Лекции с обзором языков программирования

Автор: Ермаков И.Е.,
Технологический институт им. Н.Н. Поликарпова ГУ-УНПК
http://ermakov.net.ru

2011

Предупреждение: в датах в данном тексте возможны погрешности, поскольку материал формировался непосредственно на лекциях, по памяти.

1. Проблема "примитивности" машин

С точки зрения программиста любой компьютер - это процессор + память. + устройства ввода-вывода, но для современного программиста под Windows или Unix они недоступны напрямую, а только через системные вызовы.

В большинстве моделей ЭВМ как команды процессора, так и структура памяти очень примитивны. Память - это просто последовательность байт, а команды выполняют элементарные операции.

```
x := y + z
в машинных командах это выглядит (примерно) так:
mov ax, y
mov bx, z
add ax, bx
mov x, ax
```

Это запись на ассемблере, а реально это будет последовательность двоичных кодов - и вместо x, y, z будут числовые адреса этих переменных в памяти.

Пример машинного кода для реальной программки:

```
VAR
    x, y, z: INTEGER;
BEGIN
  y := 5; z := 10;
  x := y + z;
END
```

```
55 8BEC 57 56 C705040000F005000000 C705000000F00A000000 A1040000F0
0305000000F0 A3080000F0 5E 5F 8E5 5D C3 90
```

В памяти это сплошная последовательность байт. Здесь мы отделили пробелами отдельные команды, как видим, они переменной длины. Это машинные команды архитектуры IA32 (Intel Architecture 32 bit).

На заре ВТ программы составлялись прямо в машинном коде и вводились в память ЭВМ с пульта (перемычками или переключателями).

Затем стали использоваться перфокарты и перфоленты, на которых "набивался" машинный код.

Но программист составлял программу всё равно в машинных кодах.

Потом вместо двоичных кодов стали писать текстовые команды (мнемоники), а потом отдельные люди (кодировщики) переводили это в коды.

Потом возникла идея поручить кодировку машине. На перфокартах кодируются мнемоники (как последовательность кодов символов), а машина сама переводит их в машинные коды. Такую технологию назвали **автокодами, ассемблерами**. Даже возник термин "автоматическое программирование" (из чего видно, что сначала основной работой программиста считалось составление машинного кода).

На самом деле ничего принципиально не поменялось - те же примитивные машинные команды, та же плоская память.

Проблема: пропасть между примитивностью машинных команд и сложностью решаемых задач.

Почему машины "примитивны"? Потому что электронщикам проще и дешевле, и универсальнее делать их такими.
Т.е. ликвидация "пропасти" выполняется уже на стороне ПО.

С самого начала предельная мечта - научить машину понимать человеческий язык. Исключить программирование, как специальную запись алгоритмов.
Во-первых, идея искусственного интелекта провалилась (отложилась на неопределённый срок).
Во-вторых, если подумать, то ясно, что "тупость" и точная исполнительность машины - это благо, а не вред. Например, как мы будем проверять, что машина правильно поняла нашу задачу на естественном языке? Будем с ней обсуждать?
Язык для составления алгоритмов и ПО должен быть строгим, формальным, а **программирование, как деятельность, будет иметь место всегда.** Это одна из форм математики, её продолжение, машинная реализация.
Но: язык должен быть ближе к решаемым задачам, чем к машине.
Так стали появляться **языки программирования**.

2. Появление языков программирования
Исторически первые ЯП:
Plankalkul. Автор - Конрад Цузе (1943 г.) См. на computer-museum.ru.
Plankalkul = "планирование вычислений". Но это только проект языка.

В СССР в 1949 г. начинаются первые в мире курсы программирования на мехмате МГУ. Их ведёт А.А. Ляпунов, без ЭВМ (их ещё нет в МГУ), который прямо "по ходу" изобретает некий язык и принципы "правильного программирования".
Это язык операторных схем, позволяющий удобно описать последовательность вычислений.

Идея в том, чтобы составлять программы на таких языках, а потом "тупо" механически переводить в машинный код.

Уже здесь появляется разделение во мнениях.
"Математическая", научная позиция - нужно использовать языки программирования. Нужно решать задачи, а не "выпендриваться".
"Технарская" позиция - "мы крутые, мы напишем сразу в машинных кодах".

Однако прошло время, и эта "крутость" стала невостребованной. Так происходит всегда: востребованным остаётся умение решать задачи, а знание "тёмных" углов текущей технологии быстро перестаёт быть нужным.
Так изобретение книгопечатания "упразднило" должность переписчиков.
Вывод: знать тонкости технологий нужно, но не нужно считать это основным содержанием своей профессии. Инженер должен отличать вынужденные, временные "уродства" от того, как должно быть "по-хорошему".

3. Развитие технологий трансляции
Трансляция - это автоматический перевод с одного формального языка на другой.
Например, транслятор ассемблера переводит машинные команды из текстового представления в машинный код.
Ассемблеры становились более удобными, например, вместо адресов памяти стало возможно обозначать их именами и потом использовать.

Возникла дерзкая идея построить транслятор с языка высокого уровня.

Впервые это удалось команде Дж. Бэкуса в IBM - язык FORTRAN (FORmula TRANslator), вышел в свет в 1956 г.

Основные сомнения были связаны с эффективностью - никто не верил, что машину можно научить составлять такой же быстродействующий машинный код, как и человека. FORTRAN в случае крупных программ всего на 25-40% проигрывал человеку.

Последствия Фортрана:

1) Программирование стало доступно широкому кругу математиков и инженеров.

2) Стало возможным составление более сложных и больших программ, уменьшилось число ошибок в программах.

Т.е. экономится время программиста.

Машинное время дешевеет, а объём и стоимость ПО увеличиваются. Экономия времени разработки ПО стала "перевешивать" уже в 60-х гг. А сегодня стоимость разработки ПО может в сотни и тысячи раз превышать стоимость оборудования.

