратности их использования. Работа с именованной памятью (Name-oriented) дополнена возможностью задавать дисциплину доступа к элементам памяти. Идеи более ранних языков параллельного программирования были развиты и обогащены в языке БАРС в трех направлениях:

- в качестве базовой структуры данных были выбраны мультимножества (размеченные множества с кратностью элементов);
- описание элементов памяти сопровождается предписанием дисциплины доступа к памяти;
- средства управления асинхронными процессами включали механизм сетей Петри, координирующих работу независимо созданных функциональных фрагментов.

Процедуры в таком языке приспособлены к варьированию дисциплины доступа к данным и схемы управления процессами обработки данных. Сети Петри позволяют независимые описания процессов связывать в терминах разметки. Узлы с одинаковой разметкой срабатывают одновременно. Процесс обработки данных рассматривается как распределенная система, находящаяся под сетевым управлением. Узлы такой системы могут сработать в зависимости от условий готовности разной природы: доступность ресурсов, сигналы монитора, внутри сетевые отношения, иерархия сетей, правила функционирования разносортных подсетей. Вычисления, как и в языках APL и Sisal, распространяются со скаляров на сложные структуры. Радикальное продвижение в повышении уровня программирования, предложенное в языке БАРС, заключается в переносе механизма типизации данных на проблему типизации схем управления.

Sisal – однократные присваивания и пространства итерирования

Авторы строго типизированного функционального языка параллельного программирования Sisal создали интересный прецедент по расширению и уточнению системы понятий программирования для нужд представления и реализации масштабируемых параллельных вычислений. Программа в этом языке строится из участков с однократными присваиваниями, удобными для оптимизирующих преобразований, включая распараллеливание. В структуре цикла выделены позиции для формирования пространства параллельных итераций, фильтрации или сборки параллельно полученных результатов и обработки

потоков данных при развитой системе работы с векторами, включая методику распространения скалярных операций на структуры данных.

Произошедшее в конце 1970-х годов отвлечение внимания от кризиса технологии программирования на задачу освоения микропроцессоров не остановило поиск языковых решений для представления программ, обладающих параллелизмом. Появился математически правильный[135] функциональный язык параллельного программирования Sisal, позволяющий формировать пространства итераций для эффективного распараллеливания циклов компилятором. Название языка функционального программирования Sisal расшифровывается как «Streams and Iterations in a Single Assignment Language». Система вычислений в языке Sisal использует понятие «мультизначение», позволяющее подобно языку APL распространять скалярные действия на данные любой структуры, а их обработку осуществлять на многопроцессорных конфигурациях. Отображение мультизначения рассматривается как обработка его элементов на независимых процессорах. Результаты отображения могут повергнуться свертке или фильтрации. Sisal-программа представляет собой набор функций, допускающих частичное применение, т.е. вычисление при неполном наборе аргументов. В таком случае по исходному определению функции строятся его проекции, зависящие от остальных аргументов, что позволяет оперативно использовать эффекты смешанных вычислений и определять специальные оптимизации программ, связанные с разнообразием используемых конструкций и реализационных вариантов параллельных вычислений. Основное продвижение по технике программирования в языке Sisal – развитие структуры циклов для их реализации на параллельных процессорах. Введено понятие «пространство итераций» и предложена специальная конструкция для фильтрации мультизначений, получаемых при совмещенном исполнении итераций и участков повторяемости. Формирование мультизначения управляется представлением пространства итераций и учетом зависимостей между одноуровневыми итерациями. Работа с именованной памятью (Name-oriented) освобождена от проблемы побочных эффектов методом локализации участков с однократными присваиваниями – SSA-форм, что делает программы удобными для преобразований, оптимизации и компиляции, включая распараллеливание и масштабирование.

Oberon – обучение управлению процессами

1980-е годы знаменует переход к сетевой обработке данных и признанию потенциала ООП при организации информационных бизнес-процессов. Появляются Oberon, Eiffel, SmallTalk-80, C++, Erlang, Perl и другие языки, отчасти компенсирующие недостаток базовых средств и методов реализации массово используемых императивных языков программирования в новых условиях. В наши дни Вирт позиционирует язык Oberon как кандидат на включение в школьную программу информатики вместо языка Pascal.[136]

Python и Ruby – разработка распределённых систем

Общий прогресс в эксплуатационных характеристиках оборудования с 1990-х годов резко расширил возможности сборки информационных систем из готовых компонентов и сделал доступными свободно

---

135 - математически правильный язык опирается на семантические системы, обладающие полнотой операций, их абстрактная семантика не использует состояния изменяемой памяти, программы понимаются как выражения, при вычислении которых действуют точные правила разрешения умолчаний, которых должны придерживаться разработчики компиляторов. Если в некой позиции нет нужного данного, всегда известно, что надо вместо него подставить для продолжения счёта.

136 – Собственно параллелизма нет, но организация процессов не уровне UNIX/LINUX вроде в нём имеется. Для начала этого достаточно. Вирт хотел этот язык сделать языком начального обучения школьников, приезжал к нам агитируя за эту идею. В ИСИ есть группа, контактирующая с Н. Виртом, и вроде сделавшая свой ЯП.

распространяемые программные инструменты конструирования систем программирования, как правило, поддерживающие организацию параллельных процессов, если не собственно на уровне языка, то на уровне библиотечных компонент. Мультипарадигматические языки Python и Ruby показывают хорошие результаты в программировании сетевых процессов для многопроцессорных комплексов и привлекают большое число сторонников. Язык Python зарекомендовал себя как удобное средство разработки распределенных систем и сетевого программирования.

Haskell – «ленивые» вычисления и мемоизация

Кроме того, существуют сотни функциональных языков программирования, ориентированных на разные классы задач параллельного программирования. Языки функционального программирования обогатились типовыми средствами практически всех известных подходов к представлению программ и организации вычислительного эксперимента и информационных процессов. Обеспечена организация параллельных процессов. Возможна визуализация данных и программ. Имеются средства стандартного и объектно-ориентированного программирования. Поддержано управление компиляцией и конструирование компиляторов. Методы функционального проектирования и программирования обеспечивают технику представления и отладки функциональных моделей, спецификации и верификации программ, исследования их свойств и экспериментального сравнения моделируемых параллельных процессов с моделями и прототипами. Функциональный подход исторически является основой для исследования средств и методов программирования, прототипирования и декомпозиции программируемых систем и развития современных методов параллельного и многоязыкового программирования. Разработан чисто функциональный язык Haskell, предлагающий эффективную модель «ленивых» вычислений с мемоизацией промежуточных результатов по принципу полузабытых методов «математического динамического программирования».

Норма – параллельные вычисления

Система поддержки параллельного программирования для решения задач вычислительной математики.

MPI и Open MP – средства распараллеливания программ

Появляются популярные системы MPI и Open MP, обеспечивающие эффективность параллельного программирования в рамках языков Fortran и C. В MPI взаимодействие процессов обеспечивается через посылку сообщений между процессорами. Open MP предоставляет процессам возможность использовать разные виды памяти, включая быструю общую память, что при удачном ее распределении позволяет достигать высокой эффективности.

mpC – синхронизация семейств процессоров

В мире параллелизма все базовые понятия программирования претерпели изменение или расширение (программа, ветвление, цикл, событие, память, результат). Появился ряд специфических для параллельного программирования понятий (процессор, поток, ожидание, длительность, фильтр, барьер). Программы стали многопоточными, циклы – параллельными, память обретает копии и реплики, события происходят одновременно в разных синхронизуемых процессах, вычисление результата может не означать завершение процесса. Такая ревизия понятий влияет не только на стиль программирования, но и изменяет характер компиляции на этапе генерации кода программы. Возрастает роль техники использования многократно используемых компонент схемного уровня, соответствующего средствам типизации управления процессами. Так, например, мало известный[137] язык mpC предлагает более детальный учет механизмов взаимодействия параллельных процессов в терминах барьеров и специальных категорий переменных, обладающих особым, аппаратно реализуемым поведением.

[137] – Программирование — не исключение. ИСП РАН — лидер качественных программистских проектов в нашей стране, проводящий много интересных перспективных работ. Их прежний директор В.П. Иванников (https://www.ivannikov-ws.org/) был Президентом Ассоциации свободно распространяемого программного обеспечения. Но кроме качества продукта на свете существует реклама, экономика и социальная психология. Реклама позволяет что угодно вдолбить в голову так, что даже при несогласии с ней есть высокая вероятность, что при быстром принятии решений вспомнится именно разрекламированное. Экономика нацелена на выгоду, а основная конкуренция идёт за рынки низких цен, т. к. людей с малыми доходами неизмеримо больше. Яркий пример — китайский метод экономии расходов на производство отказом от контроля качества товара. Просто покупателям объявляют, что при любой неисправности им быстро заменят на другой, хотя он может оказаться столь же неисправным. Социальная психология диктует, что высокие цены гарантируют качество (знаменитый анекдот про цену галстука: «Как, ты его купил всего за 10 долларов? Ну и дурак! Вон затем углом их продают за 100!»), вековой авторитет фирмы вызывает доверие (фирма ИБМ после 10-летного простоя, вызванного недооценкой микропроцессоров, поднялась апеллируя именно к многолетнему авторитету вопреки очевидному их отставанию не только от Японии, но и Болгарии), массовое распространение доказывает полезность — эффекты моду и общепринятого. Все так делают — железный аргумент, похожий на страх оказаться первым на танцплощадке, легче быть вторым.

Java, C#, Scala, F# – библиотеки управления синхронизацией

В новых CG для языков Java, C#, Scala, F# и т.д. существенно повысилась результативность системных решений в области работы с памятью, компиляции, манипулирования комплектами функций и классами объектов, выделение которых по существу обусловлено результатами теоретических работ в области системного статического анализа, основу которых все в большей мере составляет функциональный подход[138]. Получили значительное развитие методы декомпозиции программ в рамках объектно-, субъектно- и аспектно-ориентированных подходов к определению систем программирования на базе многократно используемых компонент. Для современных областей программирования и проектирования характерна интеграция средств и методов из разных парадигм, что может привести к профессиональной консолидации программистского корпуса.

Изучаются средства организации параллельных процессов средствами языка функционального программирования F# с библиотеками .Net. На этом языке рассматривается возможность реорганизация программ, что поддержано в языке специальными средствами типа Quotation – доступ к внутреннему представлению программ в виде структуры данных. Анализируется вклад типизации данных в построение эффективных программ. Использование модифицируемой и защищенной памяти. Прикладные аспекты работы с информацией, отражающей специфику области приложения программ (measure).

C# дает возможность встраивать функциональные построения в контекст привычных императивных программ. Рассматривается стыковка программ на новых языках с производственными системами, разрабатываемыми на базе библиотек .Net. Возможность оперирования деревом разбора программы и ее исполнимым кодом. Вопросы защиты программ и данных. Управление дисциплиной доступа к памяти и представление запросов к базам данных.

CUDA – многопроцессорные видеоплаты

Появление технологии CUDA, объединившей в графических ускорителях достоинства этих ранее сложившихся подходов, выводит параллельное программирование в ранг массово доступных методов создания программных систем благодаря преодолению стоимостного барьера. Следует ожидать, что развитие парадигмы параллельного программирования приведет к улучшению средств поддержки полного жизненного цикла программ, включая активное использование методов верификации взаимодействия процессов, автоматизацию приведения обычных программ к эффективно распараллеливаемой форме, обеспечение мобильности параллельных программ относительно параллельных архитектур, а следовательно и к повышению эффективности и надежности программ, приспособленных к многократному использованию типизированных решений особо важных наукоемких задач. На повестке дня – разработка методов архитектурно независимой кодогенерации масштабируемых параллельных программ, легко настраиваемых на особенности используемых вычислительных комплексов. Так, например, новый язык программирования OpenCL обеспечивает параллелизм на уровне инструкций и на уровне данных для различных графических и центральных процессоров.

Несколько подробнее рассмотрим функциональный язык параллельного программирования *Sisal***.**

Название языка расшифровывается как «Streams and Iterations in a Single Assignment Language», сам он представляет собой дальнейшее развития языка *VAL*, известного в середине 70-х годов. Среди целей разработки языка *Sisal* следует отметить наиболее характерные, связанные с функциональным стилем программирования:

– создание универсального функционального языка;
– разработка техники оптимизации для высокоэффективных параллельных программ;
– достижение эффективности исполнения, сравнимой с императивными языками типа Fortran и C;
– внедрение функционального стиля программирования для больших научных программ.

Эти цели создателей языка *Sisal* подтверждают, что функциональные языки способствуют разработке корректных параллельных программ. Одна из причин заключается в том, что функциональные программы свободны от побочных эффектов и ошибок, зависящих от реального времени. Это существенно снижает сложность отладки. Результаты переносимы на разные архитектуры, операционные системы или инструментальное окружение. В отличие от императивных языков, функциональные языки уменьшают нагрузку на кодирование, в них проще анализировать информационные потоки и схемы управления. Легко создать функциональную программу, которая безусловно является параллельной, если ее можно писать, освободившись от большинства сложностей параллельного программирования,

---

[138] - Речь только о тенденции. Дерево всех возможных исполнений характерно и для функционального стиля, причём не только для параллельных вычислений. В императивном подходе сложнее проблемы с синхронизацией, хотя и функциональный подход их не решает, а просто уклоняется.

связанных с выражением частичных отношений порядка между отдельными операциями уровня аппаратуры. Пользователь языка *Sisal* получает возможность сконцентрироваться на конструировании алгоритмов и раз работке программ в терминах крупноблочных и регулярно организованных построений, опираясь на естественный параллелизм уровня постановки задачи.

Параллельное программирование на языке Sisal опирается на парадигму функционального программирования. Но замысел языка нацелен на создание конкуренции вечно живому языку Fortran и, кроме того, в качестве базового языка был использован язык VAL, в свою очередь многое унаследовавший от языка Pascal. От языка Fortran унаследован ряд идей по обработке и представлению векторов.

Система вычислений в языке Sisal использует понятие «мультизначение», позволяющее подобно языку APL распространять скалярные действия на данные любой структуры, а их обработку осуществлять на многопроцессорных конфигурациях. Работа с именованной памятью (Name-oriented) освобождена от проблем с побочными эффектами с помощью локализации участков с однократными присваиваниями – SSA-форм, что делает программы удобными для преобразований, оптимизации и компиляции, включая распараллеливание и масштабирование. Отображение значений при информационной обработке рассматривается как исполняемое на многопроцессорных конфигурациях. Результаты отображения могут повергнуться свертке или фильтрации. Формирование конфигурации управляется представлением пространства итераций и учетом зависимостей между одноуровневыми итерациями. Программа строится из участков с однократными присваиваниями, что упрощает технику оптимизационных и распараллеливающих оптимизаций.

Основное продвижение по технике программирования – развитие структуры циклов для их реализации на параллельных архитектурах. Введено понятие «пространство итераций» и предложена специальная конструкция для фильтрации мультизначений, получаемых при совмещенном исполнении итераций и участков повторяемости.

Основные виды команд:

– обработка потоков (очередь = стек = список);
– контроль однократности присваиваний (SSA-формы);
– дополнение цикла участком «returns» для оформления значения распараллеленного цикла;
– формирование пространство итераций;
– канальный обмен между итерациями;
– операции по упаковке, свёртке или фильтрации серийных значений.

Фрагмент	*Пояснение*

For
Approx := 1.0;
Sign := 1.0;
Denom := 1.0;
i := 1
 	
while i <= Cycles do Sign := -Sign;
Denom := Denom + 2.0;
Approx := Approx + Sign / Denom;
i := i + 1
 	
returns Approx * 4.0
end for
	% инициирование цикла
	
	% предусловие завершения цикла
	% однократные
	% присваивания
	% образуют
	% тело цикла
	
	% выбор и вычисление результата цикла

Пример 5.1. Вычисление числа π (пи)

Фрагмент	*Пояснение*
for i in [1..Cycles/2] do val := 1.0/real(4*i-3) – 1.0/real(4*i-1); returns sum(val) end for * 4.0	% пространство параллельно исполнимых итераций % тело цикла, для каждого i исполняется независимо % выбор и свертка результатов всех итераций цикла % вычисление результата выражения

Пример 5.2. Это выражение также вычисляет число π (пи). Это выражение вычисляет сумму всех вычисленных значений val и умножает результат на 4.0

Фрагмент	*Пояснение*

for i in [1..2] dot j in [3..4] do
returns product (i+j)
end for
	% для пар индексов [1,3] и [2,4]
	% произведение сумм
	% = 24

Пример 5.3. В for-выражениях операции dot и cross могут порождать пары индексов при формировании пространства итерирования

Фрагмент	*Пояснение*

for i in [1..2] cross j in [3..4] do
returns product (i+j)
end for
	% для пар [1,3], [1,4], [2,3] и [2,4]
	% произведение сумм
	% = 600

Пример 5.4. В for-выражениях операции dot и cross могут порождать пары индексов при формировании пространства итерирования

Фрагмент	*Пояснение*

for
I := 1
while I < S do
K := I;
I := old I + 2;
J := K + I;
returns product(I+J)
end for
	% значение из предыдущей итерации

Пример 5.5. Итеративное for-выражение с обменом данными между итерациями

Как это свойственно языкам ФП, *Sisal* язык математически правильный – функции отображают аргументы в результаты без побочных эффектов, и программа строится как выражение, вырабатывающее значение. Наиболее интересна форма параллельного цикла. Она выделяет три части: *for* - генератор пространства итераций*, do* - тело цикла и *returns* - формирователь возвращаемых значений.

Sisal-программа представляет собой набор функций, допускающих частичное применение, т. е. вычисление при неполном наборе аргументов. В таком случае по исходному определению функции строятся его проекции, зависящие от остальных аргументов, что позволяет оперативно использовать эффекты смешанных вычислений и определять специальные оптимизации программ, связанные с разнообразием используемых конструкций и реализационных вариантов параллельных вычислений.

Фрагмент	*Пояснение*
function Sum (N); result (+ (sqw (1 .. N)));	% Сумма квадратов

Пример 5.6. Программа вычисления суммы квадратов на языке Sisal

Для языков сверхвысокого уровня характерна яркая специфика, связанная с поиском новых средств и методов программирования. Такая специфика может стать основой новой парадигмы.

5.5. Абстрактный комплекс

Операционная семантика языка программирования обычно базируется на определении абстрактной машины, которая в случае многопроцессорных конфигураций естественно становится конструкцией из абстрактных процессоров – абстрактным комплексом (АК). В основном определение команд абстрактной машины наследует решения secd-машины, предложенные П. Лэндиным и описанные в книге П. Хэндерсона. Для языка параллельного программирования такой комплекс можно определить следующим образом:

АМ = <Регистры, [АП-1, … АП-n], СК>
АП-i = <Пам-i, СК-i>
Регистры = < стек результатов,
структура общего контекста,
синхросеть потоков/процессоров (активных и пассивных),
очередь барьеров>

СК – система команд, включающая в себя универсальные для всех процессоров команды, типичные для языков управления заданиями.

Для параллельного программирования необходим пересмотр ряда понятий, а именно переход от АМ к абстрактному комплексу (АК):

АК = <P, R, B, D> , где

P – вектор процессоров,
R – список внешних результатов вычислений,
B – таблица синхронизованных потоков, ждущих достижения барьера другими потоками,
D – набор пассивных потоков.

Элементы P – вектора процессоров содержат номер активного потока, текущий результат его выполнения и собственно список его предстоящих действий (фрагментов):

P = <i, s, c>, где
i – уникальный номер потока, выполняемого на процессоре P;
s – стек текущих результатов i-го потока;
c – список действий i-го потока.

Определена частичная функция T(i), задающая ограничение времени выполнения потока P(i). Общая система команд АК поддерживает выполнение следующих действий:

LOAD – загрузка произвольного пассивного потока.
F-LOAD – загрузка заданного пассивного потока.
BAR – статическая синхронизация потоков по барьерам.
IF – фильтр по заданному предикату.
WHEN – ожидание истинности предиката.
FORK – развилка.
JOIN – слияние ветвей.
SEND – динамическая синхронизация активных потоков в стиле «рандеву».
WAIT – ожидание сообщения.
MASSAGE – получение сообщения.
KILL – принудительное отключение потока с диагностическим сообщением.
STOP – плановое завершение работы потока с формированием внешнего результата.

Нормальное завершение много-поточной программы происходит при исчерпании регистров B и D и плановом завершении всех активных потоков. Кроме того, общее завершение работы происходит при невозможности завершить работу каких-либо потоков регистра B – взаимоблокировки. Для простоты учебной модели здесь не рассматривается учет разнообразия категорий систем команд отдельных процессоров и видов используемой памяти с различной дисциплиной функционирования. Ради удобочитаемости общие команды АК представлены в виде системы переходов:

[i s c, ...] R B D → [i' s' c', …] R' B' D'

Естественно, на уровне абстрактной машины поддержаны обычные бинарные операции над данными и дополнительные операции локального управления процессами, возникающие при реализации системы программирования. Определение синхронизации циклов и рекурсий может быть представлено в терминах индексируемых барьеров. Динамическая синхронизация в стиле ООП несложно выражается с помощью дополнительного регистра для реализации канала обмена сообщениями.

Наполнение много-поточной программы может развиваться независимо от схем управления вычислениями в отдельных потоках, а схемы можно реорганизовывать без дополнительной отладки наполнения. Они играют роль макетов или моделей программ и работают подобно макросам (открытая подстановка), но с контролем соответствия параметров объявленным синтаксическим типам фрагментов. В новых языках программирования, поддерживающих параллелизм, таких как императивный C# и функциональный F#, имеется возможность манипулировать структурами данных, представляющими внутренний код программы.

Выделение схемного уровня упрощает включение в схему разработки программ механизмов верификации (подобие модели или соответствие аксиомам), а на их основе возможна проверка программ

на правдоподобие, логический вывод свойств, выполнение индуктивных и дедуктивных построений. Кроме того, техника отладки программ обогащается возможностью привлечения протоколов ранее выполненных вычислений и приведения программ к нормальным формам, удобным для сведения к базовым/стандартным моделям параллельных вычислений.

На уровне первичного элементарного программирования мало заметна принципиальная разница между хранением данных в оперативной и внешней памяти. Особенности работы с внешней памятью много детальнее изучены в практике организации реляционных баз данных (БД) и применения языка запросов к базам данных SQL.

Характерной особенностью хранимых в БД данных является их соответствие некоторым физическим параметрам реальных объектов. Такие параметры относительно одного объекта можно представлять в виде строки таблицы. Существуют системы управления базами данных (СУБД), обеспечивающие создание, долговременной хранение, использование и уточнение таких таблиц.

В использовании конкретных таблиц может быть заинтересовано множество пользователей для разных сфер применения. Хранение таблиц может быть организовано на базе различных устройств и файловых систем и доступ к ним может реализовываться с помощью различных дисциплин коммуникации. Для обеспечения эффективности использования аппаратуры и необходимого для практичности пользовательского интерфейса разработаны определённые системные решения, что суммарно приводит к трём уровням проектирования архитектуры БД.

Уровни архитектуры:

– внутренний (организация хранения);
– внешний (логика использования);
– промежуточный (концепции системы).

Важным достижением реализационного похода в этом направлении является понятие нормальных форм (НФ), дающее разработчику БД чёткое руководство по прогнозу эффективности принятых им решений. Число таких форм немного превышает десяток[139]. Соответствие нормальной форме определяет выбор схемы реализации БД, компактность и структуру хранения данных во внешней памяти и скорость доступа к хранимым данным. Причём во многих случаях повышение компактности хранения сопровождается повышением скорости доступа, опровергая расхожее утверждение, что выигрыш в памяти влечёт потери в скорости и наоборот. Нормальные формы образуют упорядоченную последовательность – очередная последовательность наследует свойства предыдущей.

Выводы:

1. Языки параллельного программирования нацелены на решение сложных проблем, связанных с задачами организации параллельных вычислений и эксплуатации многопроцессорных комплексов.
2. Имеется тенденция обогащать средствами параллелизма разные ЯП.
3. Результат параллельного программирования должен давать многопоточную программу, допускающую выполнение на произвольной многопроцессорной конфигурации.
4. Тенденция не менять подходы к описанию постановок задач и методов их решения препятствует формированию парадигмы параллельного программирования, необходимость в которой созревает в сфере решения сложных задач и организации параллельных вычислений на многопроцессорных комплексах .
5. Масштабируемость моделей параллелизма допускает их использование в качестве реализационной прагматики, поддерживающей разные парадигмы программирования.
6. Понимание особенностей мира параллелизма требует специального обучения.

5.6. Ознакомление с параллелизмом

В лаборатории информационных систем ИСИ СО РАН разрабатывается учебный язык программирования СИНХРОН, предназначенный для ознакомления с явлениями параллелизма, метапрограммированием и особенностями много-поточного программирования. При создании языка учтен опыт применения языков Lisp, Робик, SETL, БАРС, Sisal, Haskell, mpC, Oz. Язык СИНХРОН ориентирован на формирование навыков профилактики неудачных взаимодействий процессов, учёта равноправия независимых потоков, использования укрупнённых воздействий на общую память, прогнозирования

---

[139] Если не ошибаюсь, их вроде 12. Наиболее известны первые четыре. Впрочем, идея нормальных форм утрачивает актуальность на фоне прогресса электроники и улучшения эксплуатационных характеристик. Даже Тьюринговская премия была вручена за новую концепцию БД, освобождающую разработчиков от продумывания нормальных форм и приведения к ним своих данных.
Нечаянно увидела, вдруг это интересно: https://habr.com/ru/post/157329/ . парадигмы и новый Язык программирования o42a Руслан Лопатин @lorus Программист Github

результатов автоматизированных преобразований программ и данных, включая оптимизацию и компиляцию программ.

ФП основано на семантических принципах, таких как всеобщность (универсальность), само-применение и равноправие параметров функций. Оно опирается на прагматические принципы реализации вычислений, такие как гибкость ограничений на размеры блоков памяти, неизменяемость обрабатываемых данных и строгость результата функции. мета-программирования, верификации и факторизации программ на автономные модули. Прагматические принципы поддерживают интуитивные модели непрерывности процессов, обратимости действий и унарных функций, служащих основой для прототипов на ранних стадиях жизненного цикла программ. Всё это оказалось благоприятной почвой для параллельного программирования.

Комплекс семантических и прагматических принципов обеспечивает поддержку подготовки программ при организации параллельных вычислений, сводимых к наборам независимых потоков. Любой фрагмент, выполнение которого маловероятно, может быть перемещён из представления функции в отложенное действие. Ленивые или опережающие вычисления дают возможность оперативно перераспределять нагрузку процессоров. Любое конечное множество может работать как пространство итераций для определенной на нем функции при автоматическом распараллеливании.

Высокопроизводительное программирование требует учета перспектив многократного использования и улучшения программ.. Эксперименты на супер-компьютерах показали, что системные решения могут вносить значительный вклад в производительность параллельных вычислений, такой вклад может превышать теоретические оценки. Квалификация разработчика программных систем обычно включает в себя умение изобретать решения задач и навыки ответственного улучшения готовых решений.

При переходе к параллельному программированию многие привычные понятия претерпевают изменения, начиная с понятия «данное». Данными являются уже не только представления значений и функций, но ещё и процессоров, и разнообразных устройств. Такое обобщение распространяет сферу успешного программирования на параллельные вычисления и распределённые системы.

Много-поточная программа допускает асинхронное выполнение потоков и действий, что означает в соответствии с принципом равноправия параметров функций независимость порядка вычисления выражений от порядка их вхождения в структуры данных, рассматриваемые как функции над этими выражениями. Для приобретения навыков учёта такой независимости в языке СИНХРОН предлагается в записи структур данных разнести смысл скобок и разделителей. Круглые скобки означают, что доступ к данным возможен последовательно, а квадратные символизируют произвольный доступ. Если разделителем в перечне данных является точка с запятой «;», то элементы перечня вычисляются последовательно, в порядке вхождения в запись выражения, а в случае запятой «,» - в произвольном порядке, независимо от порядка записи (см. таблица 5.1).

Таблица 5.1. Структуры данных

Структура	*Примечание*
'('Выр(';'Выр)…')'	Список, заполняемый последовательно вычисляемыми элементами, в порядке записи в программе
|'('Выр(','Выр)…')'	Список, заполняемый в порядке записи в программе асинхронно вычисляемыми элементами, возможно в другом порядке
|'['Выр (','Выр)…']'	Вектор, заполняемый в порядке записи в программе асинхронно вычисляемыми элементами, возможно в другом порядке
|'['Выр (';'Выр)…']'	Вектор, заполняемый последовательно вычисляемыми элементами, в порядке записи в программе

Часть проблем взаимодействия потоков алгоритмически не разрешима, поэтому в задачи ознакомления с параллелизмом входит научиться предвидеть такие опасности и отлаживать программы при обнаружении неудачных взаимодействий потоков. Выражение может использовать размещённые в общей памяти данные, но не изменяет их. Директива обрабатывает память и при благополучном исходе даёт результат подобно выражению.

От потоков к процессам. Компилятор много-поточной программы строит функционально эквивалентную ей многопроцессорную программу для размещения на многопроцессорном комплексе. Набор потоков при компиляции обычно преобразуется в набор процессов, каждый из которых выполняется одним отдельным процессором, но может быть иначе при решении проблем балансировки нагрузки в соответствии с принципом гибкости ограничений.

Контекст исполнения программы при переходе к процессам — общая память, данные из неё доступны всем процессам многопроцессорной программы. Общая память содержит представления глобальных, возможно изменяемых, переменных. Существует и локальная память, соответствующая процессам, созданным по определениям потоков, функций и циклов, подчинённая иерархии определений в программе. Обработка общей и локальной памяти на уровне много-поточной программы выглядит одинаково, что позволяет при отладке перемещать фрагменты в разные позиции программы. Возможно хранение в общей памяти определений и имён. Типы именуемых данных можно устанавливать по виду их значения или функции. Имеются распознающие функции-предикаты ATOM, NUMBER, LIST, ARRAY и др.

Процесс выполняется как непустой ряд команд процессора, допускающий наращивание при исполнении. Очередная команда может начинать выполнение строго после начала предыдущей команды, но не обязана дожидаться её завершения. Каждый процессор работает по шагам, соответствующим выполнению одной команды. После шага происходит переход к общему механизму управления многопроцессорной конфигурацией.

Компилятор по каждому потоку строит корректный ряд команд процесса, согласованных для выполнения действий потока. Завершение набора процессов может включать выполнение сверток для получения единого результата, что несколько влияет на их синхронизацию. Между двумя соседними командами процесса может быть выполнена команда другого процесса, что требует особого внимания при организации воздействий на общую память.

Абстрактный процессорный комплекс способен выполнять вычисления несколько более широкого класса, чем обычно задано семантикой языка программирования. Кроме собственно процессоров к выполнению программы можно привлекать дополнительные устройства, которые рассматриваются как специальные процессоры без определённой в языке системы команд, способные выполнять некоторые действия по запросам от других процессоров. Система команд локального процессора поддерживает обработку данных, их размещение и реорганизацию в своей локальной или общей памяти, и управление ходом выполнения процессов, включая взаимодействие процессоров и устройств, и резервирование данных для их защиты от случайных изменений, подобно средствам операционных систем. Программа использует общую память, данные из которой доступны отдельным процессорам, выполняющим программу. Шаги разных процессоров асинхронны, но могут происходить одновременно.

При спецификации команд процессора использовалась предложенная П.Лендиным машина secd, дополненная учётом особенностей наиболее известных виртуальных машин, включая JVM. secd работает над четырьмя регистрами: S - стек для промежуточных результатов, E - контекст для размещения именованных значений, C - управляющая вычислениями программа, D - резервная память (Stack, Environment, Control-list, Dump). Регистры приспособлены для хранения выражений в форме символов или списков. Состояние машины полностью определяется содержимым этих регистров. К регистрам системы команд машины secd добавлен регистр «M» (Memory), что даёт уточнённое обозначение многопроцессорного комплекса: m(secd)+ – абстрактный многопроцессорный комплекс над общей памятью. Абстрактный много процессорный комплекс работает над двумя моделями памяти: обычной, безопасной, локальной, хранящей неизменяемые данные, и общей, глобальной, допускающей изменения состояния памяти с возможностью восстановления данных механизмом транзакций.

Это обозначение символизирует, что M — общая память для всех процессоров, (secd)+ – что хотя бы одни процессор обязателен, общее число процессоров произвольно и не исключено изменение их числа в динамике. M – глобальная память, условно подчинена принципу неизменяемости данных, расширенному механизмом восстановления данных (см. 3.4). Она состоит из регистров произвольного доступа, хранящих счётчик числа потоков, использующих эти регистры, имя переменной, её текущее значение и протокол произошедших изменений, устроенный как вектор или список. Поддержано восстановление значений, оттеснённых присваиваниями.

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

Разница между secd и m(secd)+ сводится к операциям над данными в общей памяти. Регистр M является списком списков или стеков, каждый элемент которого начинается с имени глобальной переменной, вслед за которым расположено её текущее значение, а дальше следует протокол изменения значений, выглядящий как последовательность номеров процессов с каждым последующим установленным этим процессом значением. В определённом смысле состояние общей памяти можно рассматривать как непременное дополнение к формальному результату работы программы. Фактически исполняются команды по очереди, последовательно, начиная с первых в регистре управляющей программы, хотя можно считать, что они исполняются в произвольном порядке по мере готовности.

Такое определение может быть машинно-независимым и переносимым. Размер стека не ограничен. Каждая команда абстрактного многопроцессорного комплекса «знает» число используемых при ее работе

элементов стека S и их форматы, элементы она удаляет из стека S и вместо них размещает один выработанный результат для обычных команд или набор результатов для многопроцессорных команд, размещая число процессов в верхнем элементе стека. Всегда известно число текущих результатов в стеке S, которые можно явно свернуть в единственный результат специальной свёрткой, необходимость в которой выясняет компилятор и размещает свёртку в регистре C абстрактной машины. Свёртка может быть встроенной в язык или программируемой, позволяющей указанное число элементов в стеке свести в одно данное. Например: список, структура, сумма, произведение, макс, мин, последний и т. п. Свёртки обычно коммутативны, за редким исключением — первый или последний по времени вычислений или позиции вхождения в программу. Фильтр можно рассматривать как частный случай свёртки.

Суммарно комплект команд включает в себя действия, часть из которых определены в книге П. Хендерсона (LD, LDC, LDF, AP, RTN, RAP, DUM, SEL, JOIN, CONS, CAR, CDR, CONS, ATOM, EQ, SUB, ADD, MUL, DIV, STOP). Этот набор команд в языке СИНХРОН пополнен для решения проблем работы с общей памятью и внешними устройствами, особенности которых проявляются на командах воздействия на общую память, взаимодействия общей и локальной памяти, доступа к устройствам и организации параллельных процессов [17-20]:

LDM – копирование данного из общей памяти в стек;

SET – запись в общую память комплекса из стека;

LET-- сохранение локальных значений в общей памяти;

DEL – удаление верхнего элемента стека;

MLL – пересылка в головной элемент списка из другого списка;

MLV– пересылка из списка в указанный элемент вектора;

MVL– пересылка из вектора в головной элемент списка;

MVV – пересылка в указанный элемент вектора из другого вектора;

CHNG – обмен данными в общей памяти комплекса.

Восстановление данных. Для языка СИНХРОН уточнён принцип неизменяемости данных, характерный для ФП. Работа с памятью поддержана в транзакционном стиле, подобно обработке записей в базах данных, т. е. каждое воздействие на память или выполняется полностью, или восстанавливается состояние до начала выполнения не завершившейся команды [22]. Каждый элемент общей памяти в любой момент времени обрабатывается только в одном процессе программы, подобно захвату-освобождению файла. Хранение данных в общей памяти сопровождается протоколом изменений, чтобы при отладке можно было видеть какой процесс внёс изменения и, при необходимости, вернуть утраченное значение.

Возможен побочный эффект присваивания, допускающий при необходимости восстановление прежних значений переменных. Такая реализация позволяет поддержать транзакции и обратимость обработки памяти при отладке. В стеке размещается список результатов однократных присваиваний. К верхнему элементу контекста прицепляется этот список в хвост, вслед за аргументами. Предполагается, что имена формальных параметров и локальных данных различны.

Усложнённые команды пересылки и обмена данными в общей памяти (MLL, MLV, MVL, MVV, CHNG) нужны для профилактики возникновения временных интервалов между взаимосвязанными присваиваниями в общей памяти. Иначе может возникать так называемая «фантомная» память или доступ к формально уже удалённым данным, что иногда обнаруживается при использовании JVM в сетях, стандарт на которую был утверждён более 20-ти лет назад.

Внешние устройства и процессоры. При работе с устройствами через конечное время вырабатывается сигнал, говорящий или об успешном выполнении действия, или о причине отказа в его завершении. Выработка сигнала об отказе внешнего устройства приводит к запоминанию в протоколе информации об ущербно выработанных результатах, что можно учесть при отладке и выполнению остаточной программы в стиле смешанных вычислений.

WRT – вывод данных из стека на внешнее устройство с сигналом успеха-провала;

RD – ввод данных от внешнего устройства в стек с сигналом успеха-провала;

ANY – выравнивание стека, если данное с устройства не введено из-за провала.

Для организации параллельных вычислений требуются ещё команды:

KIT – комплекс из процессоров для выполнения действий;

ROW – ряд действий на одном процессоре;

REZ – размещение или использование заданного числа результатов процесса в свой стек;

WAIT – приостановка с ожиданием указанного процесса (завершения или сообщения);

SEND – сообщение указанному процессу;
NEXT – ожидание события или завершения действия указанного процесса.

Так обеспечивается подключение комплексов процессоров для неупорядоченных наборов потоков, мощность комплекса известна. На каждом процессоре происходит размещение процессов в виде ряда действий, длина ряда известна в каждый момент. Предполагается выполнение последовательности действий ряда на одном процессоре, завершаемых передачей результатов процесса в стек с локализацией воздействий на транзакционную общую память и явной обработкой ошибок.

Реализация передачи сообщений подобна рандеву в языке ADA – обмен происходит между двумя активными процессами и каждый знает что и кому он передаёт или от кого сообщение получает. Процесс-отправитель перед выполнением SEND может "дожидаться ожидания" получателя, то есть проверять, что получатель сейчас выполняет команду WAIT, иначе ожидать её, чтобы произошло "рандеву". Процесс может не знать, что его завершения кто-то ждет.

Многопроцессорная программа считается идеальной, компилятор её строит без учёта возможных аварийных ситуаций, считает, что все данные на устройстве ввода расположены в соответствии с запросами программы, именно те, что нужны. Состояния стеков к моменту выполнения команд сформированы вполне корректно. Регистры локальной памяти подчинены принципу неизменяемости данных. Новые данные размещаются в первом элементе списка, а к прежним значениям, хранимым в общей памяти, можно вернуться при необходимости, например, при отладке посмотреть историю изменения данных. При выполнении обратимых действий поддержана и обратимость воздействий на общую память. Идеальная конфигурация многопроцессорной программы работает на как бы бесконечной общей памяти, не решает проблемы её исчерпания и необходимости освобождения от ставших ненужными данных. Поскольку процессоры рассматриваются как одна из категорий данных, то можно от понимания памяти как одного из регистров машины перейти к отдельному процессору общей памяти, обладающему своей системой команд.

Общая память. Освобождение общей памяти требует механизмов, подобных опробованным в практике операционных систем при решении вопросов управления распределёнными системами с совмещением работы независимых программ. Можно рассмотреть определение многопроцессорного комплекса, допускающее управление доступом к общей памяти с использованием паспортов, хранящих шкалу необходимых регистров общей памяти для доступа к глобальным переменным. Паспорт одновременно доступен и локальному процессору, и процессору общей памяти. В этом случае локальный процесс при «сборке мусора» формирует новое состояние такой шкалы, доступной процессору общей памяти. Локальные процессы вместо произвольного доступа к общей памяти обладают лишь шкалой для доступа к определённым регистрам. При освобождении общей памяти появляется возможность частичного освобождения памяти, не задействованной в объединении шкал регистров от локальных процессов, без приостановки всех процессоров. В частности, при отказе отдельного процессора можно по шкале глобальных переменных выполнить уменьшение счётчиков доступа к этим переменным для решений по освобождению или восстановлению памяти.

Несколько проще выглядит модель комплекса со специальным процессором общей памяти, поддерживающим императивную синхронизацию. Такой процессорный комплекс состоит из ряда обычных процессоров и одного процессора общей памяти, с которым могут взаимодействовать локальные процессоры. Между собой они взаимодействуют только через общую память. Каждый процессор выполняет одну последовательность команд, среди которых встречаются команды запросов к процессору общей памяти. Это активные команды, выполняемые по ходу процесса без особенностей. Команды запроса к общей памяти имеют двойников среди команд процессора общей памяти. Это пассивные команды, работающие как ленивые вычисления, возбуждаемые при выполнении запросов от локальных процессоорв. Пассивные команды процессора общей памяти собраны в ряд очередей, каждая из которых соответствует локальному процессору и содержит двойников в том же порядке, что и последовательность запросов от активных команд. Пара запрос и его двойник выполняются неразрывно в стиле рандеву языка Ada. Как и при пересылках, механизм рандеву исключает случайное вмешательство сторонних процессов. При этом происходит обмен данными между локальной памятью активного процессора и общей памятью пассивного процессора (см. Таблица 5.2).

Таблица 5.2 . Дублеты - парные команды — срабатывают только неразрывно вместе

Активные команды локальных процессоров	Пассивные команды-двойники процессора общей памяти, ждущие запросов от локальных процессоров
GIVE – запрос регистра для переменной в общей памяти	TAKE – выбор регистра для переменной, счётчик кратности доступа к нему его надо увеличить на 1.
SAFE – запрос на хранение данного в переменной в общей памяти	WRITE – размещение данного. В протокол вносится указатель на данное и номер процесса.
READ – запрос на чтение данного	VAR – доступ процесса к данному из общей памяти
UNDO – запрос на получение предыдущего значения переменной	UNDO – доступ к предыдущему значению переменной, состояние регистров переменной не меняется.
FREE – объявление, что больше переменная не нужна.	FREE – освобождение памяти, счётчик кратности доступа уменьшается на 1.

Для локальных процессоров возможны запросы на регистр для переменной, на хранение данного в переменной по её регистру, на чтение данного из переменной по регистру, на получение предыдущего данного из переменной по адресу и на объявление, что больше переменная не нужна.

Команды-двойники процессора общей памяти, пассивно ждущего запросов от локальных процессоров или отладчика, одновременно с выполнебнием активных команд выполняют выбор регистра для переменной с указанным номером (счётчик кратности доступа к нему надо увеличить на 1), размещение данного (если данное — указатель на вектор или список, то они копируются полностью в общую память и в протокол вносится копия указателя на данное и номер активного процесса), чтение данного для передачи локальному процессору (состояние регистра не меняется), чтение предыдущего данного, освобождение памяти (счётчик кратности доступа уменьшается на 1).

Процессор общей памяти может функционировать как элемент распределённой системы, включающийся по мере поступления запросов от локальных процессоров, и не требующий приостановки всех процессоров на время решения проблемы освобождения памяти.

Представленный здесь метод использует ряд процессоров со своей локальной памятью и процессор общей памяти. Локальные процессоры могут работать автономно, по мере необходимости обращаясь к общей памяти и время от времени освобождая свою локальную память. В локальной памяти хранится шкала регистров доступа к общей памяти, которая может то расширяться, то сужаться по мере инструкций из программы или сужаться в результате «сборки мусора». Общая память состоит из двух частей — регистров прямого доступа и кучи для хранения протоколов, а также структур данных и старых значений после присваивания на случай необходимости восстановления состояний памяти. Время от времени при исчерпании той или иной области общей памяти или просто по графику запускается алгоритм, похожий на «Stop-copy» в системах ФП. Система команд абстрактного комплекса образует два уровня на каждом из которых можно видеть подсистемы организации вычислений, работы с памятью, управления процессами и структурирования данных. Один уровень реализует работу отдельного потока над локальной памятью, другой поддерживает взаимодействие потоков над общей памятью и внешними устройствами.

Таблица 5.*3*

Парадигмальная характеристика языков, поддерживающих параллельное программирование

Параметр	*Конкретика*
Стартовая проблема	Существуют особо важные задачи, производительность решения которых необходимо повысить.
Эксплуатационная прагматика ЯП	Поиск методов и технологий для повышения производительности ПВ переходом к многопроцессорным конгфигурациям
Регистры абстрактной машины	M(S E C D) M – общая память. S – стек операндов и чистого результата. E – стек значений локальных переменных C – стек исполняемой программы. D – стек защиты контекста от случайных искажений.

Категории команд абстрактной машины	Работа с устройствами и поддержка транзакций в общей памяти
Реализационная прагматика	Распараллеливание алгоритмов и программ или представление естественного параллелизма задач и процессов
Парадигмальная специфика	Системы программирования обычно поддерживают механизмы распараллеливания с учётом отдельно модели параллелизма.

Таблица 5.4

Паспорт ЯФП на примере функционального языка параллельного программирования Sisal

	ПВ: Sisal	*Примечание*
Типовая задача	Много-поточное программирование	*Выбор класса программ, представления которых удобны для распараллеливания.*
Формат семантических систем	**< V; F; R >** Функции не могут быть значениями, а правила не могут быть функциями	*Накопительные эффекты ограничены модулями и интерфейсами*
Особенности правила применения операций (R)	Выделение участков однократного присваивания (SSA-формы), распараллеливаемых по пространству итерирования циклов, просачивание операций по структурам данных.	*Принципы ФП гарантируют неизменяемость данных, удобство отладки программ и гибкость их представления.*
Приоритеты	**ESCM** **E S C M** Вычисления; структуры; управление; имена	*Порядок принятия решений от обычных вычислений к обработке триплетов и распараллеливаемым циклам.*
АМ	m**(secd)*** — Бесконечные регистры с защитой неизменяемых локальных данных и транзакциями в общей памяти	*Архитектурная модель дополнена общей памятью и расширена на многопоточность.*
АС	**АС:** граф разбора	*Типовая системная модель*
Диалекты	**BPXD** **B P X D** Базис; устройства; расширения; частичность	*Восходящая последовательность освоения, от элементарных вычислений переход к пространствам итерирования, функциям и проекциям*

Выводы

На этапе ознакомления с проблемами параллелизма можно строить и отлаживать небольшие много-поточные программы взаимодействия процессов, выполняемых на модели произвольной многопроцессорной конфигурации над общей памятью. При обучении параллельному программирования опыт такой работы поможет видеть и понимать разные явления, происходящие при взаимодействии потоков, предвидеть проблемы и накапливать рецепты решения проблем, возникающих при подготовке и отладке много-поточных программ, что особенно ярко может влиять на переход к суперкомпьютерам. Главный результат ознакомления — формирование интуитивных понятий, соответствующих реальной сложности параллельного программирования. Для этих целей предлагается принцип неизменяемости данных, характерный для ФП, распространить на работу с переменными в общей памяти и дополнить его механизмом восстановления данных, используя протоколы изменения значений переменных. Проблему освобождения общей памяти может быть решена как композиция методов «сборки мусора» и учёта кратности доступа к переменным из потоков, взаимодействующих через использование данных, хранящихся в общей памяти.

Практика-5:

Примеры много-поточных программ.

5.1. Представьте пространство итераций для суммы двух матриц.

5.2. Напишите много-поточную программу подсчёта числа нулей в строках матрицы.

5.3. Опишите модель параллелизма, поддержанную в конкретном языке программирования:

Algol-68 ADA APL, БАРС bash Сигма Clisp, Clojure, Component Pascal ECMAScript, MPI Норма

Eiffel, Erlang, F# Fortran Go Golang Java, JavaScript, JCL Kotlin mpC Ластовецкого Oberon Occam Python (Rest) (Rebol) Ruby Racket Scala, Scheme, Sisal Simula 67 Self SETL. Ассемблер «Эльбрус» Unix MIX и MMIX, Open MP Oz. PL/1, Planner,

ЛЕКЦИЯ 6. Императивно-процедурное программирование

В этой лекции рассмотрим первую и самую популярную парадигму программирования на уровне языков C и Pascal и сравним с её функциональной моделью Prog в языке Lisp.

Обычно императивно-процедурное программирование (ИП) рассматривает процесс обработки данных как конечную последовательность локальных изменений состояния общей памяти (императивно-процедурный стиль). Для ИП характерно четкое разделение понятий «программа» и «данные» с учётом статических методов контроля типов данных и оптимизации программ при компиляции. Это ограничивает семантику языка накопительными эффектами <A; F; R> уровнем программируемых процедур, без перехода границ элементов семантических систем. Общий механизм такой работы стандартной программы естественно представить как автомат с отдельными таблицами имен для переменных, значения которых подвержены изменениям, и для меток и процедур, определения которых неизменны.

При трансляции программ обычно планируется распределение памяти для значений переменных в зависимости от типов данных, включая размещение локальных данных в стеке (M). Выполняется частичный контроль доступа к переменным и совместимости операций и операндов по типам данных (C) с вычислением значений константных выражений (E - констант, переменных, результатов операций и вызовов функций), элементов структур данных (S), возможны оптимизирующие преобразования на уровне внутреннего языка для повышения эффективности управления вычислениями, что соответствует последовательности приоритетов MCES - :

СП = (Текст → {Код | Адрес}): Пам [Переменная] → Пам

6.1. Особенности представления программ на C

Самый популярный язык программирования C содержит низкоуровневое подмножество, что провоцирует рассматривать его как ЯНУ. Но приспособленность C к представлению обработки иерархии структур данных с помощью программируемых функций дает основания относить его к ЯВУ.

Язык C предполагает, что программа собирается из набора файлов, содержащих фрагменты программы и библиотеки функций, подготовленные, возможно, независимо — накопительный эффект. При этом компилируемая программа представляет собой одноуровневую конструкцию из равноправных определений структур данных и функций. Это означает, что любая функция имеет доступ к элементам любой структуры данных и может изменять их значения. Учитывая, что результат программы формируется как последовательность шагов изменения данных, размещённых по конкретным адресам, программист вынужден детально и тщательно изучать все тонкости побочных эффектов как своего, так и смежных фрагментов программы. Это оказалось серьёзным препятствием к повышению продуктивности труда.

–При конструировании кода программы C-компилятор распределяет память для глобальных и локальных данных в статической памяти или в стеке, соответственно.

– Встраиваются вызовы библиотечных функций создания-удаления для динамических структур данных и обмена данными, включая ввод-вывод.

– Данные бывают константами или переменными, идентифицируемыми с помощью идентификаторов.

– Типы данных разделены на основные и производные, созданные с помощью специальных операций, включая создание неоднородных структур данных.

– Над типами данных определён конкретный набор операций, с помощью которых строятся вычислимые выражения, и схем операторов управления и описания, используемых при конструировании функций.

– Многократно используемые функции объединяются в специализированные библиотеки, часть которых включена в системы программирования на языке С.
– Компилятору для эффективности кодирования нужна информация о типах данных переменных и результатов функций, но допустимо умолчание в случае типа int.
– Возможна спецификация вида используемой памяти, но она носит рекомендательный характер – компилятор учитывает её в меру возможности.
– Определения функций обладают некоторой свободой в конкретизации списка параметров на уровне вызова функции, что позволяет программировать функции произвольного числа параметров.
– Реализация арифметических операций обладает вариантами, зависящими от основного типа обрабатываемых скаляров в соответствии с разнообразием аппаратной поддержке этих операций.

Абстрактная машина (АМ) языка С может рассматриваться как развитие и обобщение АМ для ассемблера, обеспечивающее возможность использовать более сложные структуры данных в качестве регистров (см. Лекция 3). Определение мало отличается от АМ подмножества языка Pascal (см. Лекция 2):

s e c m → s’ e’ c’ m’

Сумматор расширяется до стека промежуточных значений, и появляется дополнительный регистр «Локалы» для локализации хранимых объектов при вызовах процедур:

<Стек_ значений, Локалы, Текущая_Команда, Память>

Регистр «Локалы» может быть устроен подобно регистру «E» из secd-машины Лендина, только хранит он не любые значения, а скаляры или ссылки на значения в «Памяти». Начальное состояние памяти – вектор глобальных переменных, адресуемых подпрограмм и меток. «Локалы» и «Память» образуют контекст исполнения программы.

АМ различает следующие категории команд:
– засылка значений из памяти в стек;
– пересылка значений из стека непосредственно в память или в регистр Локалы;
– организация переходов по метке в программе;
– организация ветвлений и циклов;
– вычисления над безымянными операндами в стеке при обработке выражений;
– организация вызовов функций с сохранением или восстановлением локального контекста.

Определение	*Примечание*

Main (argc, argv, envp)
 	
Int argc;
Char **argv;
Char **envp;
{
For (i=0; i < argc; i++)
Printf (”arg%i:%s\n”, i, argv [i]);
For (p=0; *p != (char*)0; p++)
Printf (”%s\n”, *p);
}
	Заголовок функции.
	Описания параметров функции:
	– число аргументов,
	– вектор аргументов,
	– вектор системных переменных.
	
	Цикл вывода аргументов командной строки.

Пример 6.1. Программа распечатки параметров командной строки и переменных среды на языке Си

6.2 Структурное программирование

Прагматика стандартного ИП не требует подробного описания – она общеизвестна и подробно исследована во многих работах. Тем не менее, следует отметить ряд моментов, связанных со структурным и функциональным программированием.

Сложность разработки больших программ, работающих в стиле локальных изменений состояния общей памяти, привела к идеям структурного программирования, налагающим на стиль представления программ ряд ограничений, способствующих повышению продуктивности отладки программ и приближающих технику ИП к ФП:

- дисциплина логики управления с избеганием переходов по меткам (goto_less_style);
- полнота условий в ветвлениях, отказ от отсутствия ветви “else”;
- минимизация использования глобальных переменных в пользу формальных параметров процедур (global_variable_harmful);
- однотипность результатов, полученных при прохождении через разные пути в программе.

Было выполнено большое число чисто теоретических работ, исследовавших соотношения между потенциалом императивного и функционального подходов и пришедших к заключению о формальной сводимости в обе стороны при некоторых непринципиальных ограничениях на технику программирования. Методика сведения императивных программ в функциональные заключается в определении правил разметки или переписывания схемы программы в функциональные формы. Переход от функциональных программ к императивным технически сложнее – используется интерпретация формул над некоторой специально устроенной абстрактной машиной. На практике переложение функциональных программ в императивные выполнить проще, чем наоборот – может не хватать близких понятий.

6.3.Функциональная модель ИП

С практической точки зрения любые конструкции стандартных ЯП могут быть введены как функции, дополняющие исходную СП, что делает их вполне легальными средствами в рамках функционального подхода. Необходимо лишь четко уяснить цену такого дополнения и преимущества практичного компромисса, который обычно связывается с наследованием решений и расширением сообщества пользователей. В первых реализациях языка Lisp были сразу предложены специальные формы и структуры данных, служащие мостом между разными стилями программирования, а заодно смягчающие недостатки исходной, слишком идеализированной, схемы интерпретации S-выражений, выстроенной для учебных и исследовательских целей. Важнейшее средство такого рода, выдержавшее испытание временем – prog-форма**,** списки свойств атома и деструктивные операции, расширяющие язык программирования так, что становятся возможными оптимизирующие преобразования структур данных, программ и процессов, а главное – раскрутка СП — их пошаговая разработка с выделение диалектов, соответствующих текущим задачам.

Prog-форма языка Lisp 1.5 может рассматриваться как абстрактная модель ИП, в которой при интерпретации используются отдельные ассоциативные таблицы для хранения параметров доступа к значениям рабочих переменных и определениям функций и процедур в локальной памяти. Это позволяет независимо задавать механизмы доступа к именам переменных и наименованиям процедур[140].

Применение возможностей prog-выражений позволяет писать «паскалеподобные» программы, состоящие из операторов, предназначенных для исполнения (точнее, «алголоподобные» (Algol-like), т. к. они появились за десять лет до Паскаля. Теперь более известен Pascal).

Для примера prog-выражения приводится императивное определение функции LENGTH, сканирующей список и вычисляющей число элементов на верхнем уровне списка.

---

[140] – prog-форма — это встроенная в язык Lisp модель привычных императивных вычислений, позволяющая отступать при необходимости от принципов ФП: изменять значения рабочих переменных, управлять ходом вычислений с помощью передач управления и меток, объявлять вырабатываемые результаты, строить частичные ветвления. Она допускает повышение эффективности вычислений, если реализована через машинные команды на уровне ядра языка, а не просто через функциональные конструкции.

Значение функции LENGTH – целое число. Программу можно примерно описать следующими словами (Стилизация примера от Маккарти.):

Русский язык	*Примечание*
Это функция одного аргумента L.	Объявление функции.
Она реализуется программой с двумя переменными u и v. Записать число 0 в v. Записать аргумент L в u.	Объявление рабочих переменных. Инициирование рабочих переменных.
A: Если u содержит NIL, то функция выполнена и её значением является то, что сейчас записано в v.	Меткой «A» помечена проверка завершения и формирования результата.
Записать в u cdr от того, что сейчас в u. Записать в v на единицу больше того, что сейчас записано в v.	Шаг продолжения функции.
Перейти к A"	Переход на метку «A».

Пример 6.2. Описание алгоритма на естественном языке

Эту программу можно написать в виде Pascal-программы. Строкам описанной выше программы в предположении, что существует библиотека функций над списками на Pascal-е, соответствуют строки определения функции:

Pascal	*Примечание*
function LENGTH (L: list) : integer;	Объявление функции LENGTH, выдающей целое число.

label A;
var U: list;
V: integer;
begin
V := 0;
U := L;
A: if null (U)
then LENGTH := V;
	Объявление метки
	и рабочих переменных.
	Инициирование рабочих переменных.
	Меткой «A» помечена проверка завершения
	и выработка результата
	функции.

U := cdr (U); V := V+1; goto A; end;	Шаг продолжения функции с изменением значений рабочих переменных. Переход на метку «A».

Пример 6.3.. Представление программы на языке Pascal

Переписывая это определение в виде лисповского S-выражения, получаем программу:

Lisp	*Примечание*
defun LENGTH (L) (prog (U V) (setq V 0) (setq U L) A (cond ((null U) (return V))) (setq U (cdr U)) (setq V (+ 1 V)) (go A)))	Объявление функции LENGTH, реализуемой в императивном стиле с двумя рабочими переменными. Инициирование рабочих переменных. Меткой «A» помечена проверка завершения и выработка результата функции. Шаг продолжения функции с изменением значений рабочих переменных.. Переход на метку «A».
(LENGTH '(A B C D)) (LENGTH '((X . Y) A CAR (N B) (X Y Z)))	Применение функции даёт 4. Применение функции даёт 5.

Пример 6.4.. Представление программы на языке Lisp

Отметим сразу гомоиконность определений в примерах 6.2-6.4.

Последние две формы в примере 6.4 представляют применение функции. Их значения - четыре и пять, соответственно. Prog-форма имеет структуру, синтаксически подобную определениям функций и процедур в языке Pascal: (PROG, список рабочих переменных, последовательность операторов и атомов ...). Атом «A» в списке выполняет роль метки, локализующей оператор, расположенный вслед за ним. Метка A, как и в примерах 2 и 3, локализует оператор, начинающийся с ветвления.

Первый список после символа PROG называется списком рабочих переменных. При отсутствии таковых должно быть написано NIL или (). С рабочими переменными обращаются примерно как со связанными переменными, но они не могут быть связаны ни с какими значениями через LAMBDA. Значение каждой рабочей переменной есть NIL до тех пор, пока ей не будет присвоено что-нибудь другое.

Для присваивания рабочей переменной применяется форма SET. Чтобы присвоить переменной pi значение 3.14 пишется (SET (QUOTE PI)3.14). SETQ подобна SE*T*, но она еще и блокирует вычисление первого аргумента. Поэтому (SETQ PI 3.14) – запись того же присваивания. SETQ обычно удобнее. SET и SETQ могут изменять значения любых рабочих переменных из более внешних функций.[141] Значением SET и SETQ является значение их второго аргумента.

Обычно в Prog-форме действия выполняются последовательно[142]. Выполнение действия, в отличие от функции, понимается как его вычисление и отбрасывание его значения. Действие программы обычно выполняются в большей степени ради эффекта действия, чем ради вычисленного значения.

GO-форма, используемая для указания перехода (GO A) указывает, что программа продолжается действием, помеченным атомом A, причем это A может быть и из внешнего выражения PROG.

Условные выражения в качестве действий программы являются базовым средством представления ветвлений. Если ни одно из пропозициональных выражений не истинно, то программа продолжается действием, следующим за условным выражением.[143]

RETURN – нормальное завершение программы. Аргумент RETURN вычисляется, что и является значением программы. Никакие последующие действия не вычисляются. Prog-форма изменяет только рабочие переменные, не создаёт побочных эффектов в общей памяти.

Prog-выражение может быть рекурсивным.

Функция REV, обращающая список и все подсписки, столь же естественно пишется с помощью рекурсивного Prog-выражения.

Pascal	*Примечание*
function REV (x: list) :list;	Определение функции REV, вырабатывающей список.
label A, B;	Объявлены метки
var y, z: list;	и локальные переменные.
begin	
A: if null (x) then REV := y;	Меткой «A» помечена проверка завершения и результат.
z := car (x);	Выбор очередного элемента для реверсирования.
if atom (z) then goto B;	Для атома переход на метку «B»

[141] Clisp – действуют на любом уровне ради введения статической области видимости.

[142] На этой странице речь идёт о программе, представленной в виде prog-формы. Строго говоря в семантике Лиспа нигде не утверждается, что вычисления происходят всегда последовательно. Если вычисление выражений изнутри наружу можно считать последовательным, то вычисление параметров слева направо является не более чем прагматической рационализацией, согласно принципам ФП их можно вычислять в произвольном порядке, что уже некоторая форма параллелизма. Более чётко это делается введением специальных функций, манипулирующих потоками, типа multiply в Clisp.

[143] В Clisp все ветвления работают так.

	в обход реверсирования.
z := REV (z);	Замена элемента на его реверс.
B: y := cons (z, y);	Меткой «B» помечена сборка результата.
x := cdr (x);	Шаг продвижения по списку.
goto A;	Переход на метку «A» для дальнейшего
end;	реверсирования.

Пример 6.5.. Представление программы на языке Pascal. Функция rev обращает все уровни списка так, что rev от (A ((B C) D)) даст ((D (C B))A).

Lisp	*Примечание*
(defun REV (X)	Определение функции REV,
(prog (Y Z)	реализуемая императивно с переменными.
A (cond ((null X)(return Y)))	«A» помечает завершение и дан результат.
(setq Z (car X))	Выбор очередного элемента для
	реверсирования.
(cond ((atom Z)(goto B)))	Для атома переход на метку «B»
	в обход реверсирования.
(setq Z (rev Z))	Замена элемента на его реверс.
B (setq Y (cons Z Y))	«B» помечает сборка реверсируемого списка.
(setq X (cdr X))	Шаг продвижения по списку.
(go A)	Переход на метку «A» для дальнейшего
))	реверсирования.

Пример 6.6.. Представление программы на языке Lisp

В принципе, SET и SETQ могут быть реализованы с помощью ассоциативного списка примерно так же, как и поиск значения, только с сохранением связей, расположенных ранее изменяемой переменной:

(defun SET (X Y) (cons (cons X Y) Alist)))

Введенное таким образом присваивание обеспечивает вычислимость левой части присваивания, т. е. можно в программе вычислять имена переменных, значение которых предстоит поменять.

4.4. Спецификация

Таблица 6.1

Парадигматическая характеристика языков, поддерживающих процедурно-императивную парадигму программирования

Параметр	*Конкретика*
Стартовая проблема	Представление программ эффективной реализации вычислительных алгоритмов
Эксплуатационная прагматика ЯП	Программирование решений задач, обладающих точной постановкой задачи и практичным алгоритмом без претензий на тщательное исследование.
Регистры абстрактной машины	S E C M S – стек операндов и промежуточных результатов. E – стек локальных переменных и аргументов. C – поток программы. M – вектор общей памяти.
Категории команд абстрактной машины	Засылка в стек. Вычисления над стеком. Пересылки в глобальную и локальную память. Передачи управления. Ветвления. Вызов процедуры. Возврат из процедуры.
Реализационная прагматика	Предпочтение статического распределения памяти по блокам для векторов заданного размера, контроля типов данных при выполнении операций и вызове процедур, операций над машинными словами.

	Программируемое освобождение памяти. Стеки реализованы как векторы.
Парадигматическая специфика	Процедурно-императивный стиль программирования обычно дополняют простыми средствами ФП.

Таблица 6.2

Паспорт языка ИП на примере языка Pascal

	ИП: Pascal	*Примечание*
Типовая задача	Поддержка обучения программированию на уровне университетов	*Выбран концептуальный минимум, удобный для подготовки и проверки программ решения учебных задач, имеющих алгоритмы решения.*
Формат семантических систем	**< V; F; R >** — Функции не могут быть значениями, а правила не могут быть функциями	*Накопительные эффекты ограничены библиотеками процедур.*
Особенности правила применения операций (R)	Различаются функции и процедуры по синтаксическим позициям их вызова. Процедуры являются обобщением оператора как элемента логики управления, а функции — входящей в выражение операции.	*Многие учебные задачи можно ставить как организацию форматного вывода данных.*
Приоритеты	**MCES** **M C E S** Переменные; логика управления; вычисления; структуры;	*Порядок принятия решений начинается с распределения памяти и графа передач управления.*
AM	**secm** — стек результатов, таблица определения, программа и общая память	*Архитектурная модель замещает дамп общей памятью.*
AC	**AC:** синтаксический граф разбора	*Типовая системная модель*
Диалекты	**PXDB** **P X D B** устройства; расширения; частичность; базис	*Нисходящая методика освоения от устройств ввода-вывода и библиотек к средствам языка*

Выводы:

1. Парадигмальная специфика языков, поддерживающих ИП, отражает нацеленность на широкую сферу их применения при заметном внимании эффективности программ по легко демонстрируемым или измеримым характеристикам без учёта проблем полноты возможностей организации вычислений и доступа к аппаратуре.
2. В языках, поддерживающих ИП, возможно выделение подмножеств, таких как средства структурного программирования, применение которых гарантирует на начальных этапах решения задач повышение темпа отладки и надёжности программирования..
3. Существование функциональных моделей ИП показывает возможность использования их преимуществ в рамках других парадигм. Результат ИП заключается в получении многократно используемого исполнимого кода программы, эффективность которого оправдывает трудоёмкость отладки и расходы на оптимизирующую компиляцию..

Практика-6

Обработка СД с разносортными элементами

6.1. Сделайте процедуру, строящую вектор из произвольных чисел разного типа.

6.2. Задачу 3.3 представьте на языке ИП и сравните производительность решений.

6.3. Выделите в описании мультипарадигмального языка чисто императивное подмножество

Algol-60, Algol W **Algol-68** Альфа- ADA Algol ассемблер assembler,

Basic BLISS, C C++ C#, C-light C-lang Cobol, Component Pascal
D, Fortran Клу Kotlin LLVM Pascal PL/1,

Лекция 7. Логическое программирование. Недетерминизм. Представление знаний. Технологии, практичность и продуктивность.

Исследовательская и проектная работы обычно проходят фазу поиска практичного решения. Логическое программирование (ЛП) для поддержки этой фазы предлагает важное отступление от концепции определения подпрограмм, процедур и функций. В качестве укрупненного определения действий допускаются варианты, равноправно выбираемые из конечного множества так называемых «клауз». По мере изучения задачи это множество можно пополнять в произвольном порядке. Именно эта идея составляет одну из привлекательных особенностей ЛП, выделившегося в самостоятельную парадигму. Равноправие не распространяется лишь на тупиковую ситуацию, когда ни один предложенный вариант не приводит к целевому результату.