3) Написанные программы стало можно использовать на разных ЭВМ.

Реализации Фортрана сильно отличались, но адаптировать программы было не так трудно, как переписывать машинный код.

Разработка языков программирования ориентирована, в общем, на эти три задачи:

1) Сделать программирование более лёгким (доступным для непрофессионалов).

2) Сделать его более надёжным и продуктивным (возможность разрабатывать более крупные системы лучшего качества за меньшие сроки.

3) Сделать программирование независимым от платформы (от оборудования, ОС и т.п.)

1-я и 3-я задачи в основном на сегодня решены. Распространённые языки программирования позволяют создавать кроссплатформенное ПО.

Задача 1 неразрешима более лучшим образом, чем есть сейчас (программировать серьёзные задачи проще, чем "на Паскале", невозможно).

Языки программирования такого класса (независимые от машины, удобные для программиста, пригодные для любых задач) называют "языками третьего поколения" - 3GL (3 Generation Language).

Давно делаются попытки создать инструменты для разработки ПО "без программирования" (в стиле некоторого визуального конструирования). Например, Microsoft Access (или даже Excel). Такие подходы называют 4GL (или даже маркетингово 5GL). Такие инструменты неуниверсальны. Закономерность: те задачи, которые уже изучены хорошо и стали "рутинными", шаблонными, можно реализовать и решать в подходящем 4GL-инструменте. Любая достаточно интересная нешаблонная задача требует полноценного программирования. Поэтому в 4GL-инструметах появляются "языки сценариев" ("макроязыки", "скриптовые" языки), которые относятся к классу 3GL.

4. Алгол

Фортран проектировался спонтанно, это "первый блин комом". Хотя он используется интенсивно до сих пор, в вычислительных приложениях (огромное количество математических библиотек; очень эффективные оптимизирующие компиляторы; инфраструктура для параллельного выполнения Фортран-программ на суперкомпьютерах и кластерах, правда, в этом плане с ним конкурирует Си).

В конце 50-х возникает идея спроектировать язык программирования специально, чтобы он был:

1) Полностью независимым от машины.

2) Пригодным для "безмашинных" задач - обмен алгоритмами, публикации в журналах, учебная литература и т.п. Т.е. язык, удобный для человека, но при этом допускающий эффективную компиляцию в машинный код.

3) Строго описанным. Для этого была предложена БНФ-нотация (контекстно-

зависимые грамматики).

Все эти пункты стали отражением расширения задач программирования - уже нужен обмен программами, уже возникает Computing Science как академическая дисциплина.
Третий пункт должен был снять проблему, характерную для Фортрана - несовместимости разных компиляторов. Программа, написанная на Алголе, должна правильно компилироваться и одинаково работать на любом компиляторе на любой машине.

Алгол - европейский язык, разрабатывался группой учёных. Был создан комитет для стандартизации языка. Его возглавлял Питер Наура.

В СССР с 1951 г. в Новосибирске был создан Отдел программирования, который возглавил А.П. Ершов. Исходя из тех же соображений, что авторы Алгола, они проектировали "Сибирский язык". Прочитав публикации по Алголу, видят, что это практически то же самое. "Синхронизируются" и делают язык Альфа - Алгол с русской лексикой, и транслятор с него.

Основная стандартизированная версия Алгола - Алгол-60.

Та же проблема, что при введении ассемблеров - разделение на сторонников "строгого", научного подхода и "технарей".
"Технари" - за Фортран. Критикуют идею независимости от машины, идею "безмашинного" составления программ.
В США Алгол практически не распространился, распространился в Европе и СССР.

Все последующие императивные языки уже строились "от Алгола". Использовались идеи, впервые введённые в Алголе: объявления переменных, блоки begin-end, подпрограммы.

Алгол-68: собрался большой комитет, стали предлагать включить в язык "много хороших вещей". В результате получился очень большой и сложный для реализации язык, а все включенные вещи - спорные, экспериментальные.
В результате язык имел единичные реализации. В частности, реализация группы проф. Терехова в Ленинграде (сейчас это компания "Ланит-Терком", их Алгол используется в мире в сфере программ для телефонных станций).
Выводы:
1) Не всё, что можно придумать, стоит реализовывать (в программировании нет физических ограничений, поэтому можно придумать любую хрень).
2) Избыточная сложность ведёт к проблемам в реализации.
3) Добавлять в язык много разных средств невыгодно: разработка компилятора дело дорогое, что-то поменять потом, когда языком уже пользуются, трудно или вообще невозможно. Если идти путём "максимум возможностей в языке", то нет чёткой границы, где остановиться (почему это включили, а то - нет).

5. Паскаль
Из этого печального примера выводы сделал Никлаус Вирт, который в 1970 г. создал язык Паскаль и компилятор для него. Это очищенный и компактный потомок Алгола, предназначенный для обучения студентов.
1) "Язык для обучения" - это не значит "непрофессиональный язык". Наоборот: Вирт включает в него то, что является самым главным на тот момент, лучшее из придуманного в области программирования, при этом выбрасывает всякие случайные особенности и дефекты "промышленных" языков.
Идея в том, что обучать нужно на инструментах, свободных от дефектов, а затем освоение промышленных инструментов сводится к изучению их "тёмных углов" по сравнению с эталонным инструментом.
2) Вирт, хотя и профессор, но не теоретик, а инженер - до Паскаля разработал 3 языка (вариант Фортрана, язык Euler и Algol-W), входил в комитет Алгола, по

образованию - электронщик.

Вирт придумал промежуточный "псевдомашинный" код (P-код) и компилятор из Паскаля в него. Далее на любой платформе можно было легко написать интерпретатор P-кода. А затем уже при необходимости получить полноценный компилятор в машинный код. (P-код позднее был взят за основу байт-кода Java).