Логическое программирование основано на теории и аппарате формальной логики. Написанная согласно формальной логике программа является множеством логических форм, представляющих факты и правила относительно некоторой предметной области. Основным языком логического программирования признан Prolog, хотя известны и другие – Planner, ASP и Datalog. Во всех таких языках правила имеют форму клауз:

$$H :- B1_1, \ldots, BN_n,$$

понимаемую как логическое следование

$$\text{if } (B1_1 \text{ and } \ldots \text{ and } BN) \text{ then } H$$

или

$$B11 \ \& \ \ldots \ \& \ BN \rightarrow H$$

H называют головой правила, а $B1_1$, …, BN – телом.

Факты – это правила без тела. Различают атомарные и составные клаузы. Предикаты, образующие тело, могут быть выражены в стиле сопоставления с образцом. Важный механизм – использование отрицаний в теле клауз, что приводит к немонотонной логике[144]. Логика программ может использовать процедурный стиль при вычислении целей:

to solve H, solve B_1, and ... and solve B_n.

Декларативный подход к пониманию программ требует от программиста систематической проверки корректности[145]. Более того, используются преобразования логических программ в их более эффективные эквиваленты, что сближает ЛП с макротехникой. Для повышения эффективности программ программисту следует знать особенности поведения механизма вычислений и границы вычислимости используемых

---

[144] – В классической математике многое привязано к исследованию монотонных функций, которые непрерывно возрастают или убывают. Существует ряд альтернативных математик, в которых рассматриваются неклассические логики, не придающие большого значения закону исключённого третьего, меняются таблицы истинности и функции могут произвольно переходить от возрастания к убыванию и наоборот. Это менее удобно для привычной математической технике доказательств, но более соответствует компьютерным реалиям вычислений. Механизм отрицания — основа доказательств от противного, опирающегося на закон исключённого третьего. Без него нет оснований из ложности «А» делать вывод об истинности «не А», т. к. могут быть другие истинностные значения.

[145] Мнение, что декларативный подход избавлен от необходимости описания пути к достижению результата, а значит вероятность ошибки в сравнении с императивным меньше, может не подтверждаться. Кроме независимости, часто иллюзорной, от описания пути, бывают другие категории некорректностей, начиная с соответствия типов данных, сигнатур функций, последовательности сопоставления образцов, вычислимости и пр.

выражений (C). Парадигма логического программирования использует идею автоматического вывода информации на основе заданных фактов и правил (E).

7.1. Операционная семантика

Логическое программирование сводит обработку данных к выбору произвольной композиции определений (уравнений, предикатных форм), дающей успешное получение результата. Именно обработка формул является основой – вычисления рассматриваются как операция над формулой (M). При неуспехе происходит перебор других вариантов определений (S). В языках ЛП считают возможным прямой перебор вариантов, сопоставляемых с образцами, и организацию возвратов при неудачном выборе - бэктрекинг[146]. Перебор вариантов выглядит как обход графа в глубину. Имеются средства управления перебором с целью исключения заведомо бесперспективного поиска.

Интерпретирующий автомат для выполнения недетерминированных процессов можно представить как цикл продолжения вычислений при попадании в диагностическую ситуацию [147]. Продолжение заключается в выборе другого варианта из набора определений функционального объекта. Вычисление признается неудачным, лишь если не удалось подобрать комплект вариантов, позволяющий вычислить значение формулы[148].

В отличие от множества элементов набор вариантов не требует одновременного существования всех составляющих. Поэтому программирование вариантов можно освободить от необходимости формулировать все варианты сразу. В логическом программировании можно продумывать варианты отношений между образцами формул постепенно, накапливая реально встречающиеся факты и их сочетания. Содержательно такой процесс похож и на уточнение набора обработчиков прерываний на уровне оборудования. Кроме основной программы, выполняющей целевую обработку данных, отлаживается коллекция диагностических реакций и процедур продолжения счета для разного рода неожиданных событий, препятствующих получению результата программы.

Следует иметь в виду, что варианты не образуют иерархии. Их аксиоматика подобна так называемой упрощенной теории множеств. Принципиальная особенность – совпадение предикатов принадлежности и включения.

Если варианты в таком выражении рассматривать как равноправные компоненты, то неясно, как предотвратить преждевременный выбор пустого списка при непустом перечне вариантов. Чтобы решить эту задачу, в системах ЛП вводится специальная форма ESC (ТУПИК), действие которой заключается в том, что она как бы «старается» по возможности не исполняться. Иными словами, при выборе вариантов предпочитаются варианты, не приводящие к исполнению формы ESC[149]. Такая же проблема возникает при обработке пустых цепочек в грамматиках. Аналогичная проблема решена при моделировании процессов интерпретированными сетями Петри соглашением о приоритете нагруженных переходов в сравнении с пустыми.

АС сводим к формуле:

---

146 – Бектрекинг здесь означает технику автоматического продолжения вычислений при обнаружении каких-либо неудач. Для этого в программе выделяют заранее точки выбор вариантов, чтобы при неудаче с одним из них можно было испытать другой. Это похоже на внешний цикл обработки прерываний, допускающий продолжение счёта.

147 – Чисто технически обработка исключений может быть рассмотрена как модель недетерминизма потому, что обычная практика обработки исключений заключается в минимально краткой немедленной реакции типа сообщения и отложенной на неопределённый срок собственно обработке исключения, которая выполняется системой в подходящее для системы время, не известно когда конкретно.

148 В языки ФП этот механизм внедрился по следам ЛП и как правило ограничен вычислением конкретных предикатов, включая синтаксические позиции в структурах данных. В ЛП неуспех в сопоставлении рассматривается как повод для перебора других вариантов, невычислимость предиката в таком случае не прерывает общего хода вычислений.

149 – Можно заранее выполнять символьный анализ, но чаще просто нарываясь на ТУПИК делают возврат на ближайшую точку ветвления и проверяют, есть ли там нехоженая ветка

(факт | (предикат цель) | ESC)
AML = <SCL, RL>, где RL = <S, E, C, D, R>

s e c d r → s' e' c' d' r' – переход от старого состояния к новому.
s e c d те же, что и в ФП, r – предназначен для хранения не опробованных вариантов.

В книге Хендерсона приведено обобщение абстрактной машины, поддерживающее на базовом уровне работу с вариантами с использованием дополнительного дампа, гарантирующего идентичность состояния машины при переборе вариантов.

s e c d r → s' e' c' d' r'

Таблица 7.1

Расширение абстрактной машины для выполнения альтернатив ЛП

Команда	*Пояснение*
ALT	выбор подходящего варианта.
ESC	выход из тупиковой ситуации.

Таблица 7.2

Определение команд ЛП

Формат регистров	*Результат*
s e (ALT c1 c2 . c) d r	→ s e (c1 . c) (c . d) ((s e (c2 . c) d) . r)
s e (ESC) d Nil	→ Nil e (Esc) d Nil
s' e' (ESC) d' ((s e (c2 . c) d) . r)	→ s e (c2 . c) d r

7.2. Основы ЛП

Представление вариантов в чем-то подобно определению ветвлений, но без предикатов, управляющих выбором ветви, что по реализации напоминает вариантные записи или объединения в обычных ЯВУ или обработку прерываний. В некоторых языках, например, учебно-игрового характера, можно указать вероятность выбора варианта. В языках логического и генетического программирования считают возможным прямой перебор вариантов, сопоставляемых с образцами, и организацию возвратов при неудачном выборе.

Обычно понятие алгоритма и программы связывают с детерминированными процессами. Однако эти понятия не очень усложнятся, если допустить недетерминизм, ограниченный конечным числом вариантов, так что в каждый момент времени из них существует только один вариант[150].

В любом выражении можно выполнить разметку ветвей на нормальные и тупиковые. Тупики можно связать с различными тэгами и выставить ловушки на заданные тэги. При попадании в тупик формируется значение всей структуры, размещенной внутри ловушки.

Используя тупики и ловушки, можно организовать перебор вариантов до первого беступикового варианта или собрать все беступиковые варианты. Второе можно сделать, используя отображения (map), а первое допускает метод первого подходящего, чо можно реализовать как слегка модифицированный evcon с добавочной ловушкой на прерывание при достижении успеха.

Более сложно обеспечить равновероятность выбора вариантов. Наиболее серьезно возможность такой реализации рассматривалась в проекте языка *SETL*. Похожие механизмы используются в языках, ориентированных на конструирование игр, таких как Grow, в которых можно в качестве условия срабатывания команды указать вероятность.

---

[150] – Кроме сопоставления с образцом современные ЯП активно используют циклы типа всеобщности и существования, стражи и недетерминированную обработку ситуаций, а также предикатную форму представления программ для нужд верификации. Это вызвано актуальностью обработки большеобъёмных данных, где программирование более точных методов вызывают затруднения. ЛП освобождает от опасности преждевременной конкретизации методов выбора результатов, помогает постепенно двигаться к более полному своду данных для показа решаемости нужной задачи, возможно, для доведения её до уровня, пригодного для ФП.

В задачах искусственного интеллекта работа с семантическими сетями, используемыми в базах знаний и экспертных системах, часто формулируется в терминах фреймов-слотов (рамка-щель), что конструктивно очень похоже на работу со списками свойств атома в языке Lisp. Каждый объект характеризуется набором поименованных свойств, которые, в свою очередь, могут быть любыми объектами. Анализ понятийной системы, представленной таким образом, обычно описывается в недетерминированном стиле.

7.3. Язык декларативного программирования Prolog

Prolog является наиболее известным языком ЛП общего назначения, ассоциируемый с проблемами искусственного интеллекта и компьютерной лингвистики. Он базируется на логике первого порядка и в отличие от обычных ЯП приоритетно использует декларативность. Логическая программа выражается в терминах отношений, представляемых как факты и правила. Применялся для автоматизации доказательств теорем и разработки экспертных систем, включая лингвистические процессоры для естественных языков. Выполнение логической программы понимается как ответ на запрос над системой отношений. Отношения и запросы конструируются из термов и клауз.

Язык программирования Prolog предложен в 1972 году А. Колмерауэром. (Alain Colmerauer), процедурную интерпретацию языка выполнил Р. Ковальский (Robert Kowalski) и дал её описание в 1973 году, опубликованное в 1974 году. Практичность языка резко возросла благодаря созданию Д. Уорреном (David Warren) компилятора в 1977 году, что приблизило скорость символьной обработки к эффективности языка Lisp.

Существует Pure Prolog в качестве семантического базиса языка, удобного для изучения основных механизмов Prolog-машины. Итеративные алгоритмы можно программировать как рекурсивные функции. Prolog-машина ищет ответ на запрос в имеющейся системе отношений и, если находит, то сообщает его. Опубликованы алгоритмы мета-интерпретатора для языка Pure Prolog, показывающие его само-применимость и расширяемость.

Определение	*Примечание*

Factorial (N, F)
  Factorial (0, 1)
  Factorial (x+1, F) ← F :=
  Factorial (x, y) * (x + 1)
	Функция реализуется двумя клаузами:
	- констатация, что от 0 – значение 1;
	- декларация отношения между значениями функции на соседних числах

Пример 7.1.. Вычисление факториала

Определение	*Примечание*

partition([], _, [], []).
partition([X|Xs], Pivot, Smalls, Bigs) :-
  (X @< Pivot -> Smalls = [X|Rest], partition(Xs, Pivot, Rest, Bigs)
  ; Bigs = [X|Rest], partition(Xs, Pivot, Smalls, Rest)
).
 	
quicksort([]) --> [].
quicksort([X|Xs]) --> { partition(Xs, X, Smaller, Bigger) }, quicksort(Smaller), [X], quicksort(Bigger).
	Разбиение на участки
	
	Запуск сортировки

Пример 7.2.. Быстрая сортировка

Определение	*Примечание*

maplist(_P, [], []).
maplist(P, [X1|X1s], [X2|X2s]) :-
  call(P, X1, X2),
  maplist(P, X1s, X2s).
	Отображать нечего
	Можно выделить начало
	Вызов от первого
	Отображение остального

Пример 7.3. Отображение

Определение	*Примечание*
mi1(true). mi1((A,B)) :- mi1(A), mi1(B). mi1(Goal) :- Goal \= true, Goal \= (_,_), clause(Goal, Body), mi1(Body).	Равноправие вариантов. Продвижение к цели

Пример 7.4.. Расширяемый скелет Prolog-интерпретатора.

Определение	*Примечание*
provable(true, _) :- !. provable((G1,G2), Defs) :- !, provable(G1, Defs), provable(G2, Defs). provable(BI, _) :- predicate_property(BI, built_in), !, call(BI). provable(Goal, Defs) :- member(Def, Defs), copy_term(Def, Goal-Body), provable(Body, Defs).	Доказуемость Варианты. Встроенные свойства. Движение к цели.

Пример 7.5.. Выводимость цели. Обобщение интерпретатора

В определении функции provable(Goal, Defs)вырабатывается истина, если цель *Goal* выводима по отношению к *Defs*, представленным списком клауз в форме *Head-Body*. Полная реализация бектрекинга требует детерминированного накопления результатов. Отдельная альтернатива представляется как список целей и ветвей, которые могут использоваться как список альтернатив.

Определение	*Примечание*
mi_backtrack_([[]-G|_], G). mi_backtrack_(Alts0, G) :- resstep_(Alts0, Alts1), mi_backtrack_(Alts1, G).	Других вариантов нет. Перебор других вариантов.

Пример 7.6.. Бектрекинг. Возвраты при неудачном выборе клаузы

Если ни одна цель не доказана, то выбирается очередное решение из внутренней очереди. Вторая клауза описывает вычисление.

Существуют версии языка, приспособленные к работе с функциями высших порядков (HiLog, λProlog) и с модулями.

Стандарт ISO языка Prolog поддерживает компиляцию, хвостовую рекурсию, индексирование термов, хэширование, обработку таблиц.

Яркая реклама ЛП в рамках японского проекта компьютерных систем 5-го поколения утихла по мере прогресса элементной базы. Несмотря на широкое использование языка в научных исследованиях и образовании, логическому программированию пока не удалось внести существенный вклад в компьютерную индустрию. Весьма вероятно, что причина кроется в том, что любое производство предпочитает достаточно изученные задачи, а сильная сторона ЛП проявляется на классе недоопределённых задач.

Принято считать, что однозначное решение задачи в виде четкого алгоритма над хорошо организованными структурами и упорядоченными данными – результат аккуратной, тщательной работы, пытливого и вдумчивого изучения класса задач и требований к их решению. Эффективные и надежные программы в таких случаях – естественное

вознаграждение. Однако в ряде случаев природа задач требует свободного выбора одного из вариантов – выбор произвольного элемента множества, вероятности события при отсутствии известных закономерностей, псевдослучайные изменения в игровых обстановках и сценариях, поиск первого подходящего адреса для размещения блока данных в памяти, лингвистический анализ при переводе документации и художественных текстов и т. д. При отсутствии предпочтений все допустимые варианты равноправны, поэтому технология их отладки и обработки должна обеспечивать формально равные шансы вычисления таких вариантов (похожая проблема характерна для организации обслуживания в сетях и выполнения заданий операционными системами. Все узлы и задания сети должны быть потенциально достижимы, если нет формального запрета на оперирование ими.).

7.4. Функциональная модель ЛП

Первые реализации ЛП были выполнены на языке Lisp, поэтому основные механизмы можно рассмотреть как обработку списков.

По смыслу выбор варианта похож на выбор произвольного элемента множества:

{ a | b | c } = э { a, b, c }

Чтобы такое понятие промоделировать обычными средствами, нужны дополнительные примитивы. Например, при определении функции, выбирающей произвольный элемент списка, в какой-то момент L становится пустым списком, и его разбор оказывается невозможным, тогда действует вариант ESC.

Чтобы определить выбор произвольного элемента из списка L, можно представить рекурсивное выражение вида:

(любой L) = { (CAR L) | (любой (CDR L)) }

По смыслу выбор варианта похож на выбор произвольного элемента множества:

{ a | b | c } = э { a, b, c }

Чтобы такое понятие промоделировать обычными средствами, нужны дополнительные примитивы с меньшей степенью организованности, чем вектора или множества.

Уточнённую схему выбора произвольного элемента списка можно представить формулой вида:

Определение	*Примечание*
(любой L) = { (car L) | (любой (cdr L)) | ESC }	Множество из первого элемента списка, выбора любого из оставшихся элементов списка и тупика

Пример 7.7.. Тупик как равноправный вариант

Более ясное определение имеет вид:

Определение	*Примечание*
(любой L) = { (CAR L) | (любой (CDR L)) | (if (nl L) ESC) }	Выбор тупикового варианта возможен лишь при отсутствии других

Пример 7.8.. В какой-то момент L становится пустым списком, и его разбор оказывается невозможным. Тогда действует ESC

Другие построения, характерные для теории множеств:

{ x | P(X) } – множество элементов, обладающих свойством P.

Определение	*Примечание*
(F L) = {(if (P (CAR L)) (CONS (CAR L) (F (CDR L))))	Соединяются в список по проверке предиката

	(if (nl L) ESC) }	Тупик

Пример 7.9.. Выбор элементов списка, удовлетворяющих предикату

Определение, данное в этом примере, недостаточно, т. к. порождаемые варианты элементов, удовлетворяющих заданному свойству, существуют в разные моменты времени и могут не существовать одновременно. Чтобы иметь все варианты одновременно, требуется еще один примитив ALL, обеспечивающий накопление всех реально осуществимых вариантов.

Определение	*Примечание*
(F L) = (ALL {(if (P (CAR L)) (CONS (CAR L) (F (CDR L)))) | (if (nl L) ESC) })	Специальная форма для накопления всех результатов.

Пример7.10. Множество всех элементов списка, удовлетворяющих предикату

Определение	*Примечание*
(ALL (LAMBDA (x y) { (if (= x y) x) | ESC }) (любой A) (любой B))	Форма выбора элементов пересечения Перебор элементов двух множеств в произвольном порядке

Пример 7.11.. Пересечение множеств A и B

Определение	*Примечание*
(if (not a) NIL b)	a & b

Пример 7.12. b вычисляется лишь при истинном a, что результативно, но не всегда соответствует интуитивным ожиданиям (логика, предложенная в свое время Маккарти, позволяет добиться высокой эффективности).

Математически более надежны варианты, исключающие зависимость от порядка перебора:

Определение	*Примечание*
(ALL(LAMBDA x { (if (not x) NIL) | ESC }) {a | b})	Если a и b оба истины, то получается ESC

Пример 7.13. Такое значение отличается от NIL тем, что работает как истина

Аналогичная проблема возникает при построении ветвлений.

Определение	*Примечание*
((LAMBDA L {(COND ((eval (caar L)AL) (eval (cadr L)AL))) | ESC }) (любой ((p1 e1) (p2 e2) ...)))	Снятие зависимости от порядка записи

Пример 7.14. (cond (p1 e1) (p2 e2) ...)

Поддержка вариантов, каждый из которых может понадобиться при построении окончательного результата, находит практическое применение при организации высокопроизводительных вычислений. Например, мультиоперации можно организовать с исключением зависимости от порядка отдельных операций в равносильных формулах.

Определение	*Примечание*
((LAMBDA (x y z) {(if (< (+ x y) K) (+ (+ x y) z)) | ESC}) {(a b c) | (b c a) | (c a b)})	Обеспечение максимальной вычислимости

Пример 7.15.. a+b+c = (a+b)+c = a+(b+c) = (a+c)+b – профилактика переполнения

7.5. Модели недетерминизма

Необходимая для такого стиля работы инструментальная поддержка обеспечивается в GNU Clisp механизмом обработки событий *throw-catch*, для которого следует задать примерно такое взаимодействие.

Определение	*Примечание*
(DEFUN vars (xl)(catch 'ESC (COND ((null xl)(escape)) ((CAR xl) (CONS (CAR xl)(vars (CDR xl))))))) (DEFUN escape () (throw 'ESC NIL))	перебор вариантов до первого тупика vars not NIL сигнал о попадании в тупик
(print(vars ())) (print(vars '(a))) (print(vars '(a b c))) (print(vars (list 'a 'b (vars ()) 'c)))	Демонстрация

Пример 7.16. Механизм обработки событий throw-catch как модель недетерминизма вариантов

Следует отметить неисчерпаемый ряд задач, при решении которых удобно используются недетерминизм.

1. Обоснование упорядочений в традиционных алгоритмах. Выделяется доалгоритмический уровень, на котором просто анализируются таблицы возможных решений и постепенно вырабатываются комплекты упорядочивающих условий и предикатов.
2. Переформулировка задач и переопределение алгоритмов с целью исключения необоснованных упорядочений – одна из типовых задач оптимизации, особенно при переходе от обычных программ к параллельным. Приходится выяснять допустимость независимого исполнения всех ветвей и управляющих их выбором предикатов.
3. Обобщение идеи абстрактных машин с целью теоретического исследования, экспериментального моделирования и прогнозирования недетерминированных процессов на суперкомпьютерах и многопроцессорных комплексах (многопроцессорная машина Тьюринга и т.п.).
4. Конструирование учебно-игровых программ и экспериментальных макетов, в которых скорость реализации важнее, чем производительность.
5. Описание и реализация недетерминизма в языках сверхвысокого уровня, таких как Planner, *Setl*, Sisal, Id, Haskell и др.
6. Недетерминированные определения разных математических функций и организация их обработки с учетом традиции понимания формул математиками.
7. Моделирование трудно формализуемых низкоуровневых эффектов, возникающих на стыке технических новинок и их массового применения как в научных исследованиях, так и в общедоступных приборах.
8. Обработка и исследование естественно языковых конструкций, речевого поведения, культурных и творческих стереотипов, социально-психологических аспектов и т. п.
9. Организация и разработка распределенных вычислений, измерений, Grid-технологий, развитие интероперабельных и телекоммуникационных систем и т. п.

В истории ЛП можно выделить ряд специфических линий:

– абдуктивное логическое программирование;
– металогическое программирование;
– логическое программирование в ограничениях;
– параллельное логическое программирование (FGCS – японский проект 5-го поколения компьютерных систем);
– индуктивное логическое программирование;
– линейное логическое программирование;
– объектно-ориентированное логическое программирование;

– транзакционное логическое программирование.

7.6. Специфика

Таблица 7.3

Парадигмальная характеристика языков логического программирования

Параметр	*Конкретика*
Стартовая проблема	Накопление знаний о новых задачах для разработки алгоритмов их решения.
эксплуатационная прагматика ЯП	Неопределенные постановки задач. Эмпирическое накопление рецептов, достаточное для решения некоторых практических задач.
регистры абстрактной машины	S E C D R S – стек промежуточных результатов. E – стек локальных переменных. C – стек основной программы. D – дамп для защиты и восстановления данных. R – стек для перебора вариантов. Результат – заключение об успехе-неудаче вывода цели.
категории команд абстрактной машины	Кроме типов команд, характерных для ФП: - выбор варианта; - восстановление контекста при переборе вариантов.
реализационная прагматика	В дополнение к ФП обработка разностных списков, отслеживание низкого приоритета тупиков и сечений. Вместо произвольного выбора варианта реально варианты перебираются в порядке представления.
парадигмальная специфика	Расширение класса решаемых задач использованием недетерминированных моделей.

Таблица 5.4

Паспорт ЛП на примере языка Prolog

	ЛП: Prolog	*Примечание*
Типовая задача	Использование методов логического вывода при подготовке программ решения недоопределённых задач.	*Расширяется жизненный цикл программ, появляется фаза предварительной подготовки.*
Формат семантических систем	**< V, F; R >** — Функции могут быть значениями, а правила не могут быть функциями	*Накопительные эффекты допускают переход правил в данные*
Особенности правила применения операций (R)	При переборе вариантов решения используется бектрекинг, позволяющий продолжать вычисления после неудачных попыток.	*Схема вычислений обогащается методами сопоставления с образцом и представления разностных списков.*
Приоритеты	**CESM** **C E S M** Бектрекинг; вычисления; шаблоны; имена;	*Порядок принятия решений начинается оптимизацией перебора вариантов*
АМ	**secdr** — Бесконечные регистры с защитой неизменяемых локальных данных и вариантов	*Архитектурная модель дополнена регистром для вариантов.*
АС	**АС = ЛП**	*Нет необходимости в промежуточном языке*
Диалекты	**BDXP** **B D X P** Базис; частичность; расширения; устройства	*Восходящая методика освоения*с вычислений и шаблонов для сопоставления клауз с образцом

Выводы:

1. Языки, поддерживающие ЛП, смягчают требования к полноте и изученности постановки решаемых задач, что способствует расширению сферы применения ЯП.
2. Существуют классы задач, для которых модели не детерминированных вычислений являются естественными.
3. Многие ЯП включают в себя декларативные средства для организации контроля правильности программ.
4. Некоторые средства ЛП, например, сопоставление с образцом, постепенно проникают в современные мультипарадигмальные ЯП.

Результатом ЛП является демонстрация принципиальной возможности решения поставленной задачи, что похоже на доказательство теоремы существования.

Практика-7: ЛП-ФП — моделирование

7.1. Определите вывод максимального числа из трёх так, чтобы он не зависел от порядка перебора вариантов.

7.2. Представьте шаблон и защитные условия для выбора третьего элемента списка.

7.3. Опишите механизмы логического программирования в конкретном языке.
Haskell. Hope, Hope+, ML, Oz. Planner, PL/C,
Prolog, Pure Prolog, Python, Racket, Smalltalk, Snobol,

ЛЕКЦИЯ 8. Объектно-ориентированное программирование

При переходе к практическому программированию обычно возникают проблемы, связанные с изменением отношения к постановке задачи и оценке методов её решения в процессе подготовки, отладки и эксплуатации программы.

Прежде всего, необходимость в целостном решении полной задачи «с нуля» теперь встречается достаточно редко. Как правило, необходимо учитывать, что отчасти задача или похожие задачи уже имеют готовые решения. Их надо найти, изучить и выбрать фрагменты, наиболее подходящие для конструирования практичного решения нужной задачи. Обычно сложные задачи декомпозируются на модули, разрабатываемые отдельно, но функционировать модулям предстоит совместно. Кроме того, снижение трудозатрат на создание программ достигается программированием многократно используемых компонентов, условия применения которых заранее не известны.

Решение таких проблем требует углубленного анализа структур данных и логики вычислений, обоснованного выбора методов обработки данных и декомпозиции программы на удобно комплексируемые фрагменты. Объектно-ориентированное программирование (ООП) в настоящее время – самый популярный подход к решению этих проблем для широкого класса задач, не требующих предельно жестких эксплуатационных характеристик.

ООП рассматривает информационный процесс как частичную обработку объектов посредством реагирования на события с помощью методов, выбираемых в зависимости от типа обрабатываемых данных. Центральный момент – структурирование множества частных методов, используемых в программе, в соответствии с иерархией классов объектов, обрабатываемых этими методами (S), в предположении, что определяемые в программе построения могут локально видоизменяться при сохранении основных общих схем информационного процесса. Это позволяет выполнять модификации программы объявлением новых подклассов и дописыванием методов обработки объектов отдельных классов без радикальных изменений в ранее отлаженном тексте программы (M).

Связь методов с классами объектов позволяет вводить одноименные методы над разными классами объектов (полиморфизм), что упрощает логику управления: на уровне текста программы можно не распознавать принадлежность классу – это сделает СП (C). Именно так обычно реализовано сложение и другие операции, одинаково изображаемые для чисел, строк, векторов, множеств и т. п. Основной принцип – выбор метода в зависимости от типа данных (E).

Сборка программы из автономно развивающихся компонентов опирается на формулировку достигаемой ими цели, понимание которой гарантирует не только корректность полученного результата, но и рациональность его использования. Формулировать цели частей программы – процесс нетривиальный. В его основе лежат различные подходы к классификации понятий. Для демонстрации преимуществ ООП следует рассматривать задачи, решение которых требует более одного шага. Первый шаг – программирование ядра программы или его комплектация из готовых модулей с оформлением иерархии классов объектов, содержащих первичный комплект методов. Второй и последующие шаги – развитие иерархии классов и/или перегрузка методов. Задачи, постановки которых допускают более одного сценария развития, потребуют ещё одного специального шага: определение абстрактных/виртуальных классов в качестве точек варьирования возможного развития постановки задачи.

8.1. Общее представление

ООП объединяет в рамках единой методики организации программ классификацию на базе таких понятий как класс объектов, структура данных и тип значений, методы обработки данных и наследование свойств классов. Тип значений обычно отражает спектр основных низкоуровневых реализационных средств, учет которых обеспечивает эффективность кода программы, получаемого при компиляции. Структура данных обеспечивает

конструктивность построений, гарантирует ясный доступ к частям, из которых выстроено данное любой сложности. Класс объектов характеризуется понятным контекстом, в котором предполагается их корректная обработка. Обычно контекст содержит определения структуры объектов и их свойства.

Текст программы одновременно представляет и динамику управления процессами и схему информационных потоков, порождаемых при исполнении программы. Кроме того, последовательность написания программы и ее модификации по мере уточнения решаемой задачи могут соответствовать логике, существенно отличающейся и от логики процесса выбора системных и реализационных решений и от логики применения реализованных решений. Обычно программирование скрывает сложность таких деталей управления процессами путем сведения его к общим функциям, преобразующим любые аргументы в определенные результаты. Модификации программы при развитии решаемой задачи нередко осуществляются непосредственно ручной переработкой текста программы и определений входящих в нее функций.

При анализе задач, решаемых в терминах объектов, некоторая деятельность описывается так, что постепенно продумывается все, что можно делать с объектом данного класса. Потом в программе достаточно лишь упоминать методы обработки объекта. Если методов много, то они структурированы по иерархии классов, что позволяет автоматизировать выбор конкретного метода. На каждом уровне иерархии можно немного варьировать набор методов и структуру объектов. Таким образом, описание программы декомпозируется на интерфейс и реализацию, причем интерфейс скрывает сложность реализации так, что можно обозревать лишь необходимый для использования минимум средств работы с объектом.

Исходная гипотеза при программировании работы с объектами: объект не изменен, если на него не было воздействий из программы.

Однако реальность зачастую требует понимания и учета более сложных обстоятельств, что может существенно продлить время жизни программы или ее компонентов. В таком случае удобно предоставлять объектам большую свободу, сближающую их с понятием субъекта, описание которого содержит все, что он может делать. Программа может давать ему команды-сообщения и получать ответы-результаты.

Фактически субъектом является супер-класс, объединяющий классы объектов, обрабатываемые одноименными методами, т. е. функциями одного семейства. Так, при организации сложения можно считать, что существует супер-класс «слагаемые», которые умеют складываться с другими слагаемыми. При этом они используют семейство функций – методов, реализующих сложение. В зависимости от полноты семейства результат может быть получен или не получен. Семейство легко пополняется добавлением одноименных функций с новыми комбинациями типов параметров.

Дальнейшее развитие подходов к декомпозиции программ связано с выделением отдельных аспектов и шагов при решении сложных задач. Понятие аспекта связано с различием точек зрения, позволяющим описывать решение всей задачи, но отражать в описании только видимые детали. По мере изменения точек зрения могут проступать новые детали до тех пор, пока дальнейшая детализация не утрачивает смысл, т. е. улучшение трудно заметить или цена его слишком высока. Так, представление символьной информации в языке Lisp выделено в отдельный аспект, независимый от распределения памяти, вычисление значений четко отделено от компиляции программ, понятие связывания имен с их определениями и свойствами не зависит от выбора механизмов реализации контекстов исполнения конструкций программы.

Разработка сложной программы может рассматриваться как последовательность шагов процесса раскрутки программ[151], оправданным в тех случаях, когда целостное решение

---

[151] – Термин «раскрутка» теперь заменился на «инкрементальная или пошаговая разработка». «Целостное решение» подразумевает, что в графике разработки нет выделения отдельных этапов со сдачей промежуточных результатов. Такое встречается, особенно при разработке ПО для эксплуатации

задачи не может гарантировать получение приемлемого результата в нужный срок – это влечет за собой непредсказуемо большие трудозатраты.

Удобный подход к организации программ «отдельная работа отдельно программируется и отдельно выполняется» успешно показал себя при развитии операционной системы UNIX как работоспособный принцип декомпозиции программ. Но существуют задачи, например, реализация СП, в которых прямое следование такому принципу может противоречить требованиям к производительности. Возможен компромисс «отдельная работа программируется отдельно, а выполняется взаимосвязано с другими работами», что требует совмещения декомпозиции программ с методами сборки – комплексации или интеграции программ из компонентов. Рассматривая комплексацию как еще одну «отдельную» работу, описываемую, например, в терминах управления процессами, можно констатировать, что эта работа больше определяет требования к уровню квалификации программиста, чем объем программирования. При достаточно объективной типизации данных и процессов, возникающих при декомпозиции и сборке программ определенного класса, строят библиотеки типовых компонентов и разрабатывают компонентные технологии разработки программных продуктов – Corba, COM/DCOM, UML и т. п. Одна из проблем применения таких компонентов – их обширность[152].

Таким образом, ООП может отражать эволюцию подходов к организации структур данных на уровне задач и программ их решения, исходя из парадигмы императивно-процедурного программирования. От попыток реализации математически корректных абстрактных типов данных произошел практичный переход к технически простому статическому контролю типов данных при разработке и применении расширяемых программ. Расширение программы выполняется декларативно, а выбор нужного варианта при исполнении функций, обладающих неединственным определением, – в зависимости от типа данных. Введены дополнительные механизмы: инкапсуляция, уточнение типов данных при компиляции и выбор обработчиков данных, управляемый типами данных.

Механизмы ООП обеспечивают наследование свойств по иерархии классов объектов и так называемый «дружественный» доступ к произвольным классам[153]. Расширение программ при объектно-ориентированном подходе к программированию выглядит как простое дописывание новых определений. Библиотеки типов данных и методов их обработки легко вписываются в более общие системы. Спецификация интерфейсов в принципе может быть сопровождена верификацией реализации компонент. Возможна факторизация программ на компоненты и рефакторизация программных компонент в стиле экстремального программирования.

нестандартных устройств или несложных программ. При разработке сложных систем, включая разработку систем программирования или операционных систем, обычно выделяют минимальное ядро и этапы его расширения. Эксплуатация таких систем может начинаться с готовности ядра, но с некоторым ограничение на квалификацию пользователей. По мере расширения системы обычно повышаются эксплуатационные характеристики системы, достигается полнота реализации, что удобно представлять в парадигме ООП.

[152] На первый взгляд, всего лишь найти в библиотеке модуль легче и надёжнее, чем его заново разрабатывать и реализовывать. Но это может оказаться трудозатратнее потому, что найдя библиотечный модуль, надо разобраться в технике его применения, узнать реальные границы параметров, а их может быть более десятка, проверить его надёжность и безопасность. не считая возможность его адаптации к своей задаче. Всё это нередко бывает сложнее, чем просто самому с нуля написать и отладить нужное, особенно в случае задач с развивающейся постановкой и расширяющимся кругом пользователей. Пока даже стандартные библиотеки не сопровождаются формальными спецификациями и в технологию их изготовления не входит формальная верификация.

[153] – Термин «дружественный доступ» появился в середине 1980-ых годов. Он означает возможность некие классы обозначать как дружественные «friend» и из них доступаться к объектам друг друга в обход иерархии классов.

- Если инкапсуляция — сокрытие или защита данных, то friend её нарушает. Если инкапсуляция — это совместное представление объектов и методов, то friend её не нарушает, а просто даёт возможность лаконичного доступа. Любые принципы в практике иногда требуют отклонения от них.

– В других языках такая возможность не оформлена как специальное средство. Часто вводятся возможности доступа через синонимы или указатели. Адреса в ООП никто не отменял.

ООП структурирует множество частных методов, используемых в программе, в соответствии с иерархией классов объектов, обрабатываемых этими методами, реализуемыми с помощью функций и процедур, в предположении, что определяемые в программе построения могут локально видоизменяться при сохранении основных общих схем информационной обработки. Это позволяет выполнять модификации объявлением новых подклассов и дописыванием методов обработки объектов отдельных классов без радикальных изменений в ранее отлаженном тексте программы.

8.2. Абстрактная машина

АС включает в себя аналоги ИП, ФП и ЛП с добавлением команд выбора метода в зависимости от класса объектов. АМ для языков ООП можно базировать на АМ для процедурно-императивных языков стандартного прикладного программирования. Абстрактная машина для ОО-языка по структуре похожа на объединение машин для ФП и ИП:

AMO = < RO, SCO>, где SCO= <S, E, C, D, M>, где

- S – стек операндов и результатов вычислений.
- E – значения локальных переменных при вызовах функций.
- C – текущий стек программы.
- D – дамп для восстановления контекста программы при выходе из метода.
- M – общая память, хранящая константы, методы и объекты с их сигнатурами.

Обозначения:
<[cm], ts> – код метода, таблица символов
<tsi (@cm, v), ret (#data)> – таблица метода,
v – символьные переменные, сигнатура возврата

Представление класса содержит сведения о числе констант и информацию о них, флаги доступа к полям объектов класса, ссылки на объект и супер-класс, число полей и информацию о них, число методов и информацию о них. Сигнатура представляется символьной характеристикой полей объекта или параметров метода. Значения реализованы как структура данных с тегами, задающими тип элемента данных. Классы, поля и методы не являются значениями и хранятся без тэга.

Таблица 8.1

Типичные команды виртуальной машины для языка ООП

Команда	*Пояснение*
NOP	Ничего не делает
RETURN	Возврат из функции
PUT	Установка поля в объекте
GET	Доступ к полю в объекте
INVOKE	Вызов метода
CHECK	Проверка соответствия типа объекта

Формат команд АМ имеет вид:

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

Таблица 8.2

Дополнительная спецификация команд виртуальной машины для языка ООП. (t – логическое значение)

Исходной состояние	*Результат*
s e (NOP . c) d m	s e c d m
s e (RETURN . c) d m	s e c d m
(f v . s) e (PUT i . c) d m	s e c d (m | m[f,i] = v)
(f . s) e (GET i . c) d m	(m[f,i] . s) e c d m

((a1 a2 ... aK) f .s) e (INVOKE sig . c) d m	s ((a1 a2 ... aK) . e) (f[sig] . c) d m
(Obj . s) e (CHECK type . c) d m	(t . s) e c d m

Главный путь к повышению продуктивности программирования связан с упрощением процесса отладки, который зависит от искусства декомпозиции постановок задач и программ их решения на такие комплекты компонент, часть которых можно найти в библиотеках готовых модулей, а часть можно при программировании довести до уровня многократно используемых компонент.

Современное состояние имеющихся технических решений в данной области характеризуется доминированием компонентных технологий, ориентированных на ООП, обеспечивающих классификацию конструктива на уровне понятий пользователя и его отображение на уровень целевых архитектур, представимых в терминах абстрактных машин. При таком подходе не получают полного выражения функциональная декомпозиция и системные решения промежуточного уровня, что отчасти компенсируется развитием аспектно-ориентированного подхода, выглядящего как мета-надстройка над ООП. Отдельный ряд трудностей вызывают приаппаратные оптимизации, требующие более тонкой детализации ниже традиционного уровня абстрактных машин.

Более реальна перспектива снижения трудоёмкости и повышения надёжности практического программирования повышением кратности использования библиотечных модулей в рамках много-языкового программирования на базе систем программирования, создаваемых из общего, а потому более отлаженного конструктива.

8.3. Переход от Си к C++

С концепцией ООП связано представление о возможности сокрытия информации, наследования определений по иерархии классов и полиморфизма реализации операций и функций. Переход к ООП в языке C++ привел к пересмотру некоторых решений на уровне языка и компилятора. Рассмотрим особенности C++ как наиболее популярного языка ООП, в котором достижима схема, обобщающая комплекс решений задачи в виде ацикличного графа с возможными горизонтальными связями. Такие решения направлены на программирование ряда версий решения задачи без отмены ранее отлаженных решений, но с формированием новых областей видимости, в которых устаревшая часть программы может быть просто оттеснена. Важно принять во внимание следующее:

– компилируемая программа на C++ представляет собой иерархию областей видимости определений элементов классов, доступ к которым регламентирован;

– компиляция методов, конструкторов, функций и перегруженных операций управляется форматом списка фактических параметров, что привело к более жёстким правилам объявления типов данных и ограничивает свободу конкретизации списка параметров при вызове функций;

– возникают рекомендательные средства повышать эффективность кодирования вызовов функций указанием на inline-включение;

– появляется уровень программирования шаблонов для представления общих схем обработки контейнерных типов разнотипных данных.

Программа	*Пояснение*
//HELLO.CPP #include <iostream.h> void main () { cont << "\nHello, World!\n" ; }[154]	Комментарий с именем файла программы Препроцессор с вызовом библиотеки Головная функция Вывод приветствия на стандартное устройство

Пример 8.1. Программа на языке C++

Практический выигрыш от ООП можно показать на технике применения средств вывода данных.

[154] Говорят, что на новых версиях такой код не идёт :) Авторы Си кичились решением проблемы переноса ПО.

Фрагмент	*Пояснение*
printf (“x = %d, y = % s”, x, y);	Вывод целого и строки. «x» должен быть целым, а «y» – строкой

Пример 8.2. Вывод по библиотеке функций *stdio.h*

Программист на С должен знать, что первый параметр задает формат вывода и отследить согласование его с числом и типами выводимых данных и обозначениями форматов для функции *printf.*

Фрагмент	*Пояснение*
cout << “x = ” << x << “, y = ” << y;	Поток вывода управляется фактическим типом данных.

Пример 8.3. Вывод по библиотеке классов *iostream.h*

Программисту на С++ достаточно перечислить элементы вывода в естественном порядке. Библиотека классов *iostream.h* содержит перегрузку операции «<<» для всех основных типов данных, что позволяет компилятору выбрать нужный шаблон кода программы, и программист освобожден от необходимости представлять в программе сведения о формате выводимых данных.

Учитывая примеры описания и определения классов, можно сделать вывод, что внешние изменения в тексте программы на С++ в сравнении с текстом на С выглядят как появление ряда новых спецификаторов, с помощью которых как бы задается разметка программы, на области видимости, связанные с иерархией классов и дисциплиной доступа к элементам структурированных объектов класса. В результате вместо анализа последовательностей изменения состояний памяти по всей программе достаточно проанализировать изменения в пределах синтаксически выделенных путей по иерархии наследования методов объектов.

Доступность вложенного класса ограничивается областью видимости лексически объемлющего класса. Если у класса есть конструктор, он вызывается всякий раз при создании объекта этого класса. Если у класса есть деструктор, он вызывается всякий раз, когда уничтожается объект этого класса.

Чтобы можно было описать массив объектов класса с конструктором, этот класс должен иметь стандартный конструктор, вызываемый без параметров. В описании массива объектов не предусмотрено возможности указать параметры для конструктора. Когда уничтожается массив, деструктор должен вызываться для каждого элемента массива.

Производный класс наследует базовый класс, он больше своего базового класса в том смысле, что в нем содержится больше данных, и определено больше функций. Производный класс сам, в свою очередь, может быть базовым классом. Такое множество связанных между собой классов обычно называют иерархией классов. Обычно она представляется деревом, но бывают иерархии с более общей структурой в виде ориентированного графа. У класса может быть несколько прямых базовых классов. Возможность иметь более одного базового класса влечет за собой возможность неоднократного вхождения класса как базового[155].

С помощью виртуальных функций можно иметь разные версии в разных производных классах, а выбор нужной версии при вызове – это задача транслятора. Тип функции указывается в базовом классе и не может быть переопределен в производном классе. Класс, в котором есть виртуальные функции, называется абстрактным. Абстрактный класс можно использовать только в качестве базового для другого класса.

Фрагмент	*Пояснение*

[155] – Существует управление наследованием, можно заранее дозировать, что именно из базового класса попадает в производный. В разных ЯП это несколько отличается. В общем может быть, что в производном классе больше функций, даже если он мало что наследует, но может быть и меньше, он может быть специально сделан как сужение базового.

class complex {
  double re, im;
 public:
  complex(double r, double i) { re=r; im=i; }
  friend complex operator+(complex, complex);
  friend complex operator*(complex, complex);
};
	Объявление класса объектов с закрытыми полями
	и общедоступными конструкторами и перегрузкой операций.

Пример 8.4. Перегрузка арифметических операций для их использования в выражениях над комплексными числами

Члены класса создаются в порядке их описания, а уничтожаются они в обратном порядке. Член класса может быть частным (private), защищенным (protected) или общим (public).

Фрагмент	*Пояснение*

void f()
{
  complex a = complex(1,3.1);
  complex b = complex(1.2,2);
  complex c = b;
 	
  a = b+c;
  b = b+c*a;
  c = a*b+complex(1,2);
}
	Объявление функции, выполняющей процедуру,
	
	конструирования 3-х комплексных чисел и их инициирования,
	
	и обработка чисел с помощью перегруженных операций.

Пример 8.5. Интерпретация этих операций задана определениями функций с именами operator+ и operator*

Если b и c имеют тип complex, то b+c означает (по определению) operator+(b,c). Сохраняются обычные приоритеты операций, поэтому второе выражение выполняется как b=b+(c*a), а не как b=(b+c)*a. При перегрузке операций нельзя изменить их приоритеты, равно как и синтаксические правила для выражений.

Для операций преобразования ТД выбран подход, при котором проверка соответствия ТД является строго восходящим процессом, когда в каждый момент рассматривается только одна операция с операндами, типы которых уже прошли проверку[156].

Вызов функции, т. е. конструкцию выражение(список выражений), можно рассматривать как бинарную операцию, в которой выражение является левым операндом, а список выражений – правым. Операцию вызова можно перегружать, как и другие операции.

Одним из самых полезных видов классов является контейнерный класс, т. е. такой класс, который хранит объекты каких-то других типов. Списки, массивы, ассоциативные массивы и множества – все это контейнерные классы.

Фрагмент	*Пояснение*

template<class T>
 	
class stack {
  T* v;
  T* p;
  int sz;
 	
public:
  stack(int s) { v = p = new T[sz=s]; }
  ~stack() { delete[] v; }
 	
  void push(T a) { *p++ = a; }
  T pop() { return *--p; }
	Объявление параметризованного шаблона.
	
	Для создания классов стеков в зависимости от задаваемого типа хранимых элементов при известном объёме стека.
	
	Общедоступны:
	конструкторы и деструкторы стека,

[156] – Грань между обобщенной парадигмой программирования и параметрическим полиморфизмом не очень строгая. Параметрический полиморфизм затрагивает лишь подстановку конкретных видов параметров в ограниченный спектр шаблонов, предназначенных для такой подстановки. Обобщённое программирование более нацелено на абстрагирование, допускающее и другие категории преобразований программы. Иногда полиморфизм рассматривают как частный случай обобщённой парадигмы.

int size() const { return p-v; } };	функции работы со стеком

Пример 8.6.. Шаблон типа для класса. Стек, содержащий элементы произвольного типа[157].

Префикс template<class T> указывает, что описывается шаблон типа с параметром T, обозначающим тип, и что это обозначение будет использоваться в последующем описании. После того, как идентификатор T указан в префиксе, его можно использовать как любое другое имя типа. Область видимости T продолжается до конца описания, начавшегося префиксом template<class T>. Имя шаблонного класса, за которым следует тип, заключенный в угловые скобки <>, является именем класса (определяемым шаблоном типа), и его можно использовать как все имена класса.

Поскольку все функции-члены класса stack являются подстановками, то и в этом примере транслятор создает вызовы функций только для размещения в свободной памяти и освобождения.

Функции в шаблоне типа могут и не быть подстановками.

В программе может быть только одно определение функции-члена класса и только одно определение шаблона типа для функции-члена шаблонного класса. Если требуется определение функции-члена шаблонного класса для конкретного типа, то задача системы программирования найти шаблон типа для этой функции-члена и создать нужную версию функции. В общем случае система программирования может рассчитывать на указания от программиста, которые помогут найти нужный шаблон типа. Возможна передача операций как параметров функций.

Рассмотренные средства представления иерархии классов объектов с возможностью множественного наследования, использования программируемых и встроенных конструкторов и деструкторов объектов, создания массивов объектов класса с контролем доступа к элементам, перегрузки операций и задания виртуальных функций, а также, операций преобразования ТД и шаблонов типа для обработки контейнерных структур данных суммарно образуют достаточно богатый арсенал для поддержки процесса практичной разработки программ при решении расширяющейся задачи, совмещённого с процессом декомпозиции программы на многократно используемые компоненты разного уровня абстрагирования от конкретики решаемой задачи и специфики системных реализационных решений.

При организации наследования в отличие от обобщенных функций работает модель обмена сообщениями*:*

– объекты обладают свойствами;

– посылают сообщения;

– наследуют свойства *и* методы от предков.

При переходе от обычного стандартного программирования с ООП связывают радикальное изменение способа организации программ. Это изменение произошло под давлением роста мощности оборудования[158]. ООП взламывает традиционное программирование по многим направлениям. Вместо создания отдельной программы, оперирующей массой данных, приходится разбираться с данными, которые сами обладают поведением, а программа сводится к простому взаимодействию данных новой категории – объектов.

---

[157] – В первых реализациях языка Forth не было проверки на переполнение стека. Это давало виртуозные эффекты знатокам распределения памяти в СП.

[158] – «Счастливые часов не наблюдают!» С ранних, пионерских времён программирования замечала, что имеет место феномен особой гордости программиста за то, что его программа работает много суток и жрёт безумное количество памяти. Не исключено, что такой программист гордится способностью добиться привилегий на такое расточительство. Теперь многие из таких программ не выглядят исключительными.

8.4. Функциональные модели ООП

Чтобы сравнить дистанцию ООП с ФП П. Грем (Paul Graham) в описании стандарта языка Common Lisp предлагает рассмотреть модель встроенного в Lisp объектно-ориентированного языка (ОО-язык), обеспечивающего основы ООП. Встраивание ОО-языка показывает характерное применение ФП – моделирование разных стилей программирования (начиная со стандартного программирования в виде prog-формы, предложенной Дж. Маккарти). В языке Lisp есть разные способы размещать коллекции свойств. Один из них – представлять объекты как хэш-таблицы и размещать свойства как входы в нее. П. Грем приводит пример (8 строк) реализации ООП на базе хэш-таблиц. Фактически наследование обеспечивает единственная особенность языка Lisp: все это работает благодаря реализации рекурсивной версии GETHASH. Впрочем, Lisp по своей природе изначально был ОО-языком. Определение методов может достичь предельной гибкости благодаря возможности генерировать определения функциональных объектов с помощью DEFMACRO или функцией категории FSUBR.

Реализация ООП с помощью хэш-таблиц обладает слегка парадоксальной окраской: гибкость у нее больше, чем надо, и за большую цену, чем можно позволить. Уравновесить это может подобная реализация на базе простых векторов. Этот переход показывает, как ФП дает новое качество «на лету». В опорной реализации фактически не было реализационного разделения объектов на экземпляры и классы. Экземпляр – это просто класс с одним-единственным предком. При переходе к векторной реализации разделение на классы и экземпляры становится реальным, также становится невозможным превращать экземпляры в классы простым изменением свойства.

Более прозрачная модель ООП на базе обычных списков, иллюстрирующая глубинное родство ФП и ООП, реализована в системе CLOS. Показанный ниже пример работает по первому аргументу (выбор подходящего метода рассчитан на то, что достаточно разобраться с одним аргументом), CLOS делает это на всех аргументах, причем с рядом вспомогательных средств, обеспечивающих гибкий перебор методов и анализ классов объектов. Рассмотрим примеры использования *к*лассов и экземпляров объектов:

```
(defclass ob () (f1 f2 ...))
```

Это означает, что каждое вхождение объекта будет иметь поля-слоты f1 f2 ... (Слот – это поле записи или списка свойств.) Чтобы сделать представителя класса, мы вызываем общую функцию:

```
(SETF c (make-instance 'ob))
```

Чтобы задать значение поля, используем специальную функцию:

```
(SETF (slot-value c) 1223)
```

До этого значения полей были не определены.

Простейшее определение слота – это его имя. Но в общем случае слот может содержать список свойств. Внешне свойства слота специфицируются как ключевые параметры функции. Это позволяет задавать начальные значения. Можно объявить слот совместно используемым:

```
:allocation :class
```

Изменение такого слота будет доступно всем экземплярам объектов класса. Можно задать тип элементов, заполняющих слот, и сопроводить их строками, выполняющими роль документации. Нет необходимости все новые слоты создавать в каждом классе, поскольку можно наследовать их из суперклассов.

Определение	*Пояснение*
(defclass expr () ((type :accessor td) (sd :accessor ft)) (:documentation "C-expression"))	Супер-класс для всех структур Lisp-выражений. Заданы ключи доступа к общим полям объектов: тип и операнд.
(defclass un (expr) ((type :accessor td) (sd :accessor ft)) (:documentation "quote car *other *adr"))	Класс для унарных форм. Доступ можно унаследовать от супер-класса, а здесь не дублировать.
(defclass bin (expr) ((type :accessor td) (sd :accessor ft) (sdd :accessor sd)) (:documentation "cons + lambda let"))	Класс бинарных форм. Третье поле для второго операнда.
(defclass trio (expr) ((type :accessor td) (sd :accessor ft) (sdd :accessor sd) (sddd :accessor td)) (:documentation "if label"))	Если взять супер-класс (bin), то можно не объявлять первые 3 поля. Четвёртое поля для третьего операнда.
(defmethod texrp ((x expr) (nt atom)) (SETF (slot-value x 'type) nt) (SETF (td x) nt) ;;--;; variant (:documentation "объявляем тип выражения"))	Метод представления выражений для компиляции.
(defmethod spread ((hd (eql 'QUOTE)) (tl expr)) (let ((x (make-instance 'un))) (SETF (ft x) (car tl)) (SETF (td x) hd)) (:documentation "распаковка выражения"))	Метод разбора констант и выражений с унарными операциями.
(defmethod compl ((hd (eql 'QUOTE)) (tl expr)) (list 'LDC tl) (:documentation "сборка кода"))	Метод компиляции констант.
(defmethod compl ((hd (eql 'CAR)) (tl expr) N) (append (compl(ft tl) N) '(CAR)) (:documentation "сборка кода"))	Метод компиляции выражений с унарными операциями.
(defmethod spread ((hd (eql 'CONS)) (tl expr)) (let ((x (make-instance 'bin))) (SETF (ft x) (CAR tl)) (SETF (sd x) (cadr tl)) (SETF (td x) hd)) (:documentation "распаковка выражения"))	Метод разбора выражений с бинарными операциями.
(defmethod compl ((hd (eql 'CONS)) (tl bin) N) (append (compl (sd tl) N) (compl (ft tl) N) '(CONS)) (:documentation "сборка кода"))	Метод компиляции выражений с бинарными операциями с прямым порядком компиляции операндов.
(defmethod compl ((hd (eql '+)) (tl bin) N) (append (compl (ft tl) N) (compl (sd tl) N) '(ADD)) (:documentation "сборка кода"))	Метод компиляции выражений с бинарными операциями с обратным порядком компиляции операндов
(defmethod spread ((hd (eql 'IF)) (tl expr)) (let ((x (make-instance 'trio))) (SETF (ft x) (CAR tl)) (SETF (sd x) (cadr tl)) (SETF (td x) (caddr tl)) (SETF (td x) hd)	Метод разбора выражений с триадными операциями.

) (:documentation "распаковка выражения"))	
(defmethod compl ((hd (eql 'IF)) (tl expr) N) (let ((then (list (compl(sd tl)N) '(JOIN))) (else (list (compl(td tl)N) '(JOIN)))) (append (compl(ft tl)N) (list 'SEL then else)))(:documentation "сборка кода"))	Метод компиляции выражений с триадными операциями.
(defmethod parh ((x expt)) (let (ftx (ft x)) (COND ((ATOM ftx) (spread 'ADR ftx)) ((member (CAR ftx) '(QUOTE CAR CONS + IF LAMBDA LABEL LET)) (spread (CAR ftx) (CDR ftx)) (T (spread 'OTHER ftx))))(:documentation "шаг разбора"))	Метод разбора произвольных выражений.

Пример 8.7.. ОО-определение Lisp-компилятора

CLOS, естественно, использует модель обобщенных функций, но мы написали независимую модель, используя более старые представления, тем самым показав, что концептуально ООП – это не более чем перефразировка идей языка Lisp. ООП – это одна из вещей, которую Lisp изначально умеет делать. Для функционального стиля программирования в переходе к ООП нет ничего революционного. Такой переход практически не расширяет пространство решений. Это просто небольшая конкретизация механизмов перебора ветвей функциональных объектов. При переходе от императивного стиля ООП компенсирует избыточную целостность представления отлаживаемых программ, смягчает жесткость зависимости компонентов программы от потока информационных процессов.

Более интересный вопрос, что же еще может дать функциональный стиль и традиция реализации функциональных СП? – Похоже, что это средства освоения новых архитектур и технологий.

Другая модель ООП, полученная на базе обычных списков свойств (атрибутов), также иллюстрирует глубинное родство ФП и ООП. Нужно лишь уточнить определение Lisp-интерпретатора, чтобы методы рассматривались как особая категория функций, обрабатываемая специальным образом.

8.5. Специфика

Таблица 8.3

Парадигмальная характеристика парадигмы ООП

Параметр	*Конкретика*
Стартовая проблема	Трудоёмкость повторного программирования отчасти решённых задач.
Эксплуатационная прагматика ЯП	Практичное программирование, нацеленное на разумный компромисс в пространстве противоречивых критериев с приоритетом критериям сферы приложения программ.
Регистры абстрактной машины	S E C D M S – стек промежуточных результатов E – стек локальных переменных C – текущая программа D – дамп для восстановления контекста при рекурсии. M – общая память хранимых объектов. Результат рассредоточен по именованным состояниям памяти.

Категории команд абстрактной машины	Загрузка в стек. Сохранение значений. Манипулирование стеком. Арифметические и логические операции. Передачи управления. Выбор определения метода, функции, операции. Вызов метода. Возврат из метода. Обработка исключений. Работа под монитором (параллелизм).
Реализационная прагматика	Сочетание статического представления методов с динамикой размещения объектов, включая автоматизацию повторного использования памяти. Представление сигнатуры для динамического выбора конкретного метода обработки объект.
Парадигмальная специфика	Процесс программирования сводится к последовательности расширяемых по мере целесообразности программ. Использование предметной типизации классов объектов как ведущего параметра выбора конкретной техники обработки данных.

Таблица 5.4

Паспорт ООП на примере языка C++l

	ООП: C++	*Примечание*
Типовая задача	Улучшение программируемых решений при уточнении постановки ранее решённой задачи.	Поддержан полный жизненный цикл программы, включая создание версий.
Формат семантических систем	**< V; F; R >** — Функции не могут быть значениями, а правила не могут быть функциями	Накопительные эффекты ограничены модулями и интерфейсами
Особенности правила применения операций (R)	Модель вычислений использует согласование типов и структур данных с методами их обработки (инкапсуляция и полиморфизм), а также учитывает геометрию доступа к ранее представленным решениям (наследование).	Представление кода программы и данных включает хранение сигнатур, по которым вычисляется совместимость.

Приоритеты	**SMCE** S M C E Структуры; имена; управление; вычисления	Порядок принятия решений начинается с иерархии классов и размещения объектов в памяти
AM	**secdm** — Регистры с защитой неизменяемых локальных данных и общей памяти.	Архитектурная модель дополнена общей памятью.
AC	**AC:** синтаксический граф разбора	Типовая системная модель
Диалекты	**PDBX** P D B X Устройства; частичность; базис; расширения	Нисходящая методика освоения, стартовый барьер от ввода-вывода и спецификации наследования

Выводы:

1. Языки, поддерживающие ООП, приспособлены к расширению программ при развитии постановок задач и разработке многократно используемых модулей[159].
2. Функциональные модели ООП показывают возможность использования преимуществ ООП в рамках других парадигм[160].
3. Современные мультипарадигмальные и учебные ЯП как правило поддерживают ООП.
4. Результат ООП заключается в создании иерархии классов объектов, допускающей профилактику повторного программирования при развитии постановки задачи[161]

---

[159] Общепринятая критика ООП связана с тем, что ЯиСП, поддерживающие ООП, требуют больших ресурсов оборудования и объекты затрачивают больше оперативной памяти. Можно учесть, что разброс производительности компиляторов примерно 1 к 50 и уж не менее, чем 1 к 10. ООП увеличивает потребность в ресурсах не в разы, а лишь на проценты, что на таком фоне не может быть принципиальным, поклонники ООП это просто игнорируют. Их привлекает продуктивность программирования, что в бизнес-приложениях важнее эффективности.

[160] Кажется, что изучение ООП сложнее, чем функционального программирования. Формально оно сложнее, но для многих понятнее, особенно если принять во внимание пресс работодателей и вузовских администраций, идущих у них на поводу. Согласно международным рекомендациям ООП надлежит изучать после ФП, параллелизма и императивности в качестве дополнительного материала вместе с ЛП. Ценители ФП считают, что ООП не вносит ничего нового в программирование в сравнении с ФП. Это справедливо отчасти, в ФП есть поддержка основных концепций ООП, таких как инкапсуляция, полиморфизм и наследование. Но они не были подняты на щит. Это значит, что в рамках ФП возможна подготовка ООП, что и показано системой CLOS.

[161] Такое применение ООП поддерживает построение гибких и расширяемых систем за счет использования паттернов проектирования, удовлетворяющих определенной задаче Это становится модным. Другое дело, в какой мере в ООП эта проблема решена. Повторное использование кода поддержано для задач, сводимых к обработке ориентированных графов. Для более сложных задач, требующих обработки неориентированных и динамически изменяемых графов, возникают трудности. Кто-то из классиков приводил пример про моделирование процессов превращения насекомых из яйца в гусеницу, затем куколку, а потом в бабочку.

Практика-8: ООП — ФП

8.1. Пусть определён класс целых чисел и метод их размещения в векторе. Постройте подобный класс для произвольных чисел с теми же именами методов.

8.2. Перегрузите операцию «+» так, чтобы она могла сцеплять однотипные однородные вектора.

8.3. Опишите решения для ООП в конкретном языке программирования:

C++ C#, Clisp (CLOS), Clojure, Cmucl, D, Eiffel, F# Go Golang Haskell. Java, JavaScript, Kotlin Oberon Object Pascal[Open MP Oz. Perl, Python (Rest) (Rebol) Ruby Racket Scala, Simula 67

Лекция 9. Мета-программирование

В данной лекции рассматривается парадигма мета-программирования, её потенциал, перспективы развития и отличия от других фундаментальных парадигм[162]. В центре внимания синтаксически управляемые методы конструирования программ.

9.0. Общее представление

Мета-программирование радикально отличается от остальных парадигм программирования уровнем данных. Эта парадигма поддерживает обработку программ, рассматриваемых как объект особо сложного класса или элемент очень сложного типа данных, называемого «язык программирования». Для мета-программирования несколько изменяется точка зрения на программу, преобразуемую как данное. Внешне программа состоит из комментариев, имён-идентификаторов, ключевых слов, разделителей-ограничителей, парных скобок и других синтаксически различаемых категорий лексем или синтаксических конструкций, разнообразие которых учитывается при определении шаблонов для выполнения преобразований программ. Преобразования могут сохранять функциональную эквивалентность программы (оптимизация, компиляция, распараллеливание, тестирование), а могут и существенно изменять её свойства (специализация, конкретизация, создание профилей и отладочных версий) в зависимости от целей и этапа разработки программы. Подготовка преобразований может требовать доказательств, похожих на доказательства теорем. Конверторы, освобождающие от лишних переходов на уровень машины, могут способствовать выделению эквивалентных подъязыков и, следовательно, типизации средств их реализации. Долгоживущие и новые ЯП подвержены улучшениям даже при самых серьёзных и авторитетных усилиях на стандартизацию их определений. Это требует механизмов работы с данными произвольной, непредсказуемой природы, что несколько противоречит идее статического контроля типов данных ради поддержки предельного потенциала методов эффективной кодогенерации. Всё это выполняется с активным применением мета-программирования.

Механизмы мета-программирования неявно присутствует в применении любого языка и системы программирования (ЯиСП). Элементы мета-программирования можно видеть в препроцессорах и функциях инструмента MAKE в Си, обеспечивающих сборку программы из модулей, а также в средствах открытой подстановки inline и template, влияющих на формирование кода при компиляции программы. В языках функционального программирования (ФП) традиционное мета-программирование имеет вид специальных конструкторов выражений и функций, включая блокировку вычислений и их возобновление (Quote-Eval и DefMacro в языке Lisp). Известна попытка в языке Scheme применять макросы с контролем типов данных. В новых языках, например, в Clojure имеется реструктурирование списка параметров и других структур данных, формирование проекций присутствует в языках Sisal и Haskell, операции изменения арности функций curry-uncurry включены в язык Haskell, квотирование кода в языке F# позволяет на этапе вычислений редактировать код программы как и динамические преобразования кода в языке C#. И конечно самое широко известное применение мета-программирования — это синтаксически ориентированная генерация анализаторов и компиляторов для языков, обладающих формализованным определением в виде форм Бэкуса-Наура (БНФ). Такая мощная техника синтаксического конструирования использует функции высших порядков, параметрами и результатами которых могут быть не только другие функции, но любые инструменты, доступные через операционную систему (ОС). БНФ можно рассматривать как язык мета-программирования языков программирования, особенно после появления Lex, YACC и их аналогов в новых языках программирования. Простейшие механизмы мета-программирования можно видеть в работе текстовых редакторов, выполняющих контекстную замену, орфокоррекцию, вывод оглавления, поддержку списка источников, трансформацию тек-

---

[162] https://www.youtube.com/watch?v=EwBjs55FzZw А.В. Климов «Почему анализ и преобразования программ (мета-вычисления) никак не идут в широкую практику программирования?» Андрей Валентинович Климов — один из разработчиков РЕФАЛа.

стов в таблицы и обратно, обтекание картинок и таблиц, переформатирование абзацев и страниц, размещение в тексте колонтитулов и сносок и т.п.

9.1. Схемы вычислений

Исследование разных схем частичных, смешанных, «ленивых» вычислений и суперкомпиляции приводит к выводу о целесообразности совмещения таких схем в рамках общей системы программирования (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. Одновременно строится остаточная программа, которую можно выполнить потом, когда появятся недостающие данные, 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. Если такую разметку не связывать с отсутствием данных, то получается модель, практически подобная вычислениям с задержками и *возобновлением*. «Ленивые» вычисления выполняются в зависимости от необходимости операций для получения результата с запоминанием промежуточных значений выражений, что позволяет экономить повторные вычисления. Во всех таких случая обрабатывается программа совместно с комплектом типовых данных, точнее, мета-программирование представляет собой технику обработки программ, учитывающей комплекты типовых примеров данных, образцов, позволяющих обосновывать программируемые или кодируемые решения.

Многие работы по преобразованию и оптимизации программ выполнены на функциональных моделях техникой функционального программирования (ФП), позволяющей сравнительно быстро и удобно выполнять отладку экспериментальных систем. Теперь это обыденная техника оптимизирующей компиляции. Не всегда неопределенность части данных мешает получить результат вычисления.

Рассмотрим

(if (< X Y) Z T)

или эквивалент

if X < Y then Z else T

Если X и Y не определены, но известно, что X лежит в интервале [1, 4], а Y в интервале [5, 9], то логическое выражение X<Y определено, и можно сделать вывод относительно выбора ветви условного выражения и, возможно, получить его значение.

Изучение смешанных вычислений может исходить из разных толкований понятия частичности, т.е. функций, определенных не на всей области их существования.

При подготовке программ на языке Lisp неопределенность часто представляют пустым списком, предполагая, что в него просто не успели что-то записать. Такое представление не всегда достаточно корректно и может потребовать дополнительных соглашений при обработке данных, по смыслу допускающих NIL в качестве определенного значения. Так, при реализации Lisp 1.5 введено соглашение, что значение атома в списке свойств хранится упакованным в список [199].

Поэтому практичнее неопределенные данные "накрывать" рецептами, использующими специальные функции, нацеленные на раскрытие неопределенностей.

Например, роль такой функции может сыграть запрос у пользователя дополнительной информации:

(A …) => (A . (||(READ)))

В определении интерпретатора обработка неопределенностей сосредоточена в функции ERROR.