Паскаль - один из первых языков, компилятор которого написан сам на себе (принцип "раскрутки" - bootstraping).

Причины эффективности раскрутки: 1) технологическая независимость вашей разработки от других; 2) язык испытывается "в деле" как можно раньше.

Паскаль распространился во всём мире (кроме СССР, где прижился Алгол, и не видели причин для перехода. Паскаль популяризовался у нас уже на ПК, в реализации Borland).

С 1980-х в США - Turbo Pascal (компания Borland, Андрэс Хейлсберг). Первая в мире интерактивная среда разработки (IDE), доступная массовому пользователю под ПК и DOS. Библиотека оконных приложений (в текстовом режиме экрана) Turbo Vision.

Как раз к 70-м годам возникает идея писать системное ПО (например, ОС) на языках высокого уровня. Для этого начинают эффективно применяться и Алгол, и Паскаль.

6. Основные качества языков высокого уровня

К 70-м годам ПО стало очень большим, поэтому главным для хороших языков программирования стали считаться средства для борьбы со сложностью, т.е. те средства языка, которые позволяют хорошо организовать большую программу, предотвратить многие ошибки, легко изменять программу.

Дейкстра и его единомышленники (Хоар, Вирт) предлагают идею **структурного программирования**. Она заключается в том, чтобы ограничить конструкции хода выполнения программы тремя:

1) Последовательное выполнение (команды через ;)

2) Развилка (IF ELSE)

3) Цикл с условием в начале (WHILE). Как варианты, допускаются цикл с условием в конце (REPEAT - UNTIL) и цикл со счётчиком (FOR), но без них можно обойтись.

Было доказано, что любую программу можно записать только с помощью этих конструкций.

До структурного программирования широко использовался оператор произвольного перехода: GOTO номер строки. Это приводило к очень запутанным программам, так называемой "тарелке спагетти".

Кроме того, пропагандируется широкое использование подпрограмм (процедур, функций).

Структурное программирование подверглось резкой критике от "технарей" ("GOTO удобен", "программист имеет право писать, как хочет", "вызов подпрограммы - это лишние операции процессора").

Однако без структурного программирования создание крупных систем давно стало невозможным.

Дейкстра создаёт теорию строгого математического вывода циклов (на основе логических утверждений). Цель - сделать построение отдельных фрагментов программ обоснованным, доказуемым (как расчёт прочности в строительстве). Книга "Дисциплина программирования".

Структурное программирование полностью победило. Метод доказательного программирования Дейкстры в массы пока не вошёл (проблемы с образованием).

Отсюда во всех массовых языках сегодня есть дефект с точки зрения структурного программирования: присутствует оператор досрочного прерывания цикл (break). Это

ограниченный вариант goto, его применяют те, кто не умеет правильно составлять циклы.

В общем, законы такие:
1) Человек всегда ошибается. Ошибки нужно надёжно обнаруживать. При этом чем раньше найдена ошибка, тем это дешевле. В идеале компилятор должен находить всё, что он может (он может не всё, но многое).
Отсюда принцип статической строгой типизации - явное объявление переменных, их типов, строгий контроль совместимости типов и т.п.
2) Программы больше читаются, чем пишутся. Поэтому синтаксис языка должен облегчать восприятие программы, а не её написание. Особенно касается чужого кода, но свой код через два месяца уже как чужой.
3) Важнейшая проблема - контроль качества. Как можно убедиться и убедить других, что программа надёжна? Тестирование помогает найти ошибки, но не может доказать их отсутствие (в цифровых системах огромное число возможных вариантов, их невозможно покрыть никакими тестовыми наборами; хотя такие наборы тоже важны, они составляются для основных сценариев выполнения).
Отсюда вывод - программа должна разрабатываться так, чтобы она допускала проверку, верификацию. Проверка может быть формальной (математическое доказательство), но большую программу (больше 10 тыс. строк) полностью доказать не удастся. Поэтому используется полуформальная проверка - сложные фрагменты доказываются, остальные верифицируются обзорами кода, которые выполняют несколько специалистов.
Вычитывание кода ("аудит кода"), особенно перекрёстный (несколькими людьми) очень выгоден: тестирование за 1 час выявляет 1-2 ошибки, вычитывание - 3-4.

В 70-х гг. эти принципы стали проецироваться с уровня языка на уровень оборудования. Стали разрабатываться процессоры с системой команд, ориентированной исключительно на языки высокого уровня.
Машины Burroughs (США) - вместо машинного языка Алгол-60.
Машины "Эльбрус" (СССР, 70-е) - вместо машинного языка язык Эль-76 (по функционалу похож на C# и .NET). Сегодня это Эльбрус-3М (см. http://mcst.ru/), но вместо Эль-76 там уже C++.
ПК Lilith, Ceres Никлауса Вирта (машинный язык есть, но он оптимизирован под трансляцию с языка высокого уровня). Аналог Ceres - новосибирский "Кронос" (1989).
Также рабочие станции "Самсон" (ЗАО "Ланит-Терком"), с конца 80-х работающие в правительственной связи. Ориентированы на Алгол и подобные языки.
IBM AS/400.

Преимущества:
1) Система машинных команд упрощается.
2) Программы работают более эффективно (проще компилировать в быстрый код).
3) Некоторые системы (Эльбрус, Burrougs, AS/400) поддерживают аппаратный контроль типов. Рядом с каждой ячейкой памяти есть дополнительно тег типа, показывающий, что именно там лежит. Процессор контролирует правильность работы с типами.
На таких системах принципиально невозможны вирусы.

В конце 1970-х Минобороны США объявляет конкурс на единый язык программирования для военных систем. Требования - надёжность, поддержка крупных систем, поддержка параллельного программирования.
В финал конкурса выходят 4 языка, из них побеждает "зелёный язык" француза Жана Ишбиа. Потом он был назван Ada, в честь Ады Лавлейс.