(DEFUN EVAL (e AL)

…

```
  ((assoc e AL)(cdr (assoc e AL)))
  (T(ERROR '"неопределенная переменная"))
…
)
```

В определение функции ERROR можно включить обращение к READ, обрамленное сообщением о ситуации с информацией о контексте.

```
(DEFUN APPLY (f args AL)
…
  ((assoc f AL)(apply (cdr (assoc f AL))
    (evlis args AL)AL))
    (T (ERROR ‘'"неопределенная функция"))
  …
  )
```

При отладке сложных комплексов часто неразработанные определения замещают временными "заглушками", которые помогают разобраться в будущей программе по частям. Такую работу можно стандартизировать заданием предварительных определений функций в виде отображения типа аргументов в тип результата. Соответственно, исполнение предопределенной таким образом функции можно интерпретировать как проверку аргументов на соответствие типу аргументов и выдачу в качестве результата вариантов значения, принадлежащего типу результата.

При небольшом числе значений заданного типа, например, истинностные значения, может быть целесообразным полный перебор таких значений с последующим выбором реальной альтернативы пользователем.

```
(COND (e r)(T g))
=> (assoc e (list (CONS T (EVAL r AL))
                  (CONS NIL (EVAL g AL))) )
```

Таким образом, выполнятся обе ветви, их результаты ассоциируются с различными значениями заданного типа, что позволяет получить нужный результат, как только доопределится ранее не определенное значение. Это позволяет избежать повторного выполнения предшествующих вычислений, если их объем достаточно велик.

Применение библиотечных процедур, зависящих от слишком большого числа параметров, можно упростить для пользователя построением проекций на типовые комплекты трудно задаваемых параметров, понимаемых как определение режима работы процедуры.

```
(DEFUN f (x y z a b c … v t u) (g …))
(DEFUN Fi (x y z ) (f x y z ai bi ci … vi ti ui))
```

Примерно это и делает необязательный параметр вида &optional в языке Clisp.

Такое построение можно рассматривать как декомпозицию, разделение, сортировку на выполнимые и невыполнимые действия, при которой выполнимые действия в тексте определения замещаются их результатом, а невыполнимые преобразуются в остаточные, что все вместе образует проекцию процедуры на заданную часть ее параметров.

Многие выражения по смыслу используемых в них операций иногда определены при частичной определенности их операндов, что часто используется при оптимизации кода программ:

```
X * 0 = 0
CAR (A …) = A
X*1 = X при любом X
X-X = 0
X/X = 1 и т.п.
```

Представление функции в некоторых точках при отладке можно задать ассоциативной таблицей:

(SETQ f ‘((a1 . r1)(a2 . r2)(a3 . r3) …))
(DEFUN f (x) (assoc x f))

В такое точечное определение легко добавлять недостающие пары, соответствующие нужным демонстрационным тестам при макетировании программ для согласования их функций на начальных этапах разработки. Теперь это называется «мемоизация».

Итак, мы получили некоторое число полезных в разных ситуациях схем, различных с точки зрения управления вычислениями:

частичные;
интервальные;
многовариантные;
предопределения;
точечные;
проекции.

Возможны и другие, обеспечивающие оптимизацию, компиляцию, пред-вычисления, макрогенерацию текста программы, что в перспективе может покрыть полное пространство обработки программ в рамках единой методики. Например, основой единого подхода может быть так называемый трансформационный подход, заключающийся в сведении смешанных вычислений к преобразованию программ посредством набора базовых трансформаций. Мета-программирование и близкое к нему трансформационное программирование поддерживают компиляцию и оптимизирующие преобразования. Имеется и поддержка ООП.

9.2. Компилятор или интерпретатор

Система программирования нередко содержит пару «интерпретатор – компилятор». На практике достоинства интерпретации проявляются при отладке программ, а преимущества компиляции – при эксплуатации многократно используемого 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. Компилятор по такой методике получается как остаточная программа при смешанном вычислении интерпретатора.

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

9.3. Стратегические системы

Кроме программ, разработка которых поддержана сравнительно ясными классическими базовыми или основными парадигмами, есть основания выделять стратегические, инструментальные, расширяющие программные системы, нацеленные на подготовку и конструирование компиляторов, интерпретаторов, операционных систем и баз данных, поддержку работы с файлами и разнообразными конфигурациями устройств, а также обеспечение обратной связи при выполнении любых программ. Прежде всего это системы, поддерживающие системное программирование, использующее синтаксически и грамматико-ориентированную обработку определений языка при конструировании системы программирования (VDM, BNF в инструментах типа Lex/YACC и их аналогах во многих новых языках), мета-программирование (Рефал), языки и системы программирования (ЯиСП) и разработки программных инструментов (BLISS, ЯРМО, С), создание промежуточных форм (диалекты) для поддержки особо сложных работ (внутренний язык системы БЕТА), отладки программ и их тестирования.

Такие расширяющие программные системы работают со много более сложными элементами, обладающими своей жизнью, допускающими включение во многие прикладные системы и конфигурации, в которых возможно изменение их состояния. Представления программных систем, кроме сложных структур данных, формальных определений и кодов, включают в себя процессы, устройства, роли участников и комплексы. Методы обработки элементов и их взаимодействия подчинены более жёстким требованиям правильности, что влечёт поддержку улучшения элементов по частям, то есть целенаправленного развития по мере выявления ошибок или необходимости в повышении эффективности. Имеет место разделение труда по уровню квалификации и ответственности.

9.4. Отладка и мемоизация

При разработке больших программ, особенно по нисходящей методике, необходимость в тестировании и отладке возникает нередко раньше, чем подготовлен итоговый текст программы. В таких случаях возникает макет программы — это некоторый предварительный ее текст, допускающий уточнение — доопределение с «заглушками», приспособленное для демонстрации результата в авторских руках. Такой макет особенно интересен на этапе тестирования программ с использование так называемых «мемо-функций».

Представим, что вычисление каждой рекурсивной функции сопровождается сохранением пары "аргумент, результат". После этого можно запустить в дело слегка измененное правило интерпретации функций. Изменение заключается в следующем: прежде чем применять функцию к фактическому аргументу, выполняется проверка, нет ли для этого аргумента уже вычисленного результата. Готовый результат и есть результат функции, а в противном случае все работает как обычно. Механизм сохранения насчитанных результатов функций назван *"мемо-функции"* [33, 201]. Естественно, основанием для его применения является достаточная сложность вычислений и частота обработки. Примечательная особенность данного метода — любая сложность очень частых вычислений стремится со временем к линейной.

Не менее ценные следствия из унификации структурных значений и функциональных объектов дает накопительный, кумулятивный эффект ряда сеансов обработки рекурсивных программ, содержащих общие компоненты. Допустимость совместного хранения функциональных определений и тестов для их проверки в общей структуре, например в списке свойств атома, именующего функцию, позволяет строить *технологические макеты* с множественными определениями, коллекциями тестов и спецификаций, а также с документацией. Такие макеты пригодны для поддержки *полного жизненного цикла программы (ПЖЦП)*. Они позволяют организовывать оперативное сравнение результатов при обновлении системы функций. На такой основе возможно автоматическое тестирование программ. С практической точки зрения технологические макеты — универсальный инструмент динамической оптимизации прикладных систем.

Простейший макет может быть создан из небольшой коллекции тестов, иллюстрирующих поведение программы в наиболее важных точках. Выбор таких точек — необходимая работа, результаты которой многократно используются на всех фазах жизненного цикла программы: при конструировании алгоритмов, *автономном тестировании* компонентов программы, комплексной отладке программы, демонстрации программы всем заинтересованным лицам, при ее эксплуатации и развитии. Функционирование простых макетов особенно легко реализуется в языках, обладающих унификацией структур данных и функциональных объектов, таких как *LISP* и *SETL*. [199, 201] Подобный механизм определения функций с помощью коллекции фактов и ряда шаблонов характерен для декларативных языков программирования – Prolog, ML, Haskell.

SETL — язык сверхвысокого уровня, представляет собой попытку активного использования теоретико-множественных понятий в практике программирования [201].

Согласно концепции этого языка, понятие "функция" обладает двойственной природой. Функция может быть представлена в алгоритмическом стиле — определением процедуры,

выполнение которой сопоставляет результат допустимому аргументу. Но столь же правомерно представление функции в виде графика, отображающего аргументы в результаты. Оба представления могут существовать одновременно — это всего лишь две реализации одной функции. Графическое понимание функции включает в себя и табличную реализацию подобно математическим таблицам Брадиса. Кроме того, график функции не обязан быть линией — это может быть фигура произвольных очертаний. Следовательно, аргументу может соответствовать множество результатов, лежащих на пересечении вертикали с этой фигурой — графиком функции. При такой трактовке нет ничего удивительного в постепенном накоплении или построении графика функции. Можно задать небольшое множество точек графика, а потом постепенно его пополнять. По замыслу Дж.Шварца, автора языка **SETL**, такая методика может выполнять роль оптимизации особо сложных вычислений.

9.5. Понятийная таблица

Мета-программирование использует дополнительную специфику частичных вычислений и усложнения проверки правильности программ**,** привлекает обработку потоков строк (string stream), хэш-таблиц (hash table) и средства супер-компиляции — конструирования компиляторов и компиляторов компиляторов.

Визуальные диаграммы в понятийных таблицах позволяют мгновенно оценить разницу в специфике парадигм, поддержанных мультипарадигмальным языком Clisp, подтверждающую общее мнение о его возможностях программирования на протяжении ПЖЦП, начиная с функционального проектирования, переходящего в рефлексивный компьютерный эксперимент, затем императивную оптимизацию программы, создание декларативной онтологии по реализуемым символам, мета-конструирование обобщённых и настраиваемых макроопределений с переходом к практичному комплекту классов объектов. Таким образом получается нечто вроде визитной карточки ЯиСП Clisp, позволяющей быстро выполнять грубое сравнение с другими языками. В этом плане существенный вклад дают мемоизация, отложенные и опережающие вычисления, мета-программирование, позволяющее строить специализированные проблемно ориентированные DSL-языки программирования.

Таблица 9.1

Парадигмальная шкала средств Мета-программирования в языке Clisp **(ECMS -** **)**

Приоритет	*Подсистема*	*Примечание*
0	D: данные	В качестве значений выступают определения реализуемого ЯП и система команд целевой машины, на которой предстоит выполнять программы, подготовленные на базе данного ЯП
1	E: вычисления	Синтаксически управляемый автомат с переводом строит переход от ЯП к его абстрактному представлению, более удобному для дальнейшей обработки программ и кодогенерации. Константные вычисления и символьные преобразования могут выполняться в этом процессе и заменяться на результат с целью повышения эффективности программ
2	C: управление	Проверка текстов на принадлежность ЯП сопровождается диагнтичесими сообщениями и, возможно, рекомендациями по продолжению разбора текста. Результаты лексического, синтаксического и семантического анализа проверяются на соответствие шаблонам предстоящей кодогенерации. Определение ЯП работает подобно представлению типа данных. Кодогенерация может выполняться автономно. При выполнении созданной программы могут быть дополнительные проверки правильности вычислений.
3	M: память	Заполняется встроенная база данных, формируемая как таблицы, сопоставляющие адреса или идентификаторы отдельным конкретизациям понятий, выделяемых при анализе текста, или их атрибутам. Возможно привлечение промежуточных представлений для решения сложных технических задач. Строится схема программы, которая может быть достаточной для её выполнения или для кодогенерации
4	S: структуры	Используются графовые и кодовые представления понятий, выделяемых при анализе текстов на принадлежность ЯП и выводе атрибутов, полезных для предстоящей кодогенерации и исполнения программы

9.6. Преобразования и многопоточность

Общие решения по обеспечению лаконизма, универсальности, конструктивности и расширяемости приводят к трансформационной семантике ЯП, определяющей семейства допустимых преобразований программ, что является естественным полигоном для мета-программирования [16]. Часть сложностей обучения практическому программированию вытекает из достаточно очевидного сдвига понятий, вызванного тем, что реально не существует возможности использовать ЯП в чистом виде. Фактически используется входной язык системы программирования (ЯиСП), являющийся расширенным подмножеством ЯП, дополненным тем или иным комплектом библиотечных процедур на конкретной аппаратуре. Кроме того, СП подвергают программы разным, не вполне эквивалентным, оптимизирующим преобразованиям, увидеть и осознать эффект которых не всегда удаётся в учебной практике.

Такие примеры для много-поточных программ могут быть связаны с распределением действий по процессорам или с синхронизацией потоков действий [16]. Достаточно простые преобразования сети потоков позволяют варьировать схемы потоков и сводить многие конструкции языка программирования к взаимодействию простых потоков. Обратимость преобразований и чувствительность их результата к информационным связям между фрагментами программы требуют умения формализовать критерии применимости трансформаций и выбора подходящего варианта. Мета-программирование при решении таких задач в основном может использовать макротехнику и символьную обработку, дополненную средствами проверки синтаксической эквивалентности, компнсирующей отсутствие контроля типов данных, противоречащих идее параметров произвольной природы[15, 16].

9.7. Относительные и функциональные БНФ

При определении новых языков на основе уже известных полезен формализм, поддерживающий относительное определение синтаксиса и семантики ЯП, позволяющий чётко выражать разницу между языками при их сравнительном анализе. Формализм для синтаксиса представляет собой расширение БНФ, в котором имена используемых понятий можно сопровождать указанием на близкие понятия из других языков. Так можно указывать части используемого понятия, такие как варианты определения или неявные понятия, с точностью до синтаксической позиции. Такое определение синтаксиса сопровождается представлением семантических систем с указанием перечня языков, использующих эти системы.

Исследование и создание новых ЯП может использовать близкие понятия ряда предшественников. И в том, и в другом случае выделяют конструкции, соответствующие общим понятиям, вариантам синтаксиса и семантики их представления, а также версиям их прагматики, возможно развивающимся в таких процессах [1, 11, 18, 20]. Общими могут быть как полные определения исходных понятий ЯП, так и их части, включая неявные понятия, не имеющие имён в определении, но допускающие указание синтаксической позиции. Синтаксическая позиция может быть задана с помощью терминальной цепочки, символизирующей выбор варианта определения или указывающей на неявное понятие, безымянное определение которого расположено непосредственно вслед за цепочкой. Такие понятия можно выделять в отдельные вспомогательные понятия. При переходе в другой язык определение может расширяться. Известны прецеденты разработки новых ЯП на основе общепризнанных, доказавших свою жизнеспособность. Так, например, языки Val и Sisal созданы на основе языка Pascal, а mPC на основе C. Имеются ряды родственных ЯП, такие как Lisp 1, Lisp 1.5, muLisp, Clisp, CMUCL, Clojure или C, C++, C#, для которых можно точно представлять границы понятий, изменяющихся по мере продвижения по этому ряду. Известны линейки языков или подъязыков, связанных отношением наследования или влияния, допускающие чёткое представления фактических заимствований и уровня новизны или своих решений [13, 25, 27, 28]. В Приложении 4 представлено само-описание предлагаемого формализма, опирающееся на классические БНФ.

Другое развитие БНФ связано с использованием механизмов ФП и математики. Не исключено, что математический аппарат БНФ полезно пополнить использованием операции пересечения множеств и технику конструирования анализаторов обогатить параллельными алгоритмами и функциями. В целом естественно все эти мета-средства рассматривать как элементы языков программирования синтаксических анализаторов и сложных программных систем, включая системы программирования. Ещё одна сложность связана с проблемами речевой практики и неустойчивостью терминологии. В таблицах 9.2–9.7 приведены варианты подбора речевых аналогов для некоторых определений синтаксиса языка C++, приведённых в определении стандарта.

Таблица 9.2. Строка из любых символов за исключением *конца строки* или '>'

Определение (C++)	Речевые аналоги
h-char-sequence: *h-char* *h-char-sequence h-char* ***h-char:*** any member of the source character set except *new-line* and >	***h-строка*** — это *h-литера* **или** строка с *h-литерой* ***h-литера*** - это любой символ кроме *new-line* или '>'
То же самое в виде регулярного выражения	
H-char-sequence: ** \ new-line \ '>'*	***h-строка*** — это любое число символов кроме *new-line* и кроме '>'

Таблица 9.3. Цифры для изображения десятичных чисел

Определения эквивалентны	Речевые аналоги
digit: *one of* 0 1 2 3 4 5 6 7 8 9	**Цифра** — это одна из перечисленных цифр
digit: **{** 0 1 2 3 4 5 6 7 8 9 **}**	**Цифра** — это один из элемент множества

Таблица 9.4. Первая цифра для изображения десятичных чисел

Определения эквивалентны	Речевые аналоги
Nonzero-digit: *one of* 1 2 3 4 5 6 7 8 9	**Ненулевая цифра** — это одна из перечисленных цифр (нет нуля)
Nonzero-digit: *digit* \ '0'	**Ненулевая цифра** — это любая **цифра** кроме нуля

Таблица 9.5. Изображения идентификатора

Определения эквивалентны	Речевые аналоги
id-expression: *unqualified-id* *qualified-id* ***qualified-id:*** *nested-name-specifier template opt* *unqualified-id*	**Представление идентификатора** — это *unqualified-id* ***или*** *qualified-id* ***qualified-id*** — это конструкция из трёх компонентов, последний - *unqualified-id*
id-expression: *unqualified-id* *(nested-name-specifier template opt* *unqualified-id)*	**Представление идентификатора** — это *unqualified-id* ***или*** три компонента, последний - *unqualified-id*
id-expression: *unqualified-id* ***(qualified-id:*** *nested-name-specifier template opt* *unqualified-id)*	**идентификатор** — это *unqualified-id* ***или*** конструкция «***qualified-id***», *состоящая* из трёх компонентов, последний -

	unqualified-id

Таблица 9.6. Перечень без разделителей

Определения эквивалентны	Речевые аналоги
statement-seq: *statement* *statement-seq statement*	**Последовательность операторов** – это оператор ***или*** последовательность операторов с последующим оператором
statement-seq: *statement* **+**	**Последовательность операторов** – это непустая последовательность операторов

Таблица 9.7. Перечень с разделителями

Определение эквивалентны	Речевые аналоги
expression: *assignment-expression* *expression ,* *assignment-expression*	**Выражение** – это присваивание ***или*** выражение, запятая, затем присваивание
expression: *(assignment-expression l',')*	**Выражение** — это перечень присваиваний через запятую
expression: *assignment-expression* *(',' assignment-expression) **	**Выражение** — это присваивание, затем любые число групп (возможно, ни одной) из запятой и присваивания

Такое разнообразие форм привело к развитию методов синтаксического управления обработкой данных, включая создание интегрированных форм определения языков программирования, их диагностических расширений, продолжающих грамматик, смешанных и отложенных вычислений, включая частичную интерпретацию программ и СП, совмещающих преимущества интерпретации и компиляции в одном инструментальном комплексе. В таблицах 9.8–9.9 показаны основные конструкции, используемые при анализе ЯП.

Таблица 9.8.. Демонстрационный пример: текст программы на C для Clang[163].

Пояснение	*Текст программы на языке Си*
Комментарий «Example.c»	// Example.c
Включение библиотеки stdio.h	#include <stdio.h>
Объявление переменной global	int global;
Заголовок функции main	int main (int argc, char *argv[ ]) {
Объявление param со значением 1	int param = 1;
Вызов функции myPrint для печати param	myPrint (param);
Успешный выход из функции main	return 0;
Конец определения функции main	}

Анализатор производит графовое представление, удобное для предстоящего семантического анализа.

Таблица 9.9.. Демонстрационный пример: графовое Clang-представление результата анализа программы на Си

Текст на языке Си	C → AST
// Example.c #include <stdio.h>	← Комментарии не имеют представления ← Макросы не имеют представления

163 http://clang.llvm.org/get_involved.html материалы по Clang — LLVM

int global;
int main (
  int argc,
  char *argv[ ])
{
 	
int param = 1;
 	
myPrint (param);
 	
return 0;
}
	(VarDecl global 'int' **)** ← **AST** for global
	(**FunctionDecl** 'void main (int, char **)') ← **AST** for main ( )
	|-------> (**ParmVarDecl** argc 'int')
	|-------> (**ParmVarDecl** argv 'char**' ; 'char**')
	|-------> **[CompoundStat]**
	|----> **[DeclStat]**
	| |---->(**VarDecl** param 'int')
	| |----->**[| IntegerLiteral** 1 'int' **|]**
	|--->**[| CallExpr** 'void' **|]**
	| |--->**[| ImplicitCastExpr** 'void (*) ()' **|]**
	| | |--->**[| DeclRefExpr** 'myPrint' 'void (*) ()' **|]**
	| |---->**[| ImplicitCastExpr** 'int' **|]**
	| |--->**[| DeclRefExpr** 'param' 'int' **|]**
	|---->**[ReturnStmt]**
	|----**>[| IntegerLiteral** 0 'int' |]

Полученное графовое представление обладает гомоиконностью, представленой в Таблице 9.10 в списочной форме.

Таблица 9.10

Lisp-аналог Clang-представления примера программы на Си

C	AST → Lisp

// Example.c
#include <stdio.h>
int global;
 	
int main (
int argc,
char *argv[ ])
{
int param = 1;
 	
myPrint (param);
 	
return 0;
}
	; Example.c
	(**load** "stdio.h") ; Включение библиотеки
	(**VarDecl** global 'int)
	
	(**FunctionDecl** main (int, char *)
	(**ParmVarDecl** argc 'int)
	(**ParmVarDecl** argv 'char** 'char**)
	(**CompoundStat**
	(**DeclStat** (**VarDecl** param 'int)
	(**IntegerLiteral** 1 'int))
	(**CallExpr** 'void
	(**ImplicitCastExpr** 'void '(*) ())
	(**DeclRefExpr** 'myPrint 'void '(*) ())
	(**ImplicitCastExpr** 'int)
	(**DeclRefExpr** 'param 'int)
	)
	(**ReturnStmt** (**IntegerLiteral** 0 'int))
	)
	)

9.6. Функциональный синтаксис

По мере появления и развития языков программирования были созданы средства формализации их синтаксиса в виде форм Бэкуса-Наура (БНФ), для некоторых классов которых можно сравнительно легко строить автоматы генерации текстов языка и анализа принадлежности произвольного текста языку. Были исследованы вопросы эффективности таких автоматов, сформулированы критерии ассоциирования автоматов с формулами и предложены системы преобразования формул, позволяющие их приводить к нормализованным или каноническим формам, для которых гарантирована оптимальность алгоритмов анализа. Рассматривая понятия БНФ как функции над синтаксическими формулами, можно обратить внимание, что это функции без параметров. Переход к использованию параметров позволит повышать лаконизм и выразительность определений.

Методика определения языка как множества текстов сложилась в математике задолго до появления компьютера. А. Чёрч отмечал полезность метаязыков для лаконичного описания языка и условность границы между синтаксисом языка и его семантикой — такая граница определяется уровнем сложности формул, удобных для понимания. Для практических целей достаточно одного метаязыка, что не исключает использования иерархии метаязыков. Обычно метаязыки обладают само-применимостью, что дает им

лаконизм и чёткость понимания границ применения. Наиболее удобными оказались определения синтаксиса в форме контекстно свободных грамматик. Далее были созданы и исследованы методы нагрузки таких алгоритмов действиями, выполняющими перевод исходного языка в какой-нибудь другой — грамматики с переводом.

Таблица 9.11. Примеры определений разных видов строк по стандарту C++

Формальное определение	*Пояснение*
h-char: any member of the source character set except new-line and >	Любой допустимый символ, кроме конца строки и >
q-char: any member of the source character set except new-line and "	Любой допустимый символ, кроме конца строки и ”
c-char: any member of the source character set except the single-quote ', backslash \, or new-line	Любой допустимый символ, кроме ', \. конца строки
s-char: any member of the source character set except the double-quote ", backslash \, or new-line	Любой допустимый символ, кроме “, \, конца строки
r-char: any member of the source character set except a right parenthesis) followed by a double quote ".	Любой допустимый символ, кроме) “
d-char: any member of the source character set except space, the left parenthesis (, the right parenthesis), the backslash \, and newline.	Любой допустимый символ, кроме пробела. (,). \ иконца строки

Примеры функциональных шаблонов с параметризацией отдельных атрибутов, реализуемых техникой макроподстановки, можно показать на свертке фрагментов синтаксиса C++ с помощью функций F (X) и FF (X, Y). Функция F (X) генерирует правую часть синтаксического определения строки в зависимости от исключаемого символа.

F (X: { >, “, ', \,), (})
=> Любой допустимый символ, кроме {конца строки и X}
=> (_\{Enter | X})* *// Краткий вариант в стиле регулярных выражений*

Функция FF (Y, X) конструирует определение заданного вида строк в зависимости от её ключевого символа и граничного символа.

FF (Y: {H,Q,C,S,R,D}, X: { >, “, ', \,), (})
=> Y-char: Любой допустимый символ,
кроме {конца строки и X}
=> Y-char: (_\{Enter | X})*
// Краткий вариант в стиле регулярных выражений

Таблица 9.12. Примеры макрофункций для конструирования определений разных видов строк по стандарту C++

Задан исключённый символ	*Задан ключевой символ вида строки*	*Результат подстановки*
H-char: **F (>)**	**FF (H, >)**	**H-char:** Любой допустимый символ, кроме {конца строки и >}

Q-char: F (“)	FF (Q, “)	Q-char: Любой допустимый символ, кроме {конца строки и “}
C-char: F (')	FF (C, ')	C-char: Любой допустимый символ, кроме {конца строки и '}
S-char: F (\)	FF (S, \)	S-char: Любой допустимый символ, кроме {конца строки и \}
R-char: F (“)”)	FF (R, “)”)	R-char: Любой допустимый символ, кроме {конца строки и “)”}
D-char: F (“(”)	FF (D“(”)	D-char: Любой допустимый символ, кроме {конца строки и “(”}

Формулы второго столбца таблицы 9.12 допускают более компактную свёртку, используя характерные для ФП отображения и встречающийся в языках параллельных вычислений механизм просачивания, до вида:

FF ((H,Q,C,S,R,D), (>, ”, ', \,), ( ))

Такая формула означает, что в определении языка имеется шесть понятий для строк произвольной длины, имена которых начинаются с символов {**H,Q,C,S,R,D**} и среди символов каждого вида строк исключён соответствующий символ из списка { **>, ”, ', \,), (** }.

9.8. Сравнительный анализ языков программирования

Уже в 1960-ых годах, когда число ЯП исчислялось десятками, Б. Хигман обратил внимание на проблему сравнительного изучения ЯП [20]. На результатах анализа языков от ассемблера до CPL, включая Fortran, Cobol, Lisp, Algol, IPL-V и PL/1, Б. Хигман показал историю развития программистских понятий, связывая её с расширением сферы компьютерных приложений и отмечая отличия программистских понятий от математической и лингвистической терминологии. Он счёл нужным особо рассмотреть средства ввода-вывода данных, необходимые для подготовки и отладки программ. Б.Хигман особо подчёркнул, что целью создания ЯП является будущее.

В 1970-е годы число ЯП исчисляется сотнями, а в 1980-е тысячами. В эти годы появляются книги Т. Пратта и Т. Пратта с М. Зелковиц, в которых рассматривается и упоминается более 30-ти ЯП (C, Java, Perl, Pascal, ML, Fortran, C++, Smalltalk, Prolog, Lisp, ada, Postscript, XML, CGI, LaTeX, html, PL/1, Snobol, APL, Basic, Cobol, CPL, DCPL, assembler, CORC, CUPL, PL/C, Algol-60, Algol W и др.), причём их анализу предпослан обширный обзор эволюции архитектурных возможностей и методов разработки систем программирования [15, 16], включая средства отладки и верификации программ. Материал каждого ЯП привлекается по мере описания прогресса электроники и новых решений при подготовке программ. Получается ряд ЯП, ответивших на такие технологические вызовы — Fortran, Lisp, C, Pascal, Ada, Perl, ML, C++, Prolog, Smalltalk, Postscript, Java. Сравнение учебных ЯП можно рассматривать как упорядочение изучаемых средств по продуктивности программирования [8, 9].

Теперь число ЯП исчисляется десятками тысяч. Появляются новые попытки сравнивать ЯП. Например, Л. Прехельт представил результаты сравнения семи ЯП (C, C++, Java, Perl, Python, Rexx и Tcl) на основе измерения отдельных характеристик производительности программ [17]. В этом плане показательны результаты измерения качество компиляторов, учитывающее около 200 весьма различных параметров [3]. Интересны, а во многом и неожиданны, результаты сравнения языка Lisp с языком C++ и другими ЯП по продуктивности программирования [29-35]. Иногда производительность компиляторов повышают погружением их реализации на аппаратный уровень. [14]. Сравнительный анализ языков Algol и Algol 68 лёг в основу эксперимента по конвертации языка высокого уровня без привлечения промежуточного языка низкого уровня [19].

Таким образом, кроме собственно наследования существуют разные параметры отношений порядка в пространстве ЯП — расширение сферы приложений, эволюция архитектуры, производительность программ, продуктивность программирования, а также различные направления

использования результатов сравнительного анализа ЯП. Существуют и другие критерии упорядочения ЯП по востребованности и результативности [23]. Доступны ресурсы с достаточно обширными данными о современном пространстве ЯП с частичной характеристикой их «анкетных данных», включая отношение наследования без указания параметров наследования и его границ [24-26].

Для навигации в таком пространстве обычно просто сообщается о взаимовлиянии отдельных ЯП и поддержанных в них парадигмах. Например, язык Haskell создан под влияние языков Lisp, Common Lisp, Scheme, ISWIM (Landin), APL, FP (Backus), ML, Standard ML, Hope, Hope+, Clean, Id, Gofer, Sisal, Miranda (Turner), язык Kotlin — под влиянием Groovy, C#, Gosu, Java, Ruby, JavaScript, Scala, Python и ML. Язык Java повлиял на Ada 2005, BeanShell, C#, Chapel, Clojure, ECMAScript, Fantom, Gambas, Groovy, Hack, Haxe, J#, Kotlin, PHP, Python, Scala, Seed7, Vala, JavaScript, JS++, ML на Miranda, Haskell, Cyclone, Nemerle, C++. Уточнение таких сведений более детальными данными о результатах сравнительного анализа по производственно значимым параметрам может служить обоснованием выбора ЯП для программистских проектов, а также для развёртывания работ, использующих разные ЯП.

Для более точной ориентировки в пространстве ЯП нужны средства и методы детального представления сходства и различия конкретных языков, позволяющие понимать их достоинства и недостатки, причины наследования одних механизмов, отказа от других и привлечения новых реализационных решений, что входит в задачу сравнительного исследования ЯП.

Языки программирования обычно допускают представление их определений в виде форм Бэкуса-Наура (БНФ). Часть понятий в ряде языков могут обладать сходством, учёт которого полезен при изучении ЯП и выборе решений при разработке для него системы программирования (СП). Сходство понятий может быть частичным, на уровне вариантов определения или не выделенных в определении неявных понятий. Для решения задач сравнительного исследования ЯП необходимо следующие средства и методы:

- объявление группы близких языков, связанных отношение наследования;
- определение формы для представления наследуемых вариантов общих понятий;
- определение границ частично наследуемых неявных понятий;
- унификация имён общих понятий;
- приведение определений ЯП к форме, удобной для представления результатов анализа;
- инструментарий для проверки корректности представления результатов сравнительного анализа.

Классическое определение БНФ обычно использует ряд обозначений, слегка варьирующихся в разных источниках. Вариант расширенных БНФ представлен а Приложении 4. Самоопределение РБНФ наследует ряд понятий БНФ, дополняет их структурирующими скобками и соответствующими правилами, вводят дополнительные обозначения:

\ — разделитель элемента и знака в перечне, содержащем хотя бы один элемент;
( ) — парные круглые скобки для выделения группы элементов;
[ ] — парные квадратные скобки для выделения факультативной группы элементов;
{ } — парные фигурные скобки для выделения перечня из независимых элементов;
… — повторение произвольного числа предшествующего элемента.

Относительные РБНФ (ОБНФ) для представления результатов сравнительного исследования ЯП обеспечивают разметку определения языка наименованиями предшественников или близких ЯП. Для языков, использующих инородные понятия из других ЯП вводим обозначения, выражающие связь между языками, включая:

КЯП : (ЯП [.Имя] **…**) — конкретный ЯП, наследующий понятия или части определений из перечисленных ЯП.

Имя **:** ЯП [.Имя] — имя понятия, наследуемого из данного ЯП.

Для связывания синтаксиса, семантики и прагматики ЯиСП нужна грамматика с переводом. Собственно представление семантики и прагматики можно свети к набору семантических систем, дополненных перечнем ЯП, включающих их в свою реализацию.

Таблица 9.13
Паспорт мета-программирования на примере БНФ

	БНФ	*Примечание*
Типовая задача	Преобразование программ с сохранением конкретных свойств оригинала.	*Поддержка автоматизации обработки больших и сложных данных.*
Формат семантических систем	**< V, F; R >** — Правила не могут быть значениями	*Накопительные эффекты на уровне программ, рассматриваемых как данные*
Особенности правила применения операций (R)	Анализ или генерация данных с использованием гомоиконности и синтаксического управления	*Либо подстановка варианта, либо проверка данных на соответствие варианту.*
Приоритеты	**MCSE** **M C S E** имена правил; правило перебора вариантов; шаблоны; подстановки.	*Порядок определения и перебора макросов, представление шаблонов для открытой подстановки*
АМ	**secd** — Бесконечные регистры с защитой неизменяемых данных	*Архитектурная модель без типов данных*
АС	Программа представлена как структура данных, допускающая само-обработку при выполнении	*Системная модель без особенностей.*
Диалекты	**PBXD** **P B X D** Применение; Подстановки; Макросы; Подобие;	*Нисходящая методика освоения. Слабость контроля и диагностики неудачного соседства.*

Мета-программирование завершает ознакомление с фундаментальными парадигмами программирования. Можно отметить следующие выводы:

- Мета-программирование неявно присутствует в составе многих ЯиСП.
- Применение мета-программирования требует от программиста повышенного внимания.
- Разные механизмы мета-программирования используются при реализации анализа программ, их оптимизации и кодогенерации в компиляторах.