Ada - это язык на основе Паскаля, значительно расширенный, с мощной системой модулей, очень строгой типизацией и средствами параллельного программирования.
Но тоже очень объёмный (хотя и стройный, логичный). Долгое время не было эффективных реализаций, потом появились.

Язык применяется в области военных систем на Западе, в области авионики, космического ПО и т.п. (на Ada написано ПО самолёта-амфибии Бе-200).
Сильно повышает надёжность и качество программ.
http://ada-ru.org/ или см. лекцию С.И. Рыбина на oberoncore.ru в разделе "Статьи".
Компиляторы дорогие (тысячи долларов), но для академических целей предоставляются бесплатно (например, GNAT от компании AdaCore).

В конце 80-х Ada и Modula-2 выбраны Советом министров СССР как основные для оборонных задач. Но потом это решение забыли.

Хоар и Вирт сильно критиковали Аду за сложность, предсказывая ей проблемы. Предсказания сбылись, но на фоне сегодняшних "монстров" типа C++ Ada является красивым и элегантным языком.

7. C/C++

Когда появились микропроцессоры и ПК, они обладали малыми ресурсами (медленные, очень мало памяти - начиная от 1К, но максимум 32).
Это автоматически означало, что программирование для них отбрасывается назад, на уровень 50-х гг.
Кроме того, профессия программиста становится массовой - приходят новые люди, которые с энтузиазмом наступают на давно пройденные в отрасли грабли.

Снова популярно программирование на ассемблере, использование GOTO.

ОС вновь пишутся на ассемблере.
И тут в 1970 г. появляется ОС UNIX, изначально для мини-ЭВМ PDP-7.
Идея - написание ОС на языке повышенного уровня.
Д. Ричи и коллеги взяли язык BCPL, сделали на его основе язык B, а потом C, на котором и был разработан UNIX.
UNIX:
1) Обладает действительно изящной и расширяемой архитектурой (по меркам 70-х).
2) Полноценная многозадачная ОС, пригодная для работы на мини-ЭВМ.
3) Переносимая, потому что написана на языке повышенного уровня (Си).

Язык Си - типичный представитель языков повышенного уровня (т.е. "переносимых ассемблеров"). Никаких ценных особенностей не имел. Распространился вместе с UNIX.

Поскольку памяти на PDP-7 было мало, а разработка велась интерактивно (т.е. тексты программ редактировались прямо на ЭВМ), то нужно было экономить на длине программ. Поэтому синтаксис языка Си очень "птичий" - вместо ключевых слов широкое использование спец. символов и т.п. Позже этому синтаксису придумывали оправдания (вплоть до "философия Си - краткость записи"). На самом деле это противоречит принципам хороших языков.

Си - язык не высокого, а повышенного уровня. Ставилась задача максимально прямого доступа к памяти и оборудованию, чтобы это было так же просто, как из ассемблера. Поэтому типизация нестрогая - можно работать с любой переменной и с любой областью памяти как угодно. Нет контроля границ массивов. Это приводит к большому количеству ошибок в программах. В частности, на переполнении буферов (выходе за границы массивов) основаны атаки вирусов.
В Си отсутствуют модули. Исходный текст просто разбивается на файлы произвольным образом.

Си может быть удобен для написания небольших системных компонентов, работающих с оборудованием (драйверы), но очень неэффективен для разработки крупных систем. Плотность ошибок в программе на Си примерно в 16 раз выше, чем в

эквивалентной программе на современных версиях Паскаля.

```
#include <stdio.h>

int main (int argc, char** argv) {
  float x;
  scanf("%f", &x);
  float y = (x > 0) ? x : - x;
  printf("%f", y);
}
```

- программа вводит число и печатает его модуль.

Тем не менее, на Си разрабатываются даже критические по надёжности, но: 1) используется стандарт на кодирование, т.е. программистов принудительно заставляют использовать не все средства, а только "вменяемые". Программа в таком стиле похожа на паскалевскую.

2) используются специальные инструменты поиска ошибок.

3) трудоёмкость разработки сильно увеличивается.

Например, ОС реального времени QNX. Но: существует 30 лет, стоит бешеных денег. И имеет микроядерную архитектуру - очень маленькое ядро, которое работает в привилегированном режиме процессора, а все остальные компоненты ОС работают в пользовательском режиме процессора, т.е. их сбой не останавливает ОС, она их может перезапустить.

Надёжность в UNIX и подобных системах получается на уровне изоляции отдельных процессов.

Роль Си:

1) Раньше был основным языком системного программирования на ПК. Все старые ОС для ПК были на нём. Драйверы и т.п. Сейчас постепенно вытесняется другими языками (C++, C# в Windows).

2) C/C++ второй язык после Фортрана на суперкомпьютерах и кластерах.

3) Стал основой для C++, а на C++ разрабатываются очень многие приложения для Windows и Unix.

4) Синтаксис C (т.е. внешний вид программ) стал основой для многих других языков (Java, C#, JavaScript....). Не потому, что он хороший, а потому, что привычный (рыночные соображения - дать что-то привычное массам).

5) Основной язык, кроме ассемблера, при программировании микроконтроллеров и нестандартных процессоров. Если какая-то фирма выпускает свои процессоры, то она всегда поставляет C-компилятор для них. А другие компиляторы для этого процессора вообще могут не существовать.

В 80-х гг. был ряд попыток добавить в C средства для организации программ на более высоком уровне, в частности, средства ООП.

Например, Objective C получился объединением идей C и Smalltalk. Надёжность его выше. Используется как основной язык в Apple (Macintosh, iPhone...).

Но самый популярный потомок C - C++.

Бьярн Старуструп (Bell Labs) поставил своей задачей создать язык, пригодный для разработки сложных системных приложений, без потери быстродействия. Он выработал достаточно правильные требования к такому языку, но при этом решил сохранить обратную совместимость с C, чтобы любая C-программа являлась правильной и с точки зрения C++ и чтобы программисты могли постепенно переучиваться.

Это привело к двум проблемам: 1) сохранилась ошибкоопасность C; 2) долгое время доминировал смешанный стиль программирования. Использовать язык эффективно, так, как задумывал автор, стали только в 2000-х гг. Если использовать C++ таким образом, то надёжность действительно выше, чем у C - задействуется строгая типизация и т.д.

Другая проблема: Страуструп убеждён, что нужно предоставить программисту на выбор много средств, "не подавлять свободу", не навязывать одного стиля. C++ называют "мультипарадигменным" языком - на нём можно придерживаться десятков разных стилей программирования.

Это приводит ещё к двум проблемам: 1) "бардак" - приходится договариваться о том, какой именно стиль использовать в конкретной команде или проекте; 2) язык очень сложен, его компиляторы - тоже. Не существует полностью правильных компиляторов C++.

Линус Торвальдс сказал, что ядро Linux будет и дальше на чистом C, потому что нет компиляторов C++, внушающих доверие. Тем не менее, C++ используется даже в критических проектах (российская оборонка, например, компилятор C++ фирмы ЗАО "Интерстрон").

На примере C++ видны две принципиальных проблемы многих популярных языков:

1) Свобода для программиста - не благо, а скорее вред. Дисциплина помогает реальному творчеству, она освобождает программиста от мелочей и борьбы с ошибками. Поэтому - простой строгий синтаксис + строгая типизация.

2) Избыточная сложность как "раковая опухоль".

Страуструп сказал как-то:

1) Если бы не "рыночные" соображения, то он выбрал бы за основу для своего языка не C, а, например, Модулу-2 (паскаль-язык).

2) "Сейчас мне очевидно, что язык с возможностями C++ можно было бы сделать в 10 раз меньшим, если бы не обратная совместимость с C и не мультипарадигменность".

"На закуску" факты:

1) Директор Borland некогда сказал, что разработать компилятор C++ обходится в 100 млн. $. Разработать компилятор Оберона или Компонентного Паскаля можно за 5-10 млн. руб. При равном функционале и большей продуктивности работы на последних.

2) На протяжении более 10 лет никто не мог описать грамматику C++ формально - не существовало его БНФ!!! Отсюда несовместимости компиляторов между собой. Впервые БНФ разработал в 1996 г. Евгений Зуев (автор компилятора Интерстрон C++). При этом грамматика, подходящая для разработки компилятора, не подходит для изучения человеком, и наоборот.

Для Java так же грамматика была описана только через несколько лет после выхода языка.

3) ПО европейской ядерной лаборатории CERN (где Большой адронный колайдер) разрабатывается на C++, по решению руководства. Физики замучились. Невозможно эффективно решать задачи, созданные пакеты "падают" с ошибками "раз в пять минут".

Т.е. пренебрежение давно известными методами разработки языков и компиляторов.

Резюме:

массовое ИТ для ПК сильно замешано на двух составляющих - невежестве массовых программистов и на рыночных интересах компаний, "подсаживающих" на свои инструменты.

8. Появление скриптовых (сценарных) языков

С распространением ПК у множества пользователей возникает задача написания маленьких простых программок для своих нужд.

Поэтому возникло множество языков, которые позволяют "написать несколько строчек, не думая". Т.е. низкий порог вхождения для пользователей.

1) BASIC - диалоговый язык. Сначала не имел конструкций структурного программирования. Программа на традиционном BASIC-е становится совершенно нечитаемой при размере уже в несколько десятков строк. Но язык простой, его

интерпретаторы поставлялись на множестве разных ПК - был очень популярен. Потом Microsoft его "окультурила", добавив структурные средства - получился Quick Basic (поставлялся бесплатно с MS DOS), Microsoft Basic (платный) и, наконец, Visual Basic (применяется и по сей день, в том числе на платформе .NET и для Web-разработки - ASP.NET).

2) Пакетные языки ОС - shell-скрипты на UNIX, bat-скрипты в DOS, cmd-скрипты в Windows.

Для задач пакетной обработки текстов в UNIX был создан Perl. На нём долгое время разрабатывали CGI-приложения, до появления PHP. Программа на Perl совершенно нечитаема уже при длине в десяток строк.

3) Сценарные языки, встроенные в приложения - макроязыки, для написания макросов в приложениях. Например, VBA в Microsoft Office (Visual Basic for Applications). JavaScript в браузерах. Много примеров в научных пакетах и в играх.

Отдельные примеры - Python, Lua, Ruby.

Python был запланирован как скриптовый язык для ОС Amoeba Таннебаума. Как дополнение к Си. Но потом стал разрабатываться и распространяться как скриптовый язык общего назначения (в нише Бейсиков).

Lua - проектировался как язык-"клей" для интеграции компонентов (например, написанных на Си). Язык качественно спроектирован, прост и очень эффективен (используется даже в микроконтроллерах).

Ruby - нечто подобное Python.

Проблемы:

1) Все проектировались для маленьких программ (чтобы быстро написать и забыть). Нет средств обеспечения надёжности и крупной организации программ. Но теперь стали применяться для крупных приложений (Python, PHP). Качество очень низкое.

2) В частности, основная проблема этих языков - отсутствие строгой статической типизации. Во-первых, их авторы считали её неважной для маленьких программ и "непонятной" для новичков. Во-вторых, у них просто не хватало квалификации, чтобы реализовать компилятор языка со статической типизацией.

3) Часто путают низкий порог освоения языка с его пригодностью для образования. На самом деле скриптовые языки непригодны для образования, потому что не показывают ничего правильного, обладают массой дефектов.