Практика 9.

1. Измените с помощью препроцессора вид небольшой программы на Си.

2. Сравните эффективность кода программы при использовании техники template и эквивалентных определений препроцессора.

3. Выполните эксперимент на базе языков Clojure, F#, C# или Haskell по применению средств мета-программирования, влияющих на производительность программы, возможность динамического контроля типов данных и редактирования кода на лету.

Алмо, ANTLR, bash bison. BLISS, C++ C#, Сигма C-lang
Clisp, Clojure, Cmucl, ECMAScript, F# Go Golang
GPM, Haskell. Hope, Hope+, Interlisp, JavaScript, Клу Kotlin
LaTeX, LEX Lisp Lisp 1.5, muLisp Open MP Oz. Perl,
PHP, PL/1, Planner, ostscript, Pure Lisp Python (Rebol) Ruby
Racket РЕФАЛ Scheme, Smalltalk, Snobol, SETL.

TRAC, XML

Заключение

Завершена первая часть курса «Парадигма программирования», в котором **парадигма программирования** рассматривается как расширяющийся ряд отдельных парадигм, возникающих по мере необходимости в повышении продуктивности представления решений конкретных трудно решаемых проблем.

Расширение репертуара используемых парадигм — естественная реакция программирования на развитие пространства решаемых задач, совершенствование элементной базы и пополнение сообщества программистов и пользователей ИТ, что подтверждается хронологией появления парадигм, получивших признание программистского корпуса. Истинная миссия программирования — решение новых задач при отсутствии готовых средств и типовых методов. Особо важна роль системного программирования, заключающаяся в изобретении или открытии переходов от необъятного к обозримому или от неизвестного к понятному, что выполняется как разработка систем программирования. Трудоёмкость разработки систем преодолевается сведением определения реализуемого языка программирования к комплекту промежуточных языков, точнее к ряду слоёв, таких как изобразительный, абстрактный, операционный и универсальный. Определение языка программирования на таких слоях поддерживают некоторые системы построения компиляторов.

Каждая парадигма начинается со своей стартовой проблемы и формирует типовую схему постановок задач, решения которых продуктивно программируются на языках программирования, поддерживающих эту парадигму. С каждой парадигмой связаны определённые принципы и особенности накопительных эффектов, обеспечивающих достижение необходимого качества или добротности программ, что можно выражать в терминах семантических систем, используемых при определении языков программирования.

Чёткое различие парадигм как способа мышления связано с упорядочением приоритетов принятия решений в процессе подготовки и отладки программ, особо заметна разница в приоритетах принятия решений по важности разных средств и методов работы с данными и учёта особенностей архитектурных моделей. Для представления таких различий предложена система обозначений, отражающая конкретику абстрактных или виртуальных машин и периферийных устройств.

E — управление процессами вычислений
M — объявление памяти и размещение данных
C — вычислительное устройство
S — структуры данных или средства коммуникации между устройствами и их элементами
In – Средства ввода данных.
Out – Средства вывода данных.
File – Средства хранения данных.
Complex – многопроцессорные конфигурации
Net – сервера и сетевые сервисы
Cloud – облачные хранилища, репозитории

Существует два независимых подхода к определению семантики языков программирования, поддерживающих разные парадигмы программирования — семантика изменения состояний памяти и семантика вычислений, сводящихся к переходу от аргументов функций к их результатам. Оба подхода взаимодействуют в разных правилах применения операций к данным.

RE — обычные арифметические вычисления, отображающие произвольный ряд **значений** аргументов в хотя бы один результат — непосредственное применение функций.
RM — символьные вычисления, использующие **представления** аргументов, возможно без их предварительного вычисления, допуская изменение связи между именами и их определениями — расширение или укрупнение используемых средств и области видимости,

RC — частичные или смешанные вычисления, лавирующие между вычислением и символьной подстановкой в зависимости от разных условий — ограничения, прерывания и диагностика,
RS — обобщённые и параллельные вычисления, оперирующие организацией процессов как множеством потоков над комплексами из разных устройств — продуктивность и производительность.

Высокая сложность определений современных языков программирования приводит к формированию диалектов, соответствующих разным методикам освоения и фазам применения языка и его системы программирования, а также создания проблемно-ориентированных языков.

B — **B**ase — базис, kernel, core — ядро (E),
X — practical e**X**tension — расширения, макросы (M),
D — **D**iagnostics — boundary — границы вычислений, исключения (C),
P — **P**roductivity — univers — всеобщность, внешний мир, устройства (S).

Различаются подходы к классификации парадигм программирования. Чаще всего выделяют базовые, фундаментальные и основные парадигмы. Базовые парадигмы имеют корни в архитектурах, фундаментальные парадигмы ориентированы на проблемы стратегического характера. Для идентификации базовых и фундаментальных парадигм, рассмотренных в данном курсе, достаточно характеристики по отношению к архитектурным моделям.

— MCES — Императивно-процедурное программирование (ИП): Порядок принятия решений начинается с распределения памяти и графа передач управления.

— ESMC — Функциональное программирование (ФП): *Порядок принятия решений* от вычислений к структурам данных, потом имена и минимальное управление.

— **CESM** — Логическое программирование (ЛП): Порядок принятия решений начинается оптимизацией перебора вариантов.

— SMCE — Объектно-ориентированное программирование (ООП): Порядок принятия решений начинается с иерархии классов и размещения объектов в памяти.

— MCSE — Мета-программирование**:** Порядок определения и перебора макросов, представление шаблонов для открытой подстановки.

— ESCM — Параллельные вычисления (много-поточность): Порядок принятия решений от обычных вычислений к обработке триплетов и распараллеливаемым циклам.

Заметные различия можно заметить в методике освоения парадигм программирования. Для функционального, логического и много-поточного программирования типична восходящая методика освоения от элементарных вычислений к сопряжению с внешним миром. Мета-программирование, императивно-процедурное и объектно-ориентированное программирование обычно на практике изучают по нисходящей методике — сверху вниз от средств ввода-вывода и библиотечных процедур к базовым средствам организации вычислений.

— PXDB — Императивно-процедурное программирование (ИП): Нисходящая методика освоения от устройств ввода-вывода и библиотек к средствам языка.

— BXDP — Функциональное программирование (ФП): Восходящая методика освоения, стартовый барьер через обработку атомов и списков с переходом к списочному представлению выражений и функций.

—BDPX — Логическое программирование (ЛП): Восходящая методика освоения, стартовый барьер с вычислений и шаблонов для сопоставления клауз с образцом.

— PDBX — Объектно-ориентированное программирование (ООП): Нисходящая методика освоения, стартовый барьер от ввода-вывода и спецификации наследования

— PBXD — Мета-программирование**:** Нисходящая методика освоения. Слабость контроля и диагностики неудачного соседства.

— BPXD — Параллельные вычисления (много-поточность): Восходящая последовательность освоения, от элементарных вычислений переход к пространствам итерирования, функциям и проекциям.

Кроме рассмотренных базовых и фундаментальных парадигм, во второй части курса будут представлены не вошедшие в первую часть основные, комбинированные, дополнительные, вспомогательные, производные парадигмы и такое явление как мультипарадгмальность.

Приложение 1

От проблем к парадигмам

Годы	Проблема →	→ Парадигма	*Пояснение*/Решение
Машинная арифметика[164]			*Существует самое большое целое.* *Теряется универсальность коммутативности и ассоциативности сложения и умножения.* *При вычислении возникают прерывания разной природы.*
1946	Программисты не имеют опыта представлять программы и данные в памяти компьютера	Императивное программирование (система команд компьютера)	Применяют машинные языки программирования, использующие системы команд компьютера с адресами для передач управления и доступа к машинным словам — ячейкам фиксированного размера. Системы команд и размеры ячеек на разных компьютерах различны.
1947	Потребность в легко обозримом представлении процессов и алгоритмов	Автоматное программирование	Используется методика графического представления алгоритмов в виде блок-схемы, дающей наглядную форму создаваемой программе, Это активно стали применять в 1940-х и 1950-х годах в области программирования и системного анализа, когда началось массовое использование компьютеров. Первые упоминания о блок-схемах относятся к 1920-м годам.
1950	Чрезмерная трудоёмкость отладки программ из-за изменяемости адресов в передачах управления и доступе к данным.	Ассемблирование	Появляются языки ассемблеров, использующих мнемокоды для представления команд, и символьные обозначения для меток и переменных. Программа на ассемблере гомоиконна машинному коду, возможно реассемблирование. Мнемокоды на разных компьютерах могут быть одинаковыми (MOV, GO, SUM, LD и т.п.)
1952	Потребность в укрупнении действий	Операторное и процедурное программирования (AUTOCODE)	Созданы языки низкого уровня, включающие программирование процедур, такие как AUTOCODE и языки с использованием конструкций повышенной квалификации, блоков или модулей. У нас ЭПСИЛОН, СИГМА и другие появились позже.
1954	Необходимость в сопоставлении программируемых решений с математическим аппаратом при демонстрации уровня сложности решаемых задач и методов их решения.	Теоретическое программирование (PureLisp)	Формализация программ. Схемы Лаврова-Ляпунова. Уже существовали математические модели организации вычислений — машина Тьюринга, лямбла-исчисление Чёрча, продукции Поста, цепи Маркова. Они слишком далеки от компьютерных средств для непосредственного применения в программировании. В исследованиях активно применяются базовые подмножества ЯВУ, такие как PureLisp, MinAl, MiniPasfal, PureProlog, Real, C-light, а также языки функционального программирования: ML, Miranda, Scheme, Haskell и другие.
1956	Появилась классификация грамматик Н.Хомского, выделяющая контекстно-свободные грамматики как критерий удобной реализации распознавателей и генераторов текстов, принадлежащих языку.	Синтаксическое управление обработкой данных (БНФ)	Предложены формы Бэкуса-Наура (БНФ) — формализм определения лексики и синтаксиса языка программирования, допускающий приведение к нормализованным формам, с которыми ассоциируется два автомата — анализатор и генератор текстов, принадлежащих заданному языку.
1957	Избыточные затраты на повторную компиляцию отлаженных фрагментов программы.	Модульное (Fortran)	Раздельная компиляция. Библиотеки стандартных процедур. Частичность как критерий эффективности (В.Э.Иткин). Практичность и надёжность сборки программ из готовых модулей. Появляются невычисляемые операторы для объявления компилятору типов данных и размеров векторов. (Fortran)

[164] Бентли Джо Жемчужины программирования / 2002

1957	Потребность в хранении данных разной природы (финансы и бизнес, склады и магазины), а также результаты программ и скомпилированный код.	Системы управления базами данных (СУБД)	Пакетные системы обработки данных (финансы и бизнес, склады и магазины). Произвольный порядок размещения и обработки записей в базах данных. Транзакционная память. Нормализованные формы, гарантирующие качество хранения и доступа (12 н.ф.).
Границы вычислимости[165]			*Символьные вычисления позволяют выйти за пределы машинной арифметики, восстанавливают структуры, строки и числа произвольного размера.*
1958	Затруднения в исследовании новых задач, включая ИИ и системное программирование, в терминах методов вычислений над компьютерной арифметикой.	ФП (IPL, Lisp)	Символьные вычисления выбраны в качестве альтернативы численной обработке для программирования решений новых задач. Провозглашена универсальность понятия «функция», к которому можно свести любые понятия программирования. Изобретен специальный метод работы с неизменяемыми данными (Мусорщик) и динамический контроль хода вычислений. Практикуется самоопределение языка программирования, позволяющее при разработке и раскрутке системы программирования приобрести опыт применения нового языка.
1959	Возникли сложности в обмене и интеграции программного обеспечения, нужного бизнес-аналитикам и специалистам по обработке данных, которым необходимо понимать процессы обработки данных и участвовать в разработке программ для бизнеса.	Декларативное (COBOL)	Предложено использование специальных конструкций для описания данных и их структуры без необходимости детального указания алгоритмов обработки. Структурирование программ на секции, такие как раздел идентификации, среды-контекста, данных и процедур, для четкого разделения различных аспектов программы. Приоритет удобочитаемости программ, стабильности и долгосрочной поддержки бизнес-приложений. Поддержка методов обработки файлов, включая последовательные, индексированные и реляционные базы данных, а также встроенных средств для создания отчетов, что удобно для разработки бизнес-приложений, требующих генерации отчетов и анализа данных.
1960	Заметили, что подобие программ структурам обрабатываемых данных делает программы понятнее, а их отладку продуктивнее.	Гомоиконность	Соответствует принципу хранимой программе в архитектуре фон Неймана , согласно которому программы и данные могут храниться в общей памяти, нет необходимости в специальной раздельной памяти для них. The term first appeared in connection with the TRAC programming language, developed by Calvin Mooers
1960	Мешает отставание эффективности результата компиляции программ от ручного программирования.	Алгоритмическое (ALGOL)	Трансляторы программ по операторным алгоритмам выполняют локальные оптимизации распределения памяти и логики вычислений на основе статического анализа программ.
1960	Неудовлетворённость игнорированием физического параллелизма на уровне устройств.	Параллельное, векторное	Появляется язык APL, автор которого провозглашает, что истинное программирование — это организация параллельных процессов, что соответствует наличию параллеизма на уровне аппаратуры. В этом языке основным данным является вектор и поддержано просачивание операций на произвольную размерность векторов.
1963	Необходимость эксплуатации не вполне надёжного оборудования как в производстве, так и в исследованиях.	Операционные системы Многопроцессорные комплексы	Появляются программы типа мониторов, управляющих прохождением программ, содержащих вызовы независимо отлаженных библиотечных процедур (ИС-2 М.Р. Шура-Буры)
1964	Разбухание текстов про-	Метапрограммирование	Практикуется макротехника над текстами, позволяющая

[165] Нассим Николас Талеб, Чёрный лебедь Под знаком непредсказуемости (2007 г.)

	грамм из-за злоупотреблений шаблонами в производственном программировании.	(GPM)	достигать лаконичных представлений программ с помощью текстовых подстановок (открытые процедуры) в период компиляции. К. Стрейчи, «Макрогенератор общего назначения» , *The Computer Journal* , Vol. 8,1965 г., стр. 225–24. (DOI 10.1093 / comjnl / 8.3.225) - показал, что для полноты по Тьюрингу достаточно возможности именования. https://ru.wikibrief.org/wiki/General-purpose_macro_processor
1964	Появление периферийных устройств, пригодных для непосредственного взаимодействия человека с компьютером.	Интерактивное (JOSS)	Взаимодействие пользователей с компьютером через текстовые команды. Пошаговое выполнение команд, принимаемых непосредственно в диалоге с пользователем, возможно без предварительно подготовленного текста программы. Язык программирования JOSS (Johnnie's Own Scripting System)
1964	Возникла потребность объединить возможностей разных языков и дополнить их средствами работы с устройствами и обработки ошибок	Системное (PL/1)	Появляются всё более сложные языки программирования в стремлении предоставить все возможные средства и предусмотреть потенциально неожиданные ситуации. Провозглашается, что на практике можно ограничиться нужным подмножеством языка, постепенно расширяя его по мере необходимости. В след за PL/1 вскоре появляются Simula 67 и Algol 68.
1965	Необходимость эффективность кода программ совмещать с правильностью	Оптимизационное программирование	Промежуточные языки, глобальная экономия памяти. Альфа-транслятор активно использовался во многих организациях СССР. Генераторы кода для БЭСМ-6 (Альгибр. Альфа-6)
1966	Проблема доверия **правильности реализации** оптимизирующих преобразований.	Доказательное программирование	Техника математических доказательств эквивалентности преобразований программ, корректности оптимизирующих преобразований и анализ критериев и границ их применимости при кодогенерации. понятие «эквивалентность»
1966	Возникают затраты на изучение языков, средств и методов программирования	Ознакомительное и **учебное** программирование (Logo)	Были организованы Альфа-курсы, изучение средств и методов программирования на Альфа-языке. Создаются более простые языки программирования, эффективность реализации существенно выше. Стартовый барьер, изучаемость, руссификация
1966	Решение проблем, связанных с искусственным интеллектом, приводит к поиску математической простоты.	Метапрограммирование-2 (РЕФАЛ)	Предлагается методика сопоставления с образцом и переписывания термов как основного способа определения функций, что позволяет выполнять обработку сложно устроенных текстов, включая тексты программ. **(РЕФАЛ)**
1967	Появилась возможность моделировать поведения сложных систем и симуляции процессов	Процесс-ориентированное (Симула-67)	Появляется предшественник ООП (Симула-67) https://cyberleninka.ru/article/n/razrabotka-graficheskogo-formalizma-dlya-opisaniya-algoritmov-v-protsessorientirovannom-stile
1968	Возникло амбициозное стремление создать предельно универсальный язык для всех областей программирования с настройкой лексики на разные естественные языки	Обобщённое, модульное (Algol68)	Предложена двухуровневая грамматика. Выполнена унификация понятий программирования и достигнута определённая удобочитаемость, строгая семантика и ортогональность составляющих, а также моделирование параллелизма последовательными процессами с семафорами и критическими интервалами. Уделено внимание расширению и адаптации синтаксиса и семантики, (Algol68)
1968	Возросла трудоёмкость программирования на сложных ЯВУ, снизилась	Низкоуровневое программирование, приаппаратное	Создаётся фундамент дальнейших работ по разработке трансляторов, включая языки низкого уровня, пригодные для системного программирования. . Системы ЭПСИЛОН успешно эксплуатировались на

	продуктивность и результативность	Алмо, Эпсилон, Сигма	многих компьютерах, пока они (кроме БЭСМ-6) не были вытеснены отечественными ЕС ЭВМ. Где-то году в 1974-м была предпринята попытка разработать систему ЭПСИЛОН для ЕС ЭВМ. СИГМА — многоуровневая система, базисные операции не имели своей семантики, имелись лишь стандартные процедуры для адресации объектов, распределения памяти и т.д. Востребована до сих пор.
1969	Возникло чрезмерное разнообразие инструментальных средств пропуска программ и обслуживание данных на магнитных носителях	Интерфейсное[166]	UNIX – семейство переносимых, многозадачных и многопользовательских операционных систем, Унифицирован интерфейс работы с процессами, устройствами, файлами, заданиями и инструментарием, введена стандартизация опций и системных переменных. Текстовые скрипты для управления ОС.
1969	Значительная часть исследований искусственного интеллекта направлена на робототехнику, что требует средств и методов, несколько отличающихся от методов вычислений и символьной обработки.	*Робототехника (*Плэнер*)*	Предложен язык для автоматического планирования и диспетчеризации в робототехнике, обладающий адекватным набором выразительных средств для такого рода задач, включая гибкие возможности символьной обработки. Несмотря на то, что язык так и не был реализован в полном объёме, его влияние на исследования, связанные с искусственным интеллектом 1970-х годов, оказалось, во многом, определяющим. Язык содержал фактически целиком Лисп в качестве подмножества, дополненного средствами сопоставления с образцом, поиска с возвратами, вызов процедур по образцу, дедуктивные механизмы в духе логического программирования и параллелизма. (Плэнер)
1969	Взят на вооружение результат теории определения ЯП с помощью формул или синтаксических диаграмм, допускающих приведение к канонической форме, с которой ассоциированы автоматы анализа и генерации текстов языка.	Синтаксически-ориентированное-2	Lex, YACC, ANTLR, bison.
1970	Ненадёжность результатов тестирования программ вызывает поиск математических методов удостоверения или обеспечения свойств программ.	Аксиоматическое	Возникает идея абстрактного типа данных, определение которого включает в себя аксиомы, задающие его поведение с точки зрения *пользователя* данных. Похожие возможности обобщённого программирования впервые появились в виде дженериков (обобщённых функций) в 1970-х годах в языках Клу и Ада, затем — в виде параметрического полиморфизма в ML и его потомках, а затем — во многих объектно-ориентированных языках, таких как C++, Python[1], Java, Object Pascal[2], D, Eiffel, языках для платформы .NET и других.
1970	Чрезмерная трудоёмкость переноса готовых систем программирования на другие компьютеры.	Переносимое ПО	C – самоопределимый язык для программирования переносимых, многозадачных и многопользовательских программ с помощью минимального ядра языка, трудоёмкость реализации которого на новых компьютерах составляет примерно один человеко-месяц.
1970	Чрезмерные затраты на	Императивно-	Разработка методики машинно-зависимого переноса

[166] "Главное, чтобы костюмчик сидел". Песня из фильма "Чародеи" (1982)

	освоение разнообразия сложных ЯВУ и новых архитектур.	процедурное программирование-2 (ИПП) Пи-код , байт-код, Си	компиляторов с помощью определения абстрактной/виртуальной машины, легко реализуемой на любой архитектуре (Пи-код Вирта или подмножество С), концептуально соответствует абстрактной машине (АМ) Венской методике[167] определения языков программирования и secd-машине[168]. Для языков Pascal[169] и С трудоёмкость переноса свелась примерно к одному человека-месяцу.
	Востребована способность программного обеспечения работать более чем с одной аппаратной платформой или операционной системой.	кросс-платформенная	Предлагаются среды разработки и выполнения (**ActionScript Virtual Machine**, **Java Virtual Machine** и **.NET**), поддерживающих условную компиляцию, компоновку и выполнение кода для различных платформ. Созданы стандартизованные **библиотеки среды выполнения**, такие как **библиотека языка Си** (**POSIX**), — **Qt**, **GTK+**, **FLTK**, **STL**, **Boost**, **OpenGL**, **SDL**, **OpenAL**, **OpenCL**. Появились и другие кроссплатформенные интерпретируемые языки **PHP**, **Perl**, **Python**, **Tcl**, **Ruby**, Java, Java Script, Golang и Kotlin, а также ОС (**Linux** и **Windows**).
Преодоление детерминизма[170]			*Проявились задачи, программируемое решение которых допускает недетерминизм в порядке вычислений, недоопределённость и даже невычислимость отдельных значений, мягкость границ точности типов данных.*
1970	Желание решать задачи, постановки которых ещё не созрели для обычного программирования	ЛП (Prolog)	Предложен небольшой набор механизмов, включая сопоставление с образцом, древовидное представление структур данных и автоматического перебора с возвратами (бэктрекинг). Это даёт возможность вместо строгой постановки задачи рассмотреть имеющуюся коллекцию фактов и правил вывода в произвольном порядке, не придавая значения недоопределённости, препятствующей применению обычных методов вычислений. Важно показать перспективу, точность можно повысить по мере интереса к задаче.
1970	Темп появления новых машин и языков программирования превышает возможности их освоения системными программистами.	Многоязыковое и мультипарадигмальное программирования	Складывается практика совмещения типовых механизмов и адаптации систем программирования к разным языкам. Отказ от блок-схем ради снижения трудоёмкости.
1970	Необходимость автоматизации производства СП.	Компиляторы компиляторов	Многоязыковая система построения трансляторов БЕТА сначала вызвала недоверия, но теперь это типовой подход к реализации ЯП. Синтаксическое управление конструированием анализаторов и компиляторов.
1970	Интеллектуальные вызовы со стороны новых архитектур и расширения методов обеспечения качества ПО	Исследовательское программирование	Создание прототипов, имитаторов, эмуляторов, кросс-компиляторов, диалоговых отладчиков, визуализаторов, макетных образцов и других инструментов для опережающего освоения новых решений как в аппаратуре, так и в ЯВУ.

[167] Венская методика – 1968 – Разложение определения языка программирования на уровни конкретного и абстрактного синтаксиса (КС и АС) и абстрактной и конкретной машины (АМ и КМ). Эксперимент такого определения был выполнен на языке PL/1.

[168] 1964 год

[169] Fortran использовался для решения реальных задач, а Pascal представляет собой учебный язык, превосходящий Fortran в скорости изучения и получения как бы работающего решения несложной задачи, если не гнаться за призводительностью. После обретения популярности в университетской среде Pascal лет 10 был производственным языком, но не потеснил Fortran и Си, хотя обладал весьма эффективной реализацией.

[170] Ланиель Канеман Думай медленно — решай быстро. / 2011

1971	Появление микропроцессоров с очень малым объёмом памяти и низкой скоростью счёта, компенсируемых возможностью непосредственной работы, без обслуживающего персонала.	Стек-ориентированное Forth	Появляются языки, предоставляющие семантический минимум, допускающий компактное размещение в памяти и быструю реализацию, достаточную для поддержки основных техник ЯВУ. (Forth) Такой минимум характерен для реализации большинства ЯВУ.
1971	Обнаружение пользы от сосредоточенного описания рассредоточенных действий	Вертикальное слоение = АОП	Идеи вертикального расслоения программ на простые классы семантических систем и оптимизирующей кодогенерации для достижения **предельного баланса** между эффективностью аппаратуры и технологичностью разработки экспериментальных программ. https://www.computer-museum.ru/document/sorucom-2017_33_Mihalkovich_s.pdf
1972	Надежда на выигрыш от математического аппарата в определении схем программ, типов данных, спецификаций, параллельных вычислений и т. п.	Теоретико-множественное	Язык SETL изначально позиционировался как язык сверх высокого уровня (ЯСВУ) — его реализация должны была поддерживать математическую интуицию в выстраивании полноценного пространства допустимых реализационных конструкций, выбор которых должен был автоматизироваться с учётом динамики их применения. Пространство решений: язык SETL (ЯСВУ). Little, Middle, несколько макрогенераторов для решения проблем машинно-ориентированного переноса и адаптации программ и run time library.
1972	Исчерпание резервов производительности программ на однопроцессорных конфигурациях.	Параллельное-2 (Val) экспериментальное программирования	Val, затем Sisal, недетерминированные асинхронные автоматы с понятием «спусковое условие», которое могло быть любой природы. Затем сети Петри, чтобы точно моделировать взаимодействия за исключением императивно одновременного срабатывания.
1973	Систематизация взаимодействий объектов в процессе организации вычислений.	аппликативная	Появляется методика формирования систем программирования «с нуля», без предварительно определённого семантического базиса. Объекты в аппликативном программировании ведут себя как функции, но арность объекта заранее не фиксируется, она проявляется постепенно, во взаимодействиях с другими объектами. Функции и аргументы могут меняться ролями. Объект может применяться сам к себе. **ML** (Meta Language) — семейство строгих языков функционального программирования с развитой параметрически полиморфной системой типов и параметризуемыми модулями.
1974	Осознание кризиса технологий программирования	Структурное (Pascal)	Поиск методики снижения сложности программируемых решений через отказ от умственной нагрузки[171], возникающей при использовании некоторых конструкций императивного программирования в надежде, что более простые математически функциональные аналоги будут продуктивнее[172].
1975	Обнаружение возможности	Верификационное программирование	Правильность программ и свойства процессов

[171] Любимая поговорка американских президентов: *«Он не достаточно умный для простых решений»*

[172] Оказалось, что такой переход не удобен большинству программистов. Методика функционального программирования требует не меньшего напряжения. На матфаках к функциональному анализу приступают после изучения дифференциального и интегрального исчисления.

	автоматизировать обнаружение ошибок подобно проверке доказательств теорем		Обнаружение трудно удостоверяемых качеств программ и возможности обнаружения ошибок в них с помощью автоматов
1976	Потребность в цифровом представлении и идентификации данных и источников информации нечисловой природы.	СУБД-2 представление знаний	Созданы базы данных в разных сферах приложения, многопользовательский режим доступа, транзакции. Тематический классификатор и доступ к данным
1976	Появление матричных процессоров, сетей и микропроцессорных средств	Высокопроизводительное программирование Трансформационое программирование. Супер-программирование РЕФАЛ.	Идея параллельных алгоритмов отступила перед методами распараллеливания. Неудовлетворённость освоением потенциала парадигмы трансформационного программирования показана в докладе А.В. Климова, продвигающего подобную идею на материале языка РЕФАЛ. Корректность схем, процессов, систем и программ распределённые системы.
1977	Компьютеры на бытовом и факультативном уровне попадают в руки детей, возбуждают их любознательность.	Детское (Logo, Робик)	Создаются специальные ЯВУ и методика обучения, учитывающие предстоящий переход обучаемых к производственным ЯВУ. Робик, Рапира ШЮП
1977	*Появляется ряд интересных теорий без практического признания программистами на уровне ЯП, а лишь на уровне отдельных технических приёмов в сценарных языках.*	*Комбина́торное* *Конкатенативное* *Бесточечное*	*Комбина́торное программирование (function-level programming) на принципах комбина́торной логики не требующая явного упоминания аргументов, использует вместо переменных комбинаторы и композиции, что могло быть удобным для создания перемещаемых программ, на зависящих от адресов.* *Конкатенативное использует неявное указание аргументов функций, они определяются как композиция функций, выглядящая как конкатенация.* *Бесточечный стиль похож на представление конвейеров в оболочках UNIX для перенаправления ввода-вывода, не использующее промежуточные переменные, но составляются цепочки из функций.*
Улучшаемость программ[173]			*В процессе отладки программы часто достигается состояние, при котором её дальнейшая правка опасна порождением наведённых ошибок.*
1980	Потребность в улучшении программы без нарушения её целостности	ООП-2 (Smalltalk-80)	Методика представления программ как иерархии классов с поддержкой механизмов наследования, полиморфизма и инкапсуляции. Выделяются события или ситуации с поддержкой стандартных реакций и взаимодействия объектов с помощью сообщений или методов. (Smalltalk-80)
1980	Появление разных конфигураций из нескольких процессоров (SMP, ASMP, NUMA, SIMD, SISD, MIMD, MISD, МКМД), связанных с разными проблемами и ограничениями.	Многопроцессорные комплексы-2 (Эльбрус)	ОС обогащаются средствами взаимодействия процессов и в многопроцессорных комплексах появляются сверхбыстрые каналы передачи данных. (MPI., OpenMP, CUDA и др.)
1980	Возможность	Редакционное	Создаются специальные текстовые процессоры для авто-

173 Google заплатит за улучшение чужих программ https://lenta.ru/news/2013/10/10/bounty/

	автоматизации издательского дела	программирование	матизации работ по издательской деятельности, факто-графии, полиграфии и бюротике. (MultiEdit, Lexocon, Word, OpenOffice, LibreOffice и др.)
1981	Потребность в признании заслуг и способностей	Олимпиадное	Возникают программистские конкурсы и чемпионаты, на которых участники состязаются в умении быстро от-лаживать небольшие программы решения специально подобранных задач. https://www.nsu.ru/n/media/events/sobytiya-obrazovaniya/xxv-otkrytaya-vsesibirskaya-olimpiada-po-programmirovaniyu-im-i-v-pottosina/ https://www.ozon.ru/product/olimpiadnoe-program-mirovanie-laaksonen-antti-elektronnaya-kniga-934530441/? file:///C:/Users/ak/Downloads/sbornik-zadach-po-programmirovaniyu-dlya-predolimpiadnoy-podgotovki.pdf Лааксонен Антти Олимпиадное программирование. ДМК «Пресс», 2019
1983	Перенос издательской идеи лицензирования прав на ПО стал препятствием развития программистского корпуса.	Свободно распространяемое (GNU)	Разработка системы GNU началась 27 сентября 1983 года, когда Ричард Столлман опубликовал объявление о проекте в группах новостей net.unix-wizards и net.usoft. Первой программой GNU стал текстовый редактор Emacs. Официальным ядром операционной системы GNU является GNU Hurd. Система GNU/Linux, более широко известная как просто Linux, получила более ши-рокое распространение.
1984	Прогресс элементной базы спровоцировал массовое владение компьютерами	Учебное программирование	Создаются первые учебники и учебные программные системы и тренажёры для введения школьников в мир программирования - школьная информатика. «Программирование — вторая грамотность» (Е-практи-кум, Алгоритмическая нотация, Робик, Рапира, Школь-ница, КуМир, ПиктоМир и др.)
Параллелизм[174]			*Концептуально основной резерв производительности компьютеров видится в использовании разных моделей параллелизма.*
1986	Компьютерные сети приводят к созданию распределённых баз данных и вычислительных систем, обременённых своим кругом проблем_	сетевое	Появляются ЯВУ, нацеленные на представление есте-ственного параллелизма задач и процессов. (Erlang)
1986		Фрагментное (Luna)	https://steps3d.narod.ru/tutorials/fragment-program-tutorial.html
1986	Потребовали решения вопросы согласования технических заданий, проектов и качества получаемых программ.	Контрактное Eiffel	Термин предложил Бертран Мейер в связи с разработкой языка Eiffel. Контрактное программирование выросло из формальной верификации, формальной спецификации и логики Хоара. Ранее А.П. Ершов использовал термины «программирование для себя» и «программирование на заказ».
1986		ограничения	пролог
1987		Прототипная	**Self** отсутствует понятие класса, а наследование произ-водится путём *клонирования* существующего экземпляра объекта — прототипа.
1987		Компонентная	повторного использования и развёртывания и реализую-щегося в виде множества языковых конструкций (напри-мер, «классов» в объектно-ориентированных языках программирования), объединённых по общему признаку и организованных в соответствии с определёнными пра-

[174] Параллелизм вокруг нас https://parallel.ru/parallelism_around_us

			вилами и ограничениями.
1991	Появились средства связи, достаточные для создания глобальных информационных сетей	Сетевое-2 Интернет	Создание общих сервисных служб и средств, позволяющих всем заинтересованных лицам налаживать и поддерживать безопасный и надёжный обмен данными.
1991	Потребность в сетевой ОС, допускающей любительское применение.	Операционные системы-2 (Linux)	Появившись как решения вокруг созданного в начале 1990-х годов ядра, уже с начала 2000-х годов системы Linux являются основными для суперкомпьютеров и серверов, расширяется применение их для встраиваемых систем и мобильных устройств, некоторое распространение системы получили и для персональных компьютеров
1991	Напряжёнка из-за побочных эффектов. Резко возросло число программистов, не справляющихся с отладкой программ, использующих побочные эффекты присваиваний.[175]	Чистое ФП	Предпринимается попытка ограничить создание новых ЯВУ с помощью пропаганды простого мощного языка чисто функционального программирования, в котором ведущая парадигма — чистое ФП, без присваиваний, а заодно и без устройств, а недостающие парадигмы присоединяются по мере необходимости как монады. В 1999 году был опубликован «The Haskell 98 Report»
1993	Возрастает спрос на статистическую обработку данных и работу с графикой	Статистическое (R)	Предложен язык R с довольно ограниченными средствами описания данных, но с наличием библиотечных средств, которые позволяют загружать в виде таблиц наборы данных во многих формафтах, включая тексты, Excel, CSV, XML и многие другие. В языке R реализованы практически все актуальные средства универсальных статистических вычислений, такие как регрессионный анализ и анализ временных рядов, а также множество алгоритмов для решения задачв отдельных областях. На 2024 год доступно более 20 304 пакета, включая отображение результатов в виде графиков и диаграмм .
1993	Существование особо важных задач	Суперкомпьютерное	Одним из первых суперкомпьютеров считается Cray-1, созданный в 1974 году. Top500 Sisal, Java, C#, Scala, Kotlin, Clojure
Сети и мобильные устройства[176]			*Прогресс элементной базы и телекоммуникаций достиг высокой миниатюризации устройств и скорости информационной обработки, позволяющих использовать компьютерные возможности в любых условиях.*
1995	Необходимость в разных точках зрения на программу, представляемых на разных фазах её существования	Диаграммы	**UML** (*Unified Modeling Language* — унифицированный язык моделирования), дающий примеры визуального представления программы на разных фазах её жизненного цикла.
1995		скриптовое	скриптовое PHP Инструменты для создания персональных веб-страниц набора встроенных средств и дополнительных модулей для разработки веб-приложений Синтаксис PHP подобен синтаксису языка Си. Некоторые элементы, такие как ассоциативные массивы и цикл foreach, заимствованы из Perl. обеспечивающем кроссплатформенность разработанного приложения:
1995	Появляется ниша программирования бытовых электронных устройств и клиентских приложений с	рефлективное Java	Java – ООП с использованием внутреннего байт-кода для машинно-зависимого переноса программ (JVM) допускающий некоторые формы метапрограммирования

175 *Деградация профессионального корпуса из-за резкого возрастания его численности (усреднение интеллекта)*

176 “У меня зазвонил телефон. …” "зачем советскому человеку самолёт?" “Нам электричество сделать всё сумеет ….”