4) Необходимость в макроязыках в приложениях есть, но опыта разработки языков у авторов приложений нет, поэтому все без исключения такие языки низкого качества. Если в вашем приложении нужен макроязык, лучше взять качественный существующий язык, например, Оберон.

5) Все эти языки разрабатывались как интепретаторы. А это медленно. Компиляторы создать трудно. Сейчас созданы компиляторы JavaScript, поэтому в браузерах он выполняется быстро. Google создал компилятор PHP.

9. "Нетрадиционные" языки

Кроме обычных императивных языков был создан ряд языков других парадигм.

Например, языки функционального программирования (ФП). В ФП нет изменяемых переменных, присваивания и, соответственно, циклов. Используются только функции, выражения и рекурсия.

Например, расчёт факториала на императивном языке:

```
VAR i, f: INTEGER;
BEGIN
  f := 2;
  FOR i := 3 TO 10 DO
    f := f * i
  END
```

В функциональном программировании:

```
f(0) = 1;
f(n: n < 0) = error('Недопустимое n');
f(n) = f(n-1)*n;
```

- функция определена кусочно, в зависимости от диапазона значений аргумента;
или аналогично с помощью условной конструкции:

```
f(n) = if n = 0 then 1 else if n > 0 then f(n-1)*n else error('Недопустимое n');
```

Для задач математической природы ФП имеет свои преимущества.
Языки:

LISP = "LISt Processing" = "язык обработки списков" (1958) - первый интерпретируемый язык. Активно использовался и используется в исследованиях по искусственному интеллекту. Язык "без синтаксиса" - есть только скобки и списки, остальное вводит сам программист. Один тип данных - список, и сама программа - тоже список, может сама себя обрабатывать. Префиксная форма записи выражений - сначала операция, потом её операнды.

```
(defun factorial n
  (if (= n 0) 1 (* n (factorial (- n 1))))
)
```

Древовидные списки, в принципе, похожи на XML.

Синтаксический разбор программы очень прост. Легко реализовывать метапрограммирование - когда программа сама себя меняет.

LISP долго был медленным языком, потом появились компиляторы в машинный код.

LISP очень гибкий язык, но, как всегда, для крупных программ гибкость выходит боком (трудно искать ошибки). Нет строгой типизации.

Вариант LISP - Scheme, часто используется как первый язык при обучении программированию, на Западе.

ML (60-е) - язык, изначально созданный для автоматического доказательства теорем. Компактный, строгий, имеет очень мощную строгую типизацию. Повлиял на многие функциональные языки. Его расширение - OCaml. Активно используется в некоторых задачах.

Haskell - разрабатывается с начала 90-х. Ведущие специалисты по ФП и авторы функц. языков объединились, чтобы сделать единый "магистральный" ФП-язык. Очень сильно развитый ML. Воплощены многие математические концепции. Программы на Haskell можно откомпилировать и для некоторых программ получить код, не уступающий по производительности C/C++.

Но в целом для ФП стоит проблема быстродействия. Например, так как вы не можете изменить отдельный элемент массива, значит, вы его копируете целиком с новым элементом. А уже сложный оптимизирующий компилятор должен понять, что это можно реализовать без копирования. Но компиляторы становятся очень сложными и содержат ошибки. Кроме того, функциональные языки требуют специальной поддержки во время выполнения - библиотек, реализующих всякие сложные средства языка. Эти библиотеки обычно пишут на C. Т.е. принцип "раскрутки" оказывается неприменим.

Отметим Erlang - язык, предназначенный для разработки телекоммуникационных приложений. Позволяет разрабатывать высоконагруженные приложения мягкого реального времени (для телефонных станций и т.п.) Разработан компанией Ericsson (Ericsson Language). Очень популярен в этой сфере. Набирает популярность в высоконагруженных Web-проектах. Для своих задач очень хороший язык.

Две общих проблемы функциональных языков:

1) Машина императивна. Как бы мы ни пытались уйти от присваиваний, на уровне машины у нас всё равно память и её изменения.

2) Многие задачи императивны - в них есть состояния. Любой технический объект,

которым управляем, имеет состояния. Даже взаимодействие с пользователем имеет состояния. Решение таких задач на функц. яз., которые не имеют состояния, неудобно.

В 60-х гг. императивное программирование казалось "ненаучным", без математического обоснования. А функциональное казалось математически обоснованным. Возникло мнение, что оно должно вытеснить "технарское" императивное.

На самом деле Дейкстра в "Дисциплине программирования" ввёл строгую математическую основу для императивного программирования, а у ФП слишком много ограничений, чтобы применять его во всех задачах.

Но популярность набирает.

Марковские языки - см. лекции прошлого года. Рефал (жив-здоров), SNOBOL (уже не используется), система комп. алгебры Schoonship.

Языки логического программирования - Prolog и его потомки. Описывается система правил, над ними идёт автоматический логический вывод. См. лекции прошлого года.

Forth (Форт) - язык с постфиксной записью. Т.е. операции пишутся после операндов. Автор - Чарльз Мур.

Например:

3 2 + 5 *

это вычисление (3 + 2) * 5

Как идёт вычисление постфиксной записи:

идём по выражению, пока не встречаем операцию. Выполняем её над предшествующими операндами и вместо них и операции записываем результат.

Доходим до +, складываем 3 и 2, записываем результат, остаётся:

5 5 *

Форт, как и Лисп, пример языка "без синтаксиса". В нём нет никаких специальных конструкций, программа - это множество "предложений" в постфиксной записи.

Стиль программирования - "снизу вверх": определяем базовые слова, потом из них более сложные "фразы", после чего на верхнем уровне появляются крупные "команды".

Сфера применения Форта: машины с ограниченными ресурсами. Очень быстрый откомпилированный код, либо есть режим компиляции, когда код медленнее, но занимает меньше места в памяти ("шитый код").

Форт может работать без ОС, он сам как ОС с командной строкой, с расширяемой системой команд.

Его потомок - PostScript, язык команд полиграфического вывода.

Стиль программирования на Форте предполагает разбиение программ на очень мелкие части, которые "обзываются" отдельными словами.

Пример того, как удачно найденная простота имеет очень большую мощь.

Раньше, чем Форт, была подобная система в СССР - ДССП (диалоговая система структурного программирования). Автор - Брусенцов, создатель троичной ЭВМ "Сетунь".

Сетунь и ДССП использовались в советских обсерваториях, Мур изначально астроном, он бывал в СССР. .....

10. Объектно-ориентированное программирование (ООП)

Для вычислительной техники и алгоритмизации характерно разделение на алгоритм и данные. Есть память, есть программа, есть процессор, по программе изменяющий память.

В то же время в системах реального мира алгоритмы не отделены от данных. Системы структурированы как множество объектов, где каждый объект имеет состояние (память) и поведение (алгоритмы изменения состояния).

Хотелось бы разрабатывать программы таким же образом, структурируя их в виде мелких объектов, в каждом из которых объединены и данные, и алгоритмы.

Но в то же время технически у нас только один процессор и единая память. Встаёт вопрос, как сымировать с их помощью "ансамбль" незавимых объектов, взаимодействующих друг с другом. Сегодня уже можно представить параллельно выполняемые алгоритмы для каждого объекта (существуют процессоры с сотнями и тысячами ядер), но и то...

Когда появлялось ООП, речь шла исключительно об однопроцессорном исполнении программы.

В 1960-х Оле Йохан Дал (Дания) для задач имитационного моделирования придумал ООП и реализовал его в языке Simula. Распространившийся вариант - Simula-67. Далу удалось расширить язык Алгол-60 таким образом, чтобы сделать возможным программирование в объектно-ориентированном стиле. Для популяризации Симулы и ООП потом много сделал Майкл Найгард.

ООП в современных языках в основном подобно Симуле.

11. Языково-операционные среды на ПК

Выше мы рассмотрели те проблемы, которые возникли при распространении микропроцессоров, и проблемы практик программирования, созданных для них.

Как могут быть разрешены эти проблемы?

Итак: есть задача - создать эффективный инструмент программирования на ПК.

Начало 70-х, США, XEROX PARC (исследовательская лаборатория XEROX) - создаётся экспериментальный мини-компьютер LISA. Имеет цветной монитор и манипулятор "мышь" (изобретена тогда же).

Нужно ПО для LISA.

Алан Кей предлагает идею операционной среды, которая была бы "заточена" под конкретный, специальный язык программирования, была бы написана на нём же и допускала бы удобное, неограниченное расширение пользователем.

Принципиальное отличие от существовавших ОС:

традиционные ОС расчитаны на запуск приложений, хранящихся в машинном коде. Писать приложения можно на любом языке. Это хорошо, с одной стороны, но с другой: приложениям очень трудно взаимодействовать между собой. Получается, что рабочая среда пользователя состоит из многих независимых приложений.

Идея Алана Кея - создать одноязыковую ОС, в которой не было бы крупных приложений, а было бы неограниченное количество мелких объектов, свободно взаимодействующих друг с другом. Такую ОС можно назвать языково-операционной средой.

Пользователь в ЯОС получает возможность свободно "допрограммировать" систему. Написанный им объект может быть легко интегрирован в систему.

Правильная идея - "программирующий пользователь". **Потому что программирование - это не особая деятельность, а просто форма взаимодействия пользователя с компьютером.** (формулировка Ф.В. Ткачёва, см. http://inr.ac.ru/~info21/). **При решении любых сложных задач эта форма взаимодействия самая удобная.** Но: при условии, что есть удобный инструмент программирования и ЯОС.

Например, нет более удобного и быстрого способа интегрировать плеер и таймер, чем написать программу в несколько строк.

Идея того же UNIX - позволить интегрировать через командную строку разные отдельные приложения. Там концепция конвейера команд, когда поток вывода одной программы направлен на вход другой программе.

Но! Командная строка неудобна для непрофессионалов.

Поэтому и язык программирования, и ЯОС должны быть удобны для работы любого пользователя.

Алан Кей использует идею ООП и разрабатывает язык Smalltalk. Он стремится к тому, чтобы программы на Smalltalk были похожи на фразы английского языка.

В Smalltalk есть только объекты и сообщения, посылаемые им.

Любая фраза программы выполняется слева направо - объекту, стоящему слева, посылается сообщение с параметрами, стоящими справа через :.

Например:

5 + 3 * 2

- это отправка объекту "число 5" сообщения "+" с параметром "3". В результате получится объект "число 8", ему будет отправлено сообщение "*" с параметром "2" и т.д.

Условная конструкция if - это сообщение, отправляемое объекту логического типа.

(5 < 2) if then: [.....] else: [....]

Объекту "5" посылается сообщение "< 2", в результате образуется объект "ложь", ему посылается сообщение if с параметрами then и else. В результате выполняется либо один, либо другой программный блок.

Т.е., наряду с LISP и Forth, это ещё один пример языка "без синтаксиса", у которого правила синтаксиса примитивны, а всё остальное записывается очень "однородно".

Такие языки очень однородны, гибки, "как пластилин". Иногда это удобно, но в целом такой синтаксис неестественен для мозга. Видимо, для хорошего восприятия человеком разные вещи в языке должны выглядеть по-разному, а не единообразно.

Smalltalk - интерпретируемый язык. Поверх "железа" запускается виртуальная машина Smalltalk (написанная на другом языке), а в ней уже - интерпретатор Smalltalk. Далее запускается ЯОС, с графическим интерфейсом, написанная на самом Smalltalk и доступная для изменения и расширения пользователю. (Можно переписать кусок системы - и она тут же начнёт работать по-новому).

Сейчас реализации Smalltalk запускаются поверх других ОС. Например, Squeak.