	серверным программным обеспечением		
Мобильные устройства[177]			*Прогресс элементной базы достиг высокой миниатюризации устройств и скорости информационной обработки, позволяющих использовать компьютерные возможности в любых условиях.*
1995	Потребность во встраиваемых ЯВУ доступа к объектам на мобильных устройствам.	Клиент-серверное (javascript)	От попытки перестроить Scheme к потребностям Интернета перешли к созданию нового ЯВУ, предназначенного как для программирования на стороне клиента, так и для программирования на стороне сервера. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности **веб-страницам**. (javascript)
1997		Реактивное	Реактивное программирование — потоки данных и распространение изменений
1997	Массовое появление DSL-языков (предметно-ориентированных ЯВУ)	Диалектное (Rebol)	Переход от накопления правильности в форме библиотечных модулей к созданию диалектов или подъязыков, обладающих правильностью по построению. Rebol - контекстно-зависимый объектный ЯП, созданный специально для распределённых вычислений в Web. Используя любые устройства, соединенные через Интернет. быстрая платформа для Интернет приложений
2000	Возникли распределённые приложения, рассчитанные на исчерпывающую информацию о желаемом ответе сервера.	Архитектурное масштабируемое (Rest)	Rest: взаимодействия клиентов и серверов, архитектурный стиль, распределённые REST-приложения. Это набор правил того, как программисту организовать написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать. Гипермедиа как средство изменения состояния приложения (HATEOAS)
Дизайн и минимизация программирования[178]			
2000	Расширение спроса на пользовательские бизнес-приложения приводит к росту числа разработчиков-любителей вместо квалифицированных разработчиков, особенно из-за задержек, связанных с пандемией и торговых санкций.	**Минимальное** (Low-code)	Предложена технология с возможностью дописывать код вручную, создавать и развертывать пользовательские приложения без обширных знаний в области программирования. Платформы с низким уровнем кода позволяют получить максимальную отдачу от своего решения с низким уровнем кодирования и достичь своих бизнес-целей. В разработке приложения участвуют не только те, у кого есть навыки программирования, но и те, кто не является техническими специалистами, но они поднимаются на уровень опытных разработчиков. https://www.sap.com/central-asia-caucasus/products/technology-platform/build/what-is-low-code-no-code.html
2001		АОП (AspectJ)	Сквозная функциональность приводит к рассредоточенному и запутанному коду, сложному для понимания и сопровождения.
2002	Успех ООП в бизнес-приложениях привёл к решению такую парадигму применить к индустрии производства компиляторов	Много-языковое (.Net)	Проект .Net, а вслед за ним и другие проекты, приступили к разработке технологий реализации разных ЯВУ в рамках единой среды, промежуточного внутреннего языка и общих библиотек. Создан Common Language Runtime (CLR), которая подходит для различных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду

[177] “У меня зазвонил телефон. …” "зачем советскому человеку самолёт?" “Нам электричество сделать всё сумеет ….”

[178] Алан Купер Психбольница в руках пациентов. / Питер, 2022

2005		Событийное	Программирование на основе событий event-driven programming; = автоматное
2006		Облачное	Облачное https://appmaster.io/ru/glossary/oblachnoe-programmirovanie
2007	Появление многоядерных компьютеров требует пересмотра возможностей управления процессами вычислений.	Много-поточное (Go)	Golang - компилируемый много-поточный ЯП, разработанный внутри компании Google. Поддерживает объектно-ориентированный, и функциональный стили.
		непрофессиональное (Basic, Bubble, Glide)	позволяет бизнес-пользователям, которые не являются техническими специалистами, разрабатывать приложения, не написав при этом ни одной строки кода. с помощью интуитивно понятных инструментов перетаскивания, которые снижают или устраняют потребность в традиционных разработчиках, пишущих код. альтернативные методы разработки приложений используют интуитивно понятный графический интерфейс, позволяющий пользователям быстро создавать приложения и автоматизировать бизнес-процессы без необходимости написания кода последовательно. вообще не требуют умения писать код. https://habr.com/ru/articles/746162/ (или Zero-code) Зерокодинг (zero coding от англ. zero — ноль) — подход к созданию различных IT-продуктов с помощью уже готовых шаблонов, инструментов моделирования и графического интерфейса без написания кода. еще в 80-х годах в СССР в рамках создания космического корабля «Буран» был разработан язык «Дракон», в котором реализована концепция «программирования без программирования». Аналогичные языки были созданы и в других странах. Сама же концепция зерокодинга сформировалась в конце нулевых и начале 2010-х годов. В 2014 году сооснователь no-code сервиса bubble.io в своей статье, опубликованной в Huffington Post, призвал развивать программирование, основанное не на языках, понятных компьютеру, а на механизмах, естественных для человеческого мышления. https://blog.skillfactory.ru/glossary/zero-coding/
2011		Микросервисное (REST)	Микросервисная архитектура направленный на взаимодействие насколько это возможно небольших, слабо связанных и легко изменяемых модулей — *микросервисов*, получивший распространение в середине 2010-х годов в связи с развитием практик **гибкой разработки** и **DevOps** **REST**, **JSON**, **Protocol Buffers**, **Thrift**)
2013	Появляются программируемые сети, элементы которых следует формировать в стиле программирования в ограничениях или поддержки истинности некоторых условий	аксиоматическое-2 (**merlin**)	Вопросы управления ресурсами в программируемых сетях выведены на уровень на декларативного языка, включающего логические предикаты для задания ограничений по пропускной способности (constrait programming). Компилятор генерирует код, который может выполняться на сетевых элементах и динамически адаптировать механизмы делегирования контроля так, что изменения состояний памяти, не нарушают глобальные ограничения. Ограничения отличаются тем, что они определяют не последовательность шагов для исполнения, а сохраняемые свойства решения, подобно аксиоматическим системам в абстрактных типах данных.

			https://www.cs.cornell.edu/~jnfoster/papers/merlin.pdf
2019		No-code zero-code	Tilda, Bubble или Mailchimp No-code или, как некоторые говорят, zero-code платформа — это инструмент для создания сайтов, приложений, чат-ботов и других программ без написания кода.
2021	Сохраняется зависимость от системного программирования[179] Желательна возможность отлаживать код, не изучая тонкости языка, с которым программисты могут быть не знакомы.	Поверхностное (D)	Продолжается поиск преемника языка C, который будет легче и проще в программировании, при этом поддержит больше функциональности, надежности, оптимальности, улучшения безопасности и тестирования. Zig (первая стабильная версия в 2021) — язык программирования, нацеленный на системное программирование и обеспечивающий безопасность и предсказуемость.

Часто говорят, что основная задача программиста — это кодирование созданных математиками алгоритмов, выполнение подготовленных менеджерами технических заданий или реализация разработанных аналитиками спецификаций. Череда парадигм программирования, пройденных в ВЦ СОАН СССР под руководством Андрея Петровича Ершова, доказывает, что истинная миссия программирования — **решение новых задач при отсутствии готовых средств и типовых методов**. Развитие современного программирования проходит фазу формирования парадигм параллельного программирования, связанных с проблемами обработки больших массивов данных, организации суперкомпьютерных вычислений и эксплуатации сетевых компьютерно-коммуникационных взаимодействий.

[179] АПЕ: - «Системный программист — самоубийственная профессия. Чем лучше он выполняет свою работу, тем меньше заметен его вклад.»

Приложение 2

Схематичная характеристика систем разработки ЯП (кубы → параллелепипеды → таблицы)

Таблица П2.1

Полный куб понятий ЯиСП

Название слоя			**№ слоя**	**Спецификация семантических систем слоя**		
Универсальный слой 4			**U**	{РП} ↔ {КЯ↔АС↔АМ↔КМ}		
Операционный слой 3		**K**	АМ → КМ			
Изобразительный слой 2		**L**	КЯ ↔ АС			
Абстрактный слой 1	**A**	АС → АМ				
№	**столбца**	**V**	**E**	**M**	**C**	**S**
строки	**Категории функций**	V	V* → V+	T: N → V	(F → {0, 1})*	A ↔ K
B	Вычисления	Элементарные данные	Операции	Память	Управление	Структуры данных
X	Укрупнение конструкций	Составные данные	Определения	Задание	Ветвления Блоки	Иерархия Стек
D	Границы вычислимости	Диагностика Исключения Прерывания	Предикаты Error	Типы переменных	Логика Интерфейсы	Варианты
P	Практичность	Сигналы Сообщения	Библиотеки	Файлы	Устройства	Ввод-вывод

Понятийная сложность создания системы реализации развивающихся ЯиСПП может оцениваться кубом 4*4*4.

Таблица П2.2.

1960 Лисп как язык моделирования других ЯП

Название слоя			**№ слоя**	**Спецификация семантических систем слоя**		
Операционный слой 3		**K**	AM: Lap[180] → KM: IBM 704			
Абстрактный слой 1	**A**	AC: Eval (**списки** . . .) → AM: Lap				
№	**столбца**	**V**	**E**	**M**	**C**	**S**
строки	**Категории функций**	**V**	**V* → V+**	**T: N → V**	**A ↔ K**	(F → {0, 1})*
B	**Символьные вычисления**	**Атомы, Nil**	**Eq, Atom**	*GC, хэш-таблица атомов*	**Cons, Car, Cdr**	Quote, Eval ( ) - скобки
X	**Укрупнение операций**	**Списки**	**Представление функций**	**Lambda Label Prog**	**Ассоциативные списки**	Cond, Nil

[180] Lap – Лисповская ассемблерная программа.

D	**Границы (частичность)**	**Диагностические сообщения**	**Формы, Error**	**Put, Get,** *Apval, Expr/Fexpr, Subr/Fsubr*	**Строки** *Списки свойств, Вектор аргументов*	Предикаты Lazy		
P	Практичность	Числа, Мульти-значения	Defun/Define Мульти-операции	Compile Свойства атомов	Read, Print	REPL Map, Отображения, Конструкторы		

Благодаря самоприменимости и универсальности язык Lisp способен выполнять роль системы построения макетных образцов программ решения новых задач и экспериментального полигона для повышения степени изученности новых задач до уровня практичной версии, реализация которой потом может быть продуктивно выполнена на любом ЯП. Универсальная функция EVAL входит в язык Lisp, поэтому нет необходимости в универсальном слое, его инструменты программируются на языке Lisp по мере необходимости. Списочные формы языка Lisp пригодны для выполнения роли абстрактного синтаксиса, поэтому нет необходимости в изобразительном слое. Таким образом решена проблема систематического, продуктивного расширения пространства вычислимых функций. Для новой задачи достаточно предложить символьные формы данных и опробовать возможные решения с помощью функций. Именно так произошло освоение многих областей компьютерной обработки данных. Понятийная сложность реализации языка Lisp может вместо куба оцениваться параллелограммом 4*4*2, а исследования новых областей компьютерной обработки не превышает абстрактного слоя, точнее, части одной понятийной таблицы — 3*3 без преждевременного решения проблем практичности и управления вычислениями.

Таблица П2.3

1970 Си как инструмент машинно-зависимого переноса программ

Название слоя			**№ слоя**	**Спецификация семантических систем слоя**				
Операционный слой 2			**K**	{C, ассемблер} → { KMi, … }				
Изобразительный слой 2			**L**	<LEX, YACC> (КЯj) → C				
№	**столбца**		**V**	**E**	**M**	**C**	**S**	
строки	**Виды функций**		V	V* → V+	T: N → V	(F → {0, 1})*	A ↔ K	
B	Вычисления		Коды	Арифметика	Адреса	If, Case	Вектора	
X	Укрупнения		Указатели	Определение функции	Типы данных	Вызов функции	Структуры, строки	
D	Границы		Диагностика Исключения	Предикаты	Типы переменных	Циклы	Объединения	
P	Практичность		Файлы	Библиотеки вычислений	Библиотеки над памятью	Main, Препроцессор, сборка кода	Библиотеки ввода-вывода	

При решении проблемы машинно-зависимого переноса ОС и ЯП создан язык Си, трудоёмкость реализации ядра которого на конкретном компьютере примерно один человеко-месяц. Предложены инструменты LEX, YACC, позволяющие быстро решать задачу синтаксического анализа с переводом ЯП в конструкции языка Си. В качестве

промежуточного языка используется объектный код и ассемблерные вставки, позволяющие поддержать полное использование аппаратных средств. Это дало возможность продуктивной разработки Си-образных ЯП на уровне изобразительного слоя, жертвуя разнообразием семантики. Понятийная сложность реализации языка Си может вместо куба оцениваться параллелограммом 4*4*2, сложность реализации других ЯП на его основе сводится к изобразительному слою, т. е. 4*4.

Таблица П2.4

1976 БЕТА — система построения трапнсляторов

Название слоя			**№ слоя**	**Спецификация семантических систем слоя**		
Универсальный слой 4			**U**	{РП} = {КЯ: ДУБ ↔АС: ВЯЗ→АМ: СИГМ →КМ}		
Операционный слой 3		**K**	АМ → КМ: {БЭСМ-6, СМ-4}			
Изобразительный слой 2		**L**	КЯ: {Симула 67, Паскаль, Ада и Модула-2} → АС			
Абстрактный слой 1		**A**	АС: ВЯЗ → АМ: ПЯ			
№	**столбца**	**V**	**E**	**M**	**C**	**S**
строки	**Виды функций**	V	V* → V+	T: N → V	(F → {0, 1})*	A ↔ K
B	Вычисления	Коды	Команды	Память	Управление	Вектора
X	Расширения	Числа	Функции	Идентификаторы	Процедуры	Иерархия + Стек
D	Границы (частичность)	Диагностика Исключения	Предикаты	Типы переменных	Логика	Структуры, строки
P	Практичность	Матрицы	Выражения	Программа	Циклы	Ввод-вывод

При реализации проекта БЕТА был выполнен полный объём по реализационной прагматике ЯВУ, включая на уровне универсального слоя переход от АС к КС в виде специально разработанной текстовой формы ДУБ и переход от варианта АМ: СИГМА (настраиваемый макрогенератор) на произвольную архитектуру, включая многопроцессорные. Изобразительный слой выполнен для четырёх достаточно разных ЯП; операционный слой для двух архитектур. Для абстрактного слоя разработаны два внутренних языка: ВЯЗ. ориентированный на оптимизацию программ ЯВУ, и ПЯ, предназначенный для машинно-ориентированных оптимизаций. Таким образом, для освоения новой КМ достаточно выполнить разработку операционного слоя, а для нового КЯ — изобразительного слоя. И в том, и в другом случае понятийная сложность реализации других ЯП и новаых машин сводится к понятийной таблице, т. е. 4*4, а сложность реализации системы БЕТА может оцениваться кубом 4*4*4.

Таблица П2.5

1995 JVM – абстрактная/виртуальная машина дляиспользования расширенной архитектурной модели

Название слоя		**№ слоя**	**Спецификация семантических систем слоя**
Операционный слой 3	**K**	JVM → {KMi, … }	

Изобразительный слой 2		L	(КЯi, ...} → JVM				
№	столбца	V	E	M	C	S	
строки	Виды функций	V	V* → V+	T: N → V	(F → {0, 1})*	A ↔ K	
B	Вычисления	Примитивы	Операции	Память, ***CG***	Управление	Вектора Стек	
X	Укрупнения	Ссылочные	Функции	Типы переменных	Иерархия ООП	Классы Массивы Перечисления Анонимные классы	
D	Границы	Исключения, предупреждения	Предикаты	дампы памяти, стек вызовов	JMX, профайлеры	Интерфейсы	
P	Практичность	Потоки Файлы Буферы	JIT	Сети, Интернет	многопоточность	Java 2D и графика, Асинхронное I/O, Сетевое I/O	

Появление JVM повышает продуктивность доступа к средствам машинной графики и сетевым ресурсам, тем самым освобождая от подгрузки библиотек. Сборка мусора (***CG***) становится рядовым механизмом. Доступны многопоточность и ссылочные структуры данных. Достаточно на уровне изобразительного слоя для своего КС в качестве АС использовать любой JVM-язык, начиная Java. Понятийная сложность JVM может вместо куба оцениваться параллелограммом 4*4*2, а сложность реализации ЯП на уровне изобразительного слоя — 4*4, не исключена целесообразность абстрактного слоя, тогда понятийная сложность будет 4*4*2.

Таблица П2.6

2002 .Net - много-языковая система реализации систем программирования

Название слоя		№ слоя	Спецификация семантических систем слоя				
Универсальный слой 4		U	РП: {Visual Studio} = {КЯ→АС→АМ→КМ}: FCL				
Операционный слой 3	K	АМ: {CLR} → КМ: {JIT, ...}					
Изобразительный слой 2	L	КЯ:{C#, F#, C++, Visual Basic .NET[181]} → АС: {CLI}					
Абстрактный слой 1	A	АС: {CLI} → АМ: {CLR}					
№	столбца	V	E	M	C	S	
строки	Виды функций	V	V* → V+	T: N → V	(F → {0, 1})*	A ↔ K	

[181] На этапе разработки были реализованы Haskell, Sml, Scheme, Mondrian, Mercury, Perl, Oberon, Component Pascal.

B	Вычисления	Данное	Операции	Память	Управление	Структуры данных
X	Укрупнения	Библиотеки	Определения	Задание	Комплекты	Иерархия Стек
D	Границы	Диагностика Исключения	Предикаты	Типы переменных	Логика	Варианты Интерфейсы
P	Практичность	File	FileInfo, Directory, DirectoryInfo	Асинхронное I/O	BinaryReader, BinaryWriter	System.IO, StreamReader StreamWriter

Появление .Net легализует многоязыковое программирование, поддержанное общим интерфейсом (абстрактный слой), библиотеками и возможностью их пополнение на разных ЯП, наиболее удобных для решения отдельных задач. Понятийная сложность системы .Net может оцениваться кубом 4*4*4, а сложность реализации ЯП сводится к понятийной таблице 4*4, при условии, что CLI пригоден в качестве АС.

Таблица П2.7

2007 Clang-LLVM - инструмент конструирования проблемно ориентированных ЯП

Название слоя		**№ слоя**	**Спецификация семантических систем слоя**			
Операционный слой 3		**K**	LLVM[182] →{ Windows, macOS, Linux, ...}			
Изобразительный слой 2[183]		**L**	ANTLR: { C, C++, Objective-C, Objective-C++, OpenCL C, ... } → AST			
Абстрактный слой 1		**A**	IDE, AST → LLVM			
№ строки	**№ столбца**	**V**	**E**	**M**	**C**	**S**
	Виды функций	V	V* → V+	T: N → V	(F → {0, 1})*	A ↔ K
B	Вычисления	Данные	Операции	Память	Управление	Структуры данных
X	Укрупнения	Библиотеки	Определения	Задание	Модули	Иерархия Стек
D	Границы	Контекстно-ориентированные сообщения	Предикаты Сравнения	Типы переменных	Инструменты отладки	Объединение
P	Практичность	Файлы	CUDA	OpenMP	OpenCL SYCL	GPU, OpenGL RenderScript

Изобразительный слой для реализации ЯП на базе пары Clang-LLVM создаётся в системе «окна-меню» как в текстовом редакторе, гомоиконной классическим БНФ, что делает весьма продуктивной разработку проблемно-ориентированных (DSL) Си-подобных языков программирования. Понятийная сложность системы может вместо куба оцениваться параллелограммом 4*4*3, а реализация DSL-языка — к понятийной таблице 4*4.

Можно видеть, что повышение продуктивности многих средств реализации ЯП связано с повышенным вниманием к изобразительным средствам, что объясняет ограничение работ по сравнению ЯП вопросами синтаксиса и удобочитаемости.

---

182 https://habr.com/ru/articles/591979/ кросс-компиляция

183 https://habr.com/ru/articles/354718/ введение в Clang-LLVM

Разнообразие семантики ограничено возможностями небольшого числа опорных, влиятельных языков, а прагматика сосредоточена на уровне обеспечения практичности и полноты используемых доступных устройств и ресурсов.

Начальные сведения о языке Lisp

Допустимые символы языка Lisp:
– прописные и строчные латинские буквы;
– цифры;
– все специальные знаки, которые есть на клавиатуре.
Выделенные символы: <пробел> . ; ( ) ’ “ #.
Скобки должны быть парными.
Примечание. Под *пробелом* здесь подразумеваются пробельные символы: пробел, перевод строки, табуляция.

Комментарии
Все символы после знака «;» (*точка с запятой*) и до конца строки игнорируются интерпретатором, поэтому там можно писать что угодно (но лучше все-таки осмысленные комментарии к коду).

Атомы — неделимые элементы языка. Атомы могут принимать различные значения.
Атомы должны состоять из *допустимых символов* за вычетом *выделенных символов.*
Выделенные атомы: NIL и () (оба «*пустой список*» и «*ложь*»)? T («*истина*»).
Самоопределимые атомы: числа, строки и *выделенные атомы*. Их значение определяется их внешним видом, другие значения атом принимать не может.

Целые десятичные числа состоят только из цифр, могут быть произвольной длины.
Дробный числа — два целых, разделённые «/»
Вещественные десятичные числа состоят из цифр, в качестве разделителя целой и дробной частей используется точка.
Целые числа с плавающей точкой имеют вид 1.2E+3
Шестнадцатеричные числа — это атомы вида &Hnnnnnnnn, где после обязательного префикса &H следуют 1–8 «цифр» [0..9]+[a..f]+[A..F].

Строка — это последовательность любых *допустимых символов*, заключенная в двойные кавычки.

Точечная пара — это конструкция следующего вида:

(	A	.	B	)
левая скобка	атом или точечная пара	точка	атом или точечная пара	правая скобка
	ноль или более пробелов		один или более пробелов	
	один или более пробелов		ноль или более пробелов	

Примеры: (A . B), (A . (B . C)))

Список — это сокращенная (бесточечная) запись *точечной пары.* По сути, список (A B C) хранится и обрабатывается как точечная пара (A . (B . (C . NIL))).
Следовательно, список может состоять из *атомов, точечных пар* или других *списков*, причем в любых их сочетаниях.
Пример. Список (1 -z (T Nil) “str” (A . B)) состоит из самоопределимого атома 1, атома с именем -z, списка (T Nil), строки “str” и точечной пары (A . B).
Пустой список: ().
Примечание. Список () эквивалентен атому NIL.
Список (NIL) не является пустым, он содержит один элемент.

Голова списка — это первый элемент этого списка, выбирается функцией CAR.
Хвост списка — список, состоящий из всех элементов исходного списка, кроме первого, выбирается функцией CDR.
Примеры:
- головой списка (1 2 3) является атом 1, а хвостом — список (2 3);
- головой списка ((A B) C) является список (A B), а хвостом — список (C);
- хвостом списка (A) является пустой список ().

S-выражение — это *символьное выражение*, то есть атом, список или точечная пара.

Примечание. Любая синтаксически правильная конструкция языка Lisp является S-выражением. Обратное неверно: S-выражение (1 + 2) не является вычислимой конструкцией в языке Lisp.

Начальные сведения о функциях

Основной чертой языка Lisp является единство представления данных и операций — все они представлены S-выражениями (если упрощенно, то списками).
Вызов функции — это список

(<имя_функции> <список_аргументов>)

причем любой аргумент должен быть атомом, списком или результатом применения какой-либо функции.

Квотирование — это блокировка вычисления значения S-выражения, являющегося аргументом функции. Для самоопределяемых атомов квотирование не нужно, так как их значение не вычисляется.
(**quote** <аргумент>) возвращает S-выражение, совпадающее с аргументом.
Важно! Вызов (**quote** <аргумент>) эквивалентен записи с апострофом: ‘<аргумент>
Квотирование принуждает интерпретатор воспринимать список не как вызов функции, а как данные.
Таким образом, квотированный список ‘(1 2 3) — это набор чисел, а неквотированный (1 2 3) — это попытка применить «функцию» с именем 1 к аргументам (2 3). Подобная попытка приводит к ошибке, поскольку 1 — это самоопределимый атом.
Аналогичным образом, неквотированный список (A B C) — это попытка вызова функции A с аргументами (B C). Если ранее функция с таким именем не была определена, также произойдет ошибка.
Квотированный атом рассматривается просто как символ. Отсутствие квотирования (не самоопределимого) атома ведет к попытке вычислить его значение. Если ранее этому атому не было присвоено какое-либо значение, снова получим ошибку.
Квотирование действует на все S-выражение, являющееся его аргументом, поэтому нет необходимости квотировать каждый вложенный список.
Пример: в случае ‘(1 2 (3 4)) попытки вызова «функции» с именем 3 не произойдет.

(**eval** <s-expr>) - вычисление произвольного выражения.

Определение новых **функций** производится при помощи специальной функции (**defun** <atom> <s-expr> <s-expr>), что интерпретируется как

(**defun**<имя_функции> (<список_параметров>)
(<тело_функции>)),

где (<тело_функции>) должно быть S-выражением, имеющим значение.

Некоторые полезные функции

Построение списков и точечных пар
(**cons** <at> <sp>) возвращает список, головой которого стал атом <at>, а хвостом — список <sp>.
(**cons** <at1> <at2>) возвращает точечную пару (<at1> . <at2>)
(**list** <arg1> … <argN>) возвращает список, состоящий из **вычисленных значений** аргументов.

Разбор списков и точечных пар
(**car** <sp/tp>) возвращает элемент, являющийся **головой** исходного списка sp или первый элемент точечной пары tp.
(**cdr** sp/tp>) возвращает список, являющийся хвостом списка sp или второй элемент точечной пары tp.
Замечание. **Существуют также функции caar, cadr, cdar, cddr и т. д., являющиеся комбинациями функций car и cdr и служащие для выделения других элементов списка.**

Проверка на «атомность»
(**atom** <арг>) возвращает T или NIL.

Сравнение атомов и списков
(**eq** <atom1> <atom2>) возвращает T, если значением первого и второго аргумента является один и тот же атом, или NIL в противном случае (включая ситуации, когда значение одного или обоих аргументов не есть атом — даже если они совпадают).
(**neq** <atom1> <atom2>) возвращает NIL, если значением первого и второго аргумента является один и тот же атом, или T в противном случае.
(**null** <арг>) возвращает T, если аргумент имеет значение NIL, и NIL в противном случае.
(**equal** <арг1> <арг2>) сравнивает два S-выражения.

Ветвления
(**cond** (<s-expr><s-expr>) ... (<s-expr><s-expr>)), что трактуется как

(cond (<условие1> <ветвь1>)
(<условие2> <ветвь2>)
…
(<условиеN> <ветвьN>)
)

Обратите внимание

1. Функция cond возвращает значение выполненной ветви.
2. Если не выполнилось ни одно из условий, то функция cond возвращает NIL.
3. Если выполняется более одного условия, то исполнена будет только первая по списку ветвь.

Пример. При x = 1 результатом (cond ((= x 1) 1) (> x 0) 2)) будет 1, хотя выполняются оба условия.
Совет. Если последнюю ветвь снабдить условием T, то она будет исполняться в том случае, когда все остальные условия оказались ложными.

(**if** <s-expr><s-expr><s-expr>), что трактуется как
(if <условие><then-ветвь><else-ветвь>).

Обратите внимание

. <else-ветвь> выполняется, если значение <условие> равно NIL, во всех остальных случаях выполняется <then-ветвь>.
. Функция if возвращает результат выполненной ветви.
. Если <else-ветвь> отсутствует, а <условие> не выполняется, то функция if возвращает NIL.

Примеры рекурсии

Рекурсивный вызов в теле функции
Пример (суммирование элементов списка):

```
(defun sumlist (x) (cond ((null x)0)
                         (T (+(car x)(sumlist(cdr x)) ))
))
```

Результатом вычисления (sumlist ‘(1 2 3 4 5)) будет 15.

Перекрестная рекурсия
Пример (числа Фибоначчи):

```
(defun fib1 (n) (cond (= n 2) 1)
                      (T (+ (fib1 (- n 1))
                            (fib2 (- n 1))))))
(defun fib2 (k) (cond (= k 1) 1)
                      (T (fib1 (- k 1)))))
```

Присваивание значений атомам

(**setq** <атом> <S-выр>) присваивает атому, являющемуся первым аргументом, значение второго аргумента.
(**set** <S-выр> <S-выр>) значение второго аргумента присваивается атому, являющемуся значением первого аргумента.
Значением атома может быть любое S-выражение.
Например:

- в результате исполнения (setq a (atom b)) значением атома a будет T (результат применения функции atom к атому b);
- в результате исполнения (setq a ‘(atom b)) значением атома a будет список ‘(atom b), состоящий из двух элементов: атома atom и атома b.

Сходство и различие функций set и setq

1. (setq a 1) полностью эквивалентно (set ‘a 1).
2. Функции set и setq возвращают значение, присвоенное атому. Таким образом, в результате выполнения (setq a (setq b 10)) и атом a, и атом b получат значение 10.
3. а) после выполнения двух последовательных команд

(setq a b)(**setq** a 1)

атом a получит значение 1, а значение атома b не изменится;

б) напротив, выполнение (setq a b)(**set** a 1) присвоит значение 1 не атому a, а атому b.

Безымянные функции

Лямбда-выражения можно использовать везде, где могут использоваться обычные функции.
Построение безымянной функции (λ-выражения):

(lambda (<список-параметров>) <тело-функции>)

где <список-параметров> и <тело-функции> определяются так же, как для обычной функции (см. п. **«Начальные сведения о функциях»** выше).
Поскольку у безымянной функции нет имени, не получится определить ее в одном месте, а вызвать в другом. Безымянная функция определяется в месте своего вызова:

```
((lambda (<список-параметров>)
      <тело-функции>)
      <список-аргументов>)
```

Функции высших порядков (они же **функционалы**) — это функции, которые используют другие функции в качестве аргументов или результатов.
Применяющие функционалы

(funcall #'<функ> &rest <арг>)
(apply #'<функ> арг1 … аргN &rest <арг>)

применяют функцию <функ> к указанным аргументам. Разница между ними лишь в способе задания аргументов: набором или списком. Специальным знаком, обозначающим «превращение» атома функ в имя функции, является #'.
Пример. Поэлементная обработка списка при помощи различных функций. Сначала определим те функции, которые будут применены. В нашем примере это:
.стандартная функция length (длина линейного списка), не требующая отдельного определения;
.функция инкрементирования 1+:

(defun 1+ (a)(+ a 1))

Теперь можно написать применяющий функционал:

```
(defun FFF (fn x) (cond (x
                  (cons (funcall fn (car x))
                  (func fn (cdr x))))))
```

Результатом вызова (func #'1+ '(1 2 3)) будет (2 3 4).
Результатом вызова (func #'length '((1 2) () (a b c d))) будет (2 0 4).

Отображающие функционалы (для списков)

(mapcar #'<функ> <список1> &rest <списки>)
(maplist #'<функ> <список1> &rest <списки>)

применяют функцию <функ> к car'ам или cdr'ам заданных списков соответственно.
Обратите внимание! Если поданы списки различной длины, то их обработка закончится, когда подойдет к концу самый короткий из них. В более длинных «хвосты» останутся необработанными.
Пример. Результатом выполнения (mapcar #'cons '(a b c d) '(1 2 3)) будет ((a . 1) (b . 2) (c . 3)).
Существует целое семейство аналогичных функционалов: mapc, mapl, mapcon, mapcan. Предоставляем читателю возможность изучить их особенности самостоятельно.

Переопределение стандартных функций

Язык Лисп позволяет определять собственные атомы и функции, используя для этого даже имена стандартных. В каждом конкретном контексте интерпретатор сам определяет, о чем идет речь.
Пример. Результатом выполнения двух последовательных команд

```
(setq list '(1 2 3))
(list 'a 'b list)
```

будет (a b (1 2 3)).
На первом шаге атом list получает значение списка (1 2 3), а на втором вызывается стандартная функция list, третьим аргументом которой становится определенный выше атом list.
Подобные коллизии хотя и не вызывают никаких трудностей у интерпретатора, тем не менее могут послужить причиной недопонимания со стороны человека. *Поэтому мы настоятельно советуем избегать их.*
Аналогичным образом возможно переопределить стандартную функцию.

Пример. Последовательность команд

```
(defun (sum (a b) (+ a b))
(setq cons ‘sum))
```

превратит конструктор cons в суммирование. После этого результатом вызова (funcall cons 10 20) будет не точечная пара (10 . 20), а число 30.

Базис, раскрутка и персональная библиотека

В **базис** языка Lisp и его учебное расширение включаются следующие функции:

- **B E S: cons, car, cdr, atom, eq;**
- **X C M: quote, eval, lambda, defun, cond;**
- D C: if, equal, null, apply, funcall;
- P S M: mapcar, maplist, prog, setq, set;
- P E: простейшие арифметические и логические функции.

Все остальные функции могут быть выведены из этих, даже достаточно выделенных жирным шрифтом.

Персональная библиотека
(**load** “<имя_файла>.lsp”) — загрузка файла с расширением .lsp (из рабочей директории) .

Приложение 4

БНФ как язык программирования ЯП

В данном приложении приведены определения вариантов БНФ, сложившиеся при попытке дать лаконичное описание синтаксиса учебного языка программирования СИНХРОН, направленного на раннее ознакомление с параллелизмом. Определение декомпозировано на ряд диалектов, соответствующих фазам обучения.

П 4.0. Самоописание БНФ

Мета-знак ::= «|» | «::=» | «"» | <**Пусто**>
// Разделитель вариантов, частей правила, ограничителя символов
*// и **Пустая строка***
// Мета-знаки выделены шрифтом arial black
Правило ::= Понятие "::=" Цепочка
// обозначение и его замещающее определение
Понятие ::= **Имя** *// Обозначение нетерминала, имеющего определение*
Цепочка ::= Элемент Цепочка *// конечное число элементов*
| <**Пусто**> *// цепочка может быть пустой, учитывается последней*
Элемент ::= "Символ" | Понятие | Лексема // составляющие определений
Символ ::= Знак | ЗнакСимвол *// несколько знаков без пробелов*
Знак ::= *// любая литера*
Лексема ::= ***число*** | ***Имя*** | Ключ |/7 **строка** *// определяются вне языка*

П 4.1. Описание РБНФ на БНФ (расширенные)

Мета-знак ::= «|» | «::=» | «"» | <**Пусто**> | «\»
| «(» | «)» | «[» | «]» | «{» | «}» | «» | «» |
// Разделитель вариантов, частей правила, ограничителя символов,
// разделитель в перечне, скобки
Правило ::= Понятие "::=" Определение
// обозначение и его замещающее определение
Понятие ::= **Имя** *// Обозначение нетерминала, имеющего определение*
Определение ::= Вариант ["|" (Вариант \ "|")] *// конечное число Вариантов*
| <**Пусто**> *// пустой вариант выбирается последним*
// Любое число вариантов, один должен быть
Вариант ::= "Символ" | Лексема | Понятие | Правило // локально однократное

// простые составляющие определений
| Группа | Факультатив | Перечень | Любой
// составные составляющие определений
Символ ::= Знак | ЗнакСимвол *// несколько знаков без пробелов*
Знак ::= *// любая литера*
Лексема ::= Символ | Ключевое_слово | **число** | **строка**| **Имя**
// имя не входит в число понятий
// оно определяются вне языка
// Символ без кавычек, если это не мешает разбору
Ключевое_слово ::= **ТЕКСТ** *// заглавные буквы*
Группа ::= (Элемент \ " ")
// конечная цепочка элементов через пробелы в скобках
Элемент ::= **Лексема | Понятие** *// составляющие элементов*
Факультатив ::= [Элемент] *// необязательный элемент*
Перечень ::= (_ Элемент \ Лексема)
// конечное число перечисленных элементов, разделённых лексемой
Любой ::= { Лексема \ " " } *// одна из перечисленных через пробел лексем*

П 4.2. Определение ИРБНФ на РБНФ (индексируемые)

Мета-знак ::= «|» | «::=» | «"» | <**Пусто**> | «\» | «**...**» | «(» | «)»
| «[» | «]» | «{» | «}» | «⟶» | «⟷» | «.» | «_»
Понятие ::= **Имя** | ИндексПеременная | ПеременнаяЯП
// Обозначение сквозного понятия, имеющего определения в разных ЯП
ИндексПеременная ::= Имя_Номер | ПеременнаяЯП_Номер
// Имя понятия дополнено номером синтаксической позиции
// для хранения значения, полученного при разборе текста
ПеременнаяЯП ::= Имя**.**Диалект *// имя понятия из другого ЯП*
Вариант ::= | Трансформация *// определены в диалекте Трансформ*
Многократно ::= Группа **…**

П 4.3. Примерная схема трансформаций

Трансформация ::= Макрос *// выглядит как функция*
| Шаблон *// выглядит как структура данных*
| Изменение *// работает как замена на другую форму*
| Преобразование *// правая часть эквивалентна левой*
| Перевод *// правая часть на другом диалекте*

Изменение ::= Шаблон_1 → Шаблон_2 *// переход к новому состоянию*
Перевод ::= Образец_1 ↔ Образец_2 *// в другой диалект*
Образец ::= "Символ" | Понятие | Лексема // составляющие определений
| Группа | Перечень *// нет факультативов и любых*
Фрагмент ::= Символ | Лексема | Слог *// используется в макросах*
Слог ::= Лексема Понятие *// для простой автоматизации разбора*
| Лексема Понятие Лексема

П 4.4. Синтаксис диалекта Мульти

Мета-знак ::= «|» | «::=» | «\» | «(» | «)» | «[» | «]» | «{» | «}» | «_»
Программа ::= [Контекст] [Набор] **? Данное**
// Всегда выводится сообщение о результате
Контекст ::= ((Имя = Данное) \ **;**) **;** *// Обстановка и протоколы*
| (**Доска ::=** Число_1*Число_2) Контекст
// Размеры клетчатой доски
Данное ::= Число | Текст | Клетка | Перечень
// записи или координаты на доске или в специальном регистре
| Изображение | Исполнитель
//Рисунки доступны в общей памяти — регистр M
Клетка *// одноклеточная картинка или спрайт*
Изображение *//Комплект жёстко связанных клеток*
// для прорисовки реквизита или исполнителя
Набор ::= [Имя =] [(Поток \ [,])] *// В любом порядке*
Поток ::= [Имя =] ((Действие \ [**;**])) *// По очереди*
Действие ::= [Имя =] Команда | **@!!** Имя | (Действие)
//вызов исполнителя пополняет список команд
// подобно библиотеке, модулю или пакету
| [ВЕРОЯТНО] **%** Число_1/Число_2 Действие
| ЕСЛИ Условие ТО Действие
// возможен вариант КОГДА — обязательно будет
| КРОМЕ Условие ТО Действие
| ЖДУ Имя *// исполнитель рядом, вызван*
|| ПАУЗА Число *// синхронизующий интервал времени*
| ЦИКЛ Действие_1 [ПОКА Условие] ПОВТОР [Число] Действие_2

// цикл может быть бесконечным, в этом нет ничего плохого,
//это полезно при организации сервисов

Условие ::= Имя | Данное | Имя ИЗ Перечень
// выбранное данное ИЗ перечня исчезает

Команда ::= Имя | ШАГ [**@!!** Имя]
// очередной шаг сценария, возможно другого исполнителя
| { ПУСК СТОП ВВЕРХ ВНИЗ ВПРАВО ВЛЕВО ПАС }
| **[** ВМЕСТЕ (Действие \ [,]])**]** *// в любом порядке*
| **(** ОЧЕРЕДЬ (Действие \ [**;**]) **)** *// последовательно*
| ВОЗВРАТ [**@!!** Имя] *// вернуть шаг исполнителя*

Имя *— лексема, зависит от реализации*

Перечень ::= (Данное **...**)
// перечень произвольного числа данных через пробел

Исполнитель ::= Имя | Кнопка | АПК | Диалект
// абстрактная машина в качестве исполнителя
| Дежурик | Муравей | Художник | Математик | Писатель
// Звенигородский - Робик
| Торговый_Автомат *// по мотивам книги Хоара*
| [Изображение] **(** (Инструкция \ [**,**]) Сценарий **)**
//безымянный исполнитель
// Изображение, система команд и сценарий
| Имя = Исполнитель *//именованный исполнитель*

// Исполнители автономны, общаются только через наполнение доски

Инструкция ::= Команда
// доступна после вызова исполнителя до остановки его сценария

Сценарий ::= **Программа** ***// запускается при вызове исполнителя***
// линейный участок с обходами, допуская самоприменение
// может быть подготолен на другом диалекте

Диалект ::= Тансформ | Асинхр | Синпар | Мульти | АМК

Тансформ *// создание и преобразование исполнителей*
Асинхр *// асинхронные потоки*
Синпар *// синхронизация потоков*
Мульти *// визуализация исполнительных на клетчатой доске*
АПК *// абстрактный многопроцессорный комплекс*

П 4.5. Синтаксис диалекта Трансформ (дополнение Мульти)

// Работает с программами

Мета-знак ::= «|» | «::=» | «\» | «(» | «)»
| «[» | «]» | «{» | «}» | «'» | «.» | «_»
| «⟶» | «↔» | «...»

Диалект ::= Трансформ | Мульти | Асинхр | Синпар | АПК

Тансформ *// создание и преобразование исполнителей и программ*

Мульти // ознакомление с параллелизмом

Асинхр *// асинхронные потоки*

Синпар *// синхронизация потоков*

АПК *// базовая семантика — абстрактная машина*

Программа ::= [Контекст] Метапрограмма **?** **Данное**
// в контексте заданы макросы и трансформации
// Всегда выводится сообщение о результате

Метапрограмма ::= [Контекст.Диалект] [Набор.Диалект]
((Трансформация { Участок }) \ Набор.Диалект)
[Набор.Диалект]

Трансформация ::= Макрос *// выглядит как функция*
| Шаблон *// выглядит как структура данных*
| Изменение *// работает как замена на другую форму*
| Преобразование *// правая часть эквивалентна левой*
| Перевод *// правая часть на другом диалекте*
| Трансформация / Действие
// можно что-то доделать после трансформации

Участок ::= Контекст.Диалог *// входит один раз*
| Набор.Диалог *// преобразуемые части программы*

Изменение ::= Шаблон ⟶ Шаблон'
// один шаг перехода к новому состоянию
// неизменяемая структура данных с изменяемыми значениями

Шаблон ::= (Имя \ Порядок_1) *// в левой части*
// Регистр = адрес или структура в памяти
| (Имя' \ Порядок_2) *// в правой части*
// Тот же самый Регистр с изменённым значением

// Порядки не обязаны совпадать

Преобразование ::= Образец_1 ↔ Образец_2

// обратимая трансформация текста

// при формировании правой части левая копируется в комментарии

Образец ::= (Функция АргТ **...**) | (Фильтр АргТ)

| (Структура АргТ **...**)

// в Образцах могут быть общие имена слева и справа

// слева — именование, справа — подстановка

| Фрагмент

| Определение

| Образец / Условие *// дополнительная защита, страж*

Перевод ::= Определение**.**Диалект_1 ↔ Определение**.**Диалект_2

Фрагмент ::= Символ | Лексема | Слог

Слог ::= Лексема Понятие

| Лексема Понятие Лексема

Знак ::= { пробел + * / - = /= < > **%** ~~ , ; := <=> → ← _ ~~ **@!!** ? ! ОК **:** }

// разновидность лексем

Ключ ::= { МАКРО ОШИБКА Мин Макс Длина Первый Последний ВЕРОЯТНО ЕСЛИ ТО КРОМЕ ЖДУ ПАУЗА ЦИКЛ ПОКА ПОВТОР СООБЩЕНИЕ РЕЗУЛЬТАТ СИГНАЛ ФУНКЦИЯ ПРИОСТАНОВКА ГОТОВЫ ПУСК СТОП ШАГ ВВЕРХ ВНИЗ ВПРАВО ВЛЕВО ВМЕСТЕ ОЧЕРЕДЬ БЛОКИРОВКА ПУСК ВОЗВРАТ УДАЛИТЬ ВСТАВИТЬ }

// разновидность лексем

Понятие ::={ КонтекстД НаборД **// у всех**

ВырД ИмяД ОпределениеД ФункцияД Мемо-функцияД СхемаД ДанноеД ФрагментД МакроПарД ТекстД РисункиД СтруктураД БарьерД ПотокД СвёрткаД ДействиеД КомандаД ЧислоД УсловиеД ФормулаД АргД ОперацияД АдресД ИсполнительД ИндексД } **// в образцах трансформаций**

АргТ ::= ВырТ | «_» *// хоть что*

// текст, выражение или что-то до разделителя

// аргументы вычисляются только явными командами

ВырТ ::= текст *// невычисляемый параметр*

Данное ::= Рисунки | Структура | Фрагмент | Программа | Исполнитель

// кроме изображения реквизита и исполнителей, структуры и
// редактируемые данные из протоколов и сценариев исполнителей

Контекст ::= ((Имя = Определение) \ «;»)
// В обстановке определения, включая протоколы

Определение ::= Функция | Мемо-функция | Схема *//Макрос*
| Исполнитель *// Создание нового*
| Трансформация *// Альтернативы и части*

Схема ::= Макрос *// Шаблон для подстановки или схема*

Макрос ::= ИмяМ = МАКРО (ИмяП …)
Фрагмент (МакроПар \ «,») →Фрагмент′ (МакроПар′ \ «,»)
? «//» ?«Фрагмент (МакроПар \ «_»)»
// вывод исходной формулы в виде комментария

МакроПар ::= Текст1 [~~ Текст2] *// проверка корректности*
| Имя *// символьная константа без кавычек*
//текст, обладающий синтаксическим подобием виду параметра

Действие ::= **Команда** | [ОШИБКА] Данное
//допуск ошибочных ситуаций

Условие ::= Имя | Данное | Имя ~~ Текст *// Проверка на синтаксис*

Команда ::= РЕДАКТИРОВАТЬ **@!!** Имя Данное Адрес
//включая АМП и диалекты
| УДАЛИТЬ **@!!** Имя Данное Адрес
//из протокола исполнителя, включая АМП
| ВСТАВИТЬ **@!!** Имя Данное Адрес *// в протокола исполнителя*

Адрес ::= Число | Имя *// координаты для редактирующих воздействий*

Фрагмент ::= Данное *// не обязательно текст*
//данное, подготовленное в схему или сценарий исполнителя

Исполнитель ::= Редактор | Отладчик | Тестировщик | Оптимизатор
| НовыйИсполнитель

Выр ::= Имя | Данное | (ГенСим Имя) | (Имя ~~ Текст)
| (Функция Арг …) | (Фильтр Арг) | (Структура Арг …)
| (Схема Фрагмент …)

Арг ::= Выр
// выражение, вырабатывающее для функции входное значение

Структура ::= (Выр \ Порядок) | [Выр \ Порядок]
| { Выр \ Порядок }
| [((Индекс **:** Выр) \ Порядок)]
| {[Имя**:**] Выр [***** Целое] \ Порядок}
Порядок ::= { **«;» «,»** }
Индекс ::= Целое
НовыйИсполнитель ::= РОБОТ Имя СистемаКоманд Сценарий
СистемаКоманд ::= [Команда \ **«,»**]
| Редактор | Отладчик | Тестировщик | Оптимизатор

П 4.6. Диалект Асинхр — ФП многопоточных программ (дополнение к Трансформ+Мульти)

Программа ::= [Контекст] [Набор] ? Выр /
// Всегда есть сообщение о вычисленном результате
Определение ::= Функция | Мемо-функция *// Иерархия определений*
Данное ::= @ Программа | @ Данное | @ Функция | @ Мемо-функция
// адресуемые данные
Набор ::= [Имя =] ([Поток \ [,]) Свёртка]]
// свёртка результатов независимых потоков
Поток ::= [Имя =] (Действие) \ ([**;**]) Свёртка]]
// свёртка результатов последовательности действий потока
Свёртка ::= { + * Мин Макс Длина Первый Последний }
Действие ::= СООБЩЕНИЕ (Выр) | Имя = Выр | *Имя1* <=> *Имя2*
| [РЕЗУЛЬТАТ] Выр | СИГНАЛ (Имя) | **!** | **?** Выр
| *(Выр Фильтр) >> структура в СинПар*
| (Имя1 … ИмяК) → (Имя1′ … ИмяК′) | Команда [(Арг …)]
| **@!!** Имя *//вызов исполнителя или диалекта*
// пополняет систему команд и/или набор процессоров
Функция ::= ИмяФ = [ФУНКЦИЯ] (ИмяП …) Поток
Мемо-Функ
// Определение функции, иерархия
Схема ::= ИмяМ = МАКРО (ИмяП …) Фрагмент
// Шаблон для подстановки или схема Трансформ
Условие ::= Имя | Данное | Выр // любой предикат
Выр ::= Формула | Фильтр

Команда ::= Формула
Формула ::= (Арг1 ОперацияИ Арг2) | (Арг ОперацияН)
| (ОперацияК Арг) // списки операций
ОперацияИ ::= { = + * / - = /= < > % ~~ , ; := <=> →} // 1 из
// все инфиксные
ОперацияН ::= { -, ?, !, ОК {Операция, Команда}} // все префиксные
ОперацияК ::= { : _//барьер, @!! //исполнитель}
Фильтр ::= (Арг1 ОперацияФ Арг2) *// все постфиксные*
ОперацияФ ::= {= /= < > ~~ >> →} *//в СИНПАР*
// перенос элементов структуры в другую

П 4.6. Диалект Синпар — параллельное программирование (расширение Аспар+Трансформ+Мульти)

Данное ::= Барьер *// Синхронизаторы*
| @ Программа | @ Данное | @ Функция | @ Мемо-функция
// адресуемые данные
Набор ::= [Имя =] [Барьер] ([Поток \ [,]) Свёртка]]
// Глобальный барьер для синхронизации всех потоков
| [Имя =] ([Поток \ [,]) Свёртка]]
// свёртка результатов независимых потоков
Поток ::= [Имя =] [Барьер] (Действие) \ ([;]) Свёртка]]
// свёртка результатов последовательности действий потока
Действие ::= [Барьер :] Действие *// локальный барьер*
| СООБЩЕНИЕ (Выр) | Имя = Выр | Имя1 <=> Имя2
| [РЕЗУЛЬТАТ] Выр | СИГНАЛ (Имя) | **!** | **?** Выр
| (Выр Фильтр) >> структура
| (Имя1 … ИмяК) → (Имя1′ … ИмяК′) | Команда [(Арг …)]
| **@!!** Имя *//вызов исполнителя или диалекта*
// пополняет систему команд и/или набор процессоров
Условие ::= (ПРИОСТАНОВКА? Барьер) | (ГОТОВЫ Барьер)
// достигнут локальный барьер
// достигнуты все одноимённые локальные барьеры
| Имя | Данное | Выр *// любой предикат*
Команда ::= (БЛОКИРОВКА Барьер) | (ПУСК Барьер)
// если барьер совпадает с глобальным, то приостановка потока
// Глобальный барьер достигнут во всех потоках,

// можно все их активировать
| Формула
Барьер ::= Имя

П 4.7. Интегральная сводка синтаксиса языка СИНХРОН

Программа ::= [Контекст] [Набор] ? Выр /
Контекст::= ((Имя = Определение) \ **;**)
Определение ::= Функция | Мемо-функция | Схема | Данное
Схема ::= ИмяМ = МАКРО (ИмяП …) Фрагмент *(*МакроПар \ «_»*)*
→ Фрагмент′ *(*МакроПар′ \ «_»*) ? «//» ?«*Фрагмент *(*МакроПар \ **«_»***)»*
МакроПар ::= Текст1 ~~ Текст2
Данное ::= Рисунки | Структура | Барьер | Фрагмент | @ Данное
| @ Программа | @ Определение | ОШИБКА Данное
Набор ::= [Имя =] [Барьер] ([Поток \ [,]) Свёртка]]
Барьер ::= Имя
Поток ::= [Имя =] (Действие) \ ([**;**]) Свёртка]]
Свёртка ::= { + * Мин Макс Длина Первый Последний }
Действие ::= [Имя =] Команда | **@!!** Имя *//вызов исполнителя*
| [Барьер :] Действие *// локальный барьер*
| [ВЕРОЯТНО] Число/число Действие
| ЕСЛИ Условие ТО Действие
| КРОМЕ Условие ТО Действие
| ЖДУ (Имя) *// исполнитель рядом*
| ПАУЗА Число *// синхронизующий интервал времени*
| ЦИКЛ Действие1 ПОКА Условие ПОВТОР Действие2
| [ОШИБКА] Данное *//допуск ошибочных ситуаций*
| СООБЩЕНИЕ (Выр) | Имя = Выр | Имя1 <=> Имя2
| [РЕЗУЛЬТАТ] Выр | СИГНАЛ (Имя) | **!** | **?** Выр
| (Выр Фильтр) >> структура
| (Имя1 … ИмяК) → (Имя1′ … ИмяК′) | Команда [(Арг …)]
Функция ::= ИмяФ = [ФУНКЦИЯ] (ИмяП …) Поток
Схема ::= ИмяМ = МАКРО (ИмяП …) Фрагмент
Условие ::= Имя | Данное | Выр *// любой предикат*
| (ПРИОСТАНОВКА? Барьер) | (ГОТОВЫ Барьер)
Исполнитель ::= Имя | Кнопка | Дежурик | Муравей | Художник
| Математик | Писатель | ((Команда \ [,]) Программа)
| Редактор | Отладчик | Тестировщик
Выр ::= Имя | Данное | (Имя ~~ Текст **)** | Формула
| (Функция [Арг …]) | (Фильтр Арг) | (Структура Арг …)

| (Схема [Фрагмент …])

Арг – *вырабатывает для функции подходящее значение*

Формула ::= (Арг1 Операция Арг2) | (Арг Операция)
| (Операция Арг) *// списки операций*

(МИНФ Операция) = (Арг Операция Арг)
(**Операция** : { = + * / - = /= < > **%** ~~ , ; := <=> →})

(МПРЕФ Операция) = (Операция Арг)
(**Операция**: { **-, ?, !**, ОК{Операция, Команда}})

(МПОСТ Операция) = (Арг Операция)
(**Операция** : { **:** _барьер, **@!!** исполнитель})

Формулы = (МИНФ { + * / - = /= < > **%** ~~ , ; := <=> →})
| (МПРЕФ { -, ?, !, ОК}) *// все префиксные*
| (МПОСТ { **:** _барьер, @!! исполнитель})

Фильтр ::= (МИНФ {= /= < > ~~ >> →})

Команда ::= Имя | ПУСК | СТОП | ШАГ **@!!** Имя
| ВВЕРХ | ВНИЗ | ВПРАВО | ВЛЕВО
| ВМЕСТЕ (Действие) \ ([,]) *// одновременно*
| ОЧЕРЕДЬ (Действие) \ ([**;**]) *// последовательно*
| (БЛОКИРОВКА Барьер) | (ПУСК Барьер)
| ВОЗВРАТ **@!!** Имя *// вернуть шаг исполнителя*
| РЕДАКТИРОВАТЬ **@!!** Имя Данное Адрес
| УДАЛИТЬ **@!!** Имя Данное Адрес
| ВСТАВИТЬ **@!!** Имя Данное Адрес

Имя — *//лексема, зависит от реализации*

Адрес ::= Число | Имя *// координаты для редактора*

Фрагмент ::= Данное *// не обязательно текст*

Структура ::= (Выр \ **;**) | (Выр \ **,**) | [Выр \ **,**] | [Выр \ **;**] |{ Выр \ , }
| [(Индекс) **:** (Выр [,])] | [(Индекс) **:** (Выр ;)]
| { [Имя**:**] Выр [***** Целое } *// Кратность*

Индекс ::= Целое | Имя

Приложение 5

Обозначения

АК - абстрактный комплекс
АМ - абстрактная машина
АС - абстрактный синтаксис
БС - базовые средства
ВС - вспомогательная семантика
ИП - императивно-процедурное программирование
ИТ - информационные технологии
КМ - конкретная машина
КП - компонентное программирование
КС - конкретный синтаксис
КЯ - концептуальные языки программирования
ЛП - логическое программирование
ЛШЮП - летняя школа юных программистов
МП - минимальное программирования
ООП - объектно-ориентированное программирование
ОС - операционная система
ПВ - параллельные вычисления
ПЖЦП - полный жизненный цикл программы
ПО - правило применения операций
ПП - парадигма программирования
ПС - прикладные семантические системы
ПЯ - промежуточный язык
РИС - распределённая информационная система
РП — реализационн ая прагматика
СД - структура данных
СП - система программирования
СУБД - система управления базами данных
ТД - тип данных
ТП - технологии программирования
УФ - универсальная семантическая функция языка
ФП - функциональное программирование

ЧФП - чистого функциональное программирование
ЯВУ - язык высокого уровня
ЯиСП - язык и система программирования
ЯНУ - язык низкого уровня
ЯП - язык программирования
ЯСВУ — язык сверх высокого уровня
ЯФП - язык функционального программирования

API - *интерфейс программируемого приложения* (***a****pplication* ***p****rogramming* ***i****nterface)*)
DSL - предметно-ориентированные языки
GC — «сборка мусора» (*garbage collection)*
value-oriented – сведение к значениям
name-oriented - с именованными переменными
pointer-oriented - с прямыми указателями
stack-oriented - с неявными копиями значений в стеке

Список литературы

Основные источники

[Г1] *Городняя Л. В.* Основы функционального программирования. Курс лекций : учеб. пособие для студентов вузов, обучающихся по специальности 351400 "Прикладная информатика". М.: Интернет Ун-т информ. Технологий, 2004. 272 с.
[Г2] *Городняя Л. В.* Парадигмы программирования: анализ и сравнение. Новосибирск: Изд-во СО РАН, 2017. 232 с.
[Г3] *Городняя Л. В.* Функциональное программирование. Парадигма, модели и методы. Новосибирск: Изд-во СО РАН, 2022. 482 с.
[ЛГ] *Лавров С. С., Городняя Л. В.* Функциональное программирование. Принципы реализации языка Лисп // Компьютерные инструменты в образовании. 2002. № 5. С. 49–58.
[ФХ] *Филд А., Харрисон П.* Функциональное программирование / пер. под ред. В. А. Горбатова. М.: Мир, 1993. 638 с.
[Х] *Хендерсон П.* Функциональное программирование. М.: Мир, 1983. 349 с. 22, 43, 152, 164, 177, 194

Он-лайн курсы и материалы к ним

[кА] *Андреева Т. А.* Программирование на языке Pascal. URL:https://intuit.ru/studies/courses/41/41/info
[кБГ] *Березин Н. А., Городняя Л. В.* Введение в программирование на Лиспе.
URL: http://intuit.ru/studies/courses/1026/158/info
[кБД] *Богословский Н. Н., Данилкин Е.А.* Параллельное программирование с использованием OpenMP и MPI. URL: https://mooc.tsu.ru/mooc-openedu/mpi/
[кГ1] *Городняя Л. В.* Курс лекций «Парадигма программирования».

URL: http://iis.nsk.su/files/books/file/FIT-Gor-PP3.pdf
[кГ2] *Городняя Л. В.* Основы функционального программирования.
URL: http://intuit.ru/studies/courses/29/29/info
[кГ3] *Городняя Л .В.* Парадигмы программирования.
URL: http://intuit.ru/studies/courses/1109/204/info

Дополнительные материалы

[дА] *Андреева Т. А.* Программирование на языке Pascal. М.: НОУ «ИНТУИТ», 2016. 277 с.
[дАГ] *Андреева Т.А., Городняя Л.В.* О вопросе измерения вклада программистских решений в производительность программ // Электронные библиотеки. Том 26 № 6 (2023), часть 1. С. 738–759.
URL: https://rdl-journal.ru/article/view/806
[дГ1] *Городняя Л. В.* Неформальное введение в функциональное программирование (образы, аналоги, подобия).
URL: http://iis.nsk.su/files/preprints/ 188_preprint_92.pdf
[дГ2] *Городняя Л. В.* О функциональном программировании // Компьютерные инструменты в образовании, 2021, № 3. С. 57-75. Опубликовано 26.09.2021.
URL: http://cte.eltech.ru/ojs/index.php/kio/article/view/1700/1706
[дГ3] *Городняя Л. В.* Первые реализации языка Lisp в СССР // Виртуальный компьютерный музей. URL http://computer-museum.ru/histsoft/lisp_sorucom_2011.htm
[дЕ] *Ершов А. П.* Предварительные соображения о лексиконе программирования // В сб. «Кибернетика и вычислительная техника», Вып. 1. М., 1985.
URL: http://rkka21.ru/ershov-lexicon.htm
[дИ] *Иткин В. Э.* Алгебра смешанного выполнения программ. // В сб. «Доклады АН СССР». М., 1984. Т. 275:6. С. 1332–1336.

[дЛ1] *Лавров С. С.* Научная автобиография // Компьютерные инструменты в образовании. 2003, № 2. С. 4−12.
[дЛ2] *Лавров С. С.* Так что же такое информатика? // Компьютерные инструменты в образовании. 1999, № 3–4. С. 3–12.
[дЛ3] *Лавров С. С.* Функциональное программирование // Компьютерные инструменты в образовании. 2002, № 2. С. 42−52.
[дЛ4] *Лавров С. С.* Что изучает информатика? // Компьютерные инструменты в образовании. 2000, № 1. С. 13–17.
[дЛС1] *Лавров С. С.*, Силагадзе Г. С. Автоматическая обработка данных. Язык Лисп и его реализация. М.: Наука, 1978. 176 с.
[дЛС2] *Лавров С. С.*, *Силагадзе Г. С.* Входной язык и интерпретатор системы программирования на базе языка ЛИСП для машины БЭСМ-6. М.: ВЦ АН СССР, 1967.
[дЛС3] *Лавров С. С., Силагадзе Г. С.* Входной язык и интерпретатор системы программирования на базе языка ЛИСП для машины БЭСМ-6. М.: ИТМ и ВТ АН СССР, 1969.
[дЛПШ] *Любимский Э.З., Поттосин И.В., Шура-Бура М.Р.* От программирующих программ к системам программирования (российский опыт) // В сб. «Становление новосибирской школы программирования: мозаика воспоминаний». Новосибирск, изд-во ИСИ СО РАН, 2001. С. 17–27. URL: https://www.iis.nsk.su/memories/lyubimskii
[дП] *Покровский С.Б.* Внутренний язык в Бета // В сб. «Становление новосибирской школы программирования: мозаика воспоминаний». Новосибирск, изд-во ИСИ СО РАН, 2001. С. 98–102. URL: https://www.iis.nsk.su/memories/pokrovskii
[дПот] *Поттосин И.В.* История АЛЬФА-проекта // В сб. «Становление новосибирской школы программирования: мозаика воспоминаний». Новосибирск, изд-во ИСИ СО РАН, 2001. С. 46–55. URL: https://www.iis.nsk.su/memories/pottosin_alpha
[дС] *Степанов Г.Г.* История СИГМЫ // В сб. «Становление новосибирской школы программирования: мозаика

воспоминаний». Новосибирск, изд-во ИСИ СО РАН, 2001. С. 80–84. URL: https://www.iis.nsk.su/memories/stepanov_sigma
[дХС] *Хювенен Э., Сеппанен Й.* Мир Лиспа. Т. 1, 2. М.: Наука, 1994.
[дG] *Graham P.* ANSI Common Lisp. Prentice Hall, 1996. 432 p.
[дMcC] *McCarthy John.* Recursive Functions of Symbolic Expressions and Their Computation by Machine. Communications of the ACM, April 1960.
[дMcCAE] *McCarthy J. Abrahams P. W., Edwards D. J. et al.* LISP 1.5 Programming Manual. The MIT Press, Cambridge, 1963. 106 p.
[дL] *Lombardi Lionello A.* LISP as the language for an incremental computer. M.I.T., Cambridge, 1964. 52 p.
[дS] *Stoyan H.* The Influence of the Designer on the Design – J. McCarthy and LISP. Artificial Intelligence and Mathematical Theory of Computation. Academic Press, 1991. P. 409–426.
[дW] *Wegner Peter.* Concepts and paradigms of object-oriented programming. ACM Digital library, 1990. 81 p. URL: https://dl.acm.org/doi/pdf/10.1145/382192.383004

Интернет-ресурсы

[и1] Домашняя страница HomeLisp. URL: http://homelisp.ru/
[и2] Описание работы с языком Clojure.
URL: https://www.tutorialspoint.com/execute_clojure_online.php
[и3] Описание работы с языком F#.
URL: https://replit.com/languages/fsharp
[и4] Описание работы с языком Haskell.
URL: https://replit.com/languages/haskell
[и5] Описание работы с языком SISAL.
URL: http://pco.iis.nsk.su/CPPS/publ/181.pdf
[и6] Платформа JDoodle. URL: https://www.jdoodle.com

[и7] *Файфель Б.Л.* Рабочая версия системы программирования для языка Homelisp. URL: http://catstail.narod.ru/homelisp/index/html

[и8] *Gouy Isaac.* The Computer Language Benchmarks Game “Which programming language is fastest?” URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

[и9] *Sachin Patil.* Lisp as an Alternative to Java. URL: https://psachin.gitlab.io/lisp-java-notes.html

Оглавление

Введение 2.....
Часть 1 Общее представление.....
Лекция 1. Предварительные сведения.....4
1.0. Появление термина.....
1.1. Иллюстрация понятийной сложности.....
1.2. Расширение репертуара парадигм.....
1.3. Подходы к систематизации ЯП.....
1.4. От проблем к парадигмам.....
1.5. Приоритеты приятия решений.....
Выводы.....
Практика 1. Описание ЯП.....19
Лекция 2. Визуальная символика.....20
2.0. Лексика и синтаксис.....
2.1 Семантика и прагматика.....
2.2. Абстрактная машина.....
2.3. Структуры данных.....
2.4. Семантическая система.....
2.5. Архитектурные модели.....
2.7. Пример представления результатов сравнения ЯП.....
2.9. Определитель парадигм.....
Выводы.....
Практика 2. Семантический базис знакомого ЯП или его части
.....59
.....
Список литературы.....
Основные источники.....
Он-лайн курсы и материалы к ним.....
Дополнительные материалы.....
Интернет-ресурсы.....