В языке полностью динамическая типизация. Т.е., если мы посылаем объекту animal сообщение fly, то только в момент выполнения программы будет ясно, умеет ли данный animal летать. Если это неподходящий animal (например, cat), то будет ошибка времени выполнения. Как обычно, отсутствие строгой статической типизации даёт гибкость, но создаёт проблемы при разработке крупных систем.

Smalltalk имеет стабильное сообщество поклонников. В качестве его плюсов называют лёгкость экспериментов - можно быстро создавать макеты-прототипы систем и потом их менять.

IBM и Sun когда-то в своих реализациях Smalltalk разработали технологию динамической компиляции в машинный код. Заранее откомпилировать Smalltalk-программу нельзя (таков язык), но во время выполнения кусочками это возможно. Технология называется JIT (Just-In Time compilation/codegeneration). Она используется в Java и .NET.

Также в Smalltalk использована революционная идея пользовательского интерфейса: текст как интерфейс. Форматированный текст с активными элементами. Этот интерфейс объединяет преимущества графического интерфейса и командной строки. При этом по сравнению с командной строкой всё сильно проще и удобнее (не надо помнить команды и каждый раз их набирать).

12. Системы Н. Вирта: Модула-2

Вирт был на стажировке в XEROX PARC (1975), работал с ПК LISA и Smalltalk, загорелся идеей ПК и ЯОС.

Вернулся в Цюрихский университет с идеей сделать первый европейский ПК.

В 1980 г. он (с коллектив) завершает ПК Lilith (графический экран, мышь, локальная

сеть, лазерный принтер...).
Создаёт ОС для Lilith: ОС Medos.
А для написания ОС создаёт язык Modula-2 (просто Modula была экспериментальной и совсем непохожей, она нигде не применялась, кроме экспериментов Вирта в 1974 г.).

Modula-2 получена усовершенствованием Паскаля. Язык ориентирован на системное программирование на ПК (т.е. на нишу С). Кроме того, ориентирован на образование, как и Паскаль.
Сделан более красивый и удобный синтаксис, убраны лишние "begin".
Если в Паскале:

```
if x > 0 then
begin
  A;
  B
end
else
begin
  C;
  D
end;
```

то в Модуле-2:

```
IF x > 0 THEN
  A;
  B
ELSE
  C;
  D
END
```

(кстати, как и в Ada).

Просто исчезает сокращённая форма if then A else B, теперь мы всегда должны написать END: IF THEN A ELSE B END.
В такой форме конструкции записываются не длиннее, чем в "хвалёном кратком Си":

```
WHILE x > 0 DO
  A
END
```

```
while (x > 0) {
  A;
}
```

Синтаксис требует заглавных букв. Это удобно - можно читать текст в любом редакторе или распечатке, без спец. программ, подсвечивающих его. С другой стороны, в потомке - Компонентном Паскале в среде Блэкбокс - жирность, цвет и прочее можно использовать для любого смыслового выделения (т.е. подсветка не "тратится" на ключевые слова, они и так видны).

Основные изменения - усовершенствована система типов данных; самое главное - введены **модули**. Модули - средство крупного структурирования программной системы и средство сокрытия деталей внутреннего устройства.

При том, что язык стал значительно "мощнее", он стал меньше: описание Паскаля - 60 стр., описание Модулы-2 - 40 стр.

Модула-2 широко распространилась как язык системного программирования. Второй уровень ОС для IBM AS/400 написан на Модуле-2 (нижний уровень написан на C++). В Канаде разрешены для применения в ПО АЭС только Ада и Модула-2.
Советом министров СССР в 1989 г. выбраны Модула-2 и Ада как основные языки для

оборонки и космоса (решение потом не оказало влияния).
НПО Решетнёва, российские спутники связи - всё ПО на Модуле-2.
Имея Модулу-2, выбирать С - невежество и преступление (если системы ответственные).

Borland планировала заменить Turbo Pascal на Turbo Modula-2. Но посчитали, что это опасно на рынке. Разработчики ушли, основали компанию Top Speed и долго продавали популярный компилятор и среду TopSpeed Modula-2.
А Borland стала "раздувать" Паскаль (Object Pascal, Delphi) и "плохо кончила".

Основная проблема Модулы-2 - нет поддержки ООП. Язык "почти объектно-ориентированный", но..

В 1987 г. появляется объектно-ориентированная Modula-3 (кажется, разработана в университете Нью-Йорка) и ОС, написанная на ней. Неплохой язык.

12. Системы Н. Вирта: Oberon

В 1986-1989 г. коллектив Вирта (в первую очередь его друг Юрг Гуткнехт) работают над первым европейским 32-битным ПК Ceres ("Церера").
Для него разрабатывается ОС "Оберон" и новый язык Оберон. (названо в честь спутника Сатурна или Юпитера).
ОС "Оберон" построена полностью по принципу языково-операционной среды (как Smalltalk).
Её языком является Оберон. Её интерфейс - графический, основанный на принципах активного текста.

Оберон:
- из Модулы-2 исключены некоторые маловажные или устаревшие средства (например, вложенные модули и вариантные записи);
- изменена система типов данных: во-первых, она сделана "герметичной", безопасной (см. ниже); во-вторых, добавлены расширяемые записи, что сделало язык объектно-ориентированным.

Объектная ориентация достигнута Виртом с помощью всего одного средства - расширяемых записей. Его идея в том, что суть ООП - в расширении типов данных (условно говоря, когда мы можем расширить тип "корова" от типа "парнокопытное", доопределив характерные для коровы признаки. И потом мы можем использовать "корову" во всех алгоритмах, где требуется "парнокопытное"). Основное значение ООП - возможность создавать расширяемые системы. Вы вводите новый, расширенный тип, а все старые компоненты могут с ним работать и не требуют внесения изменений.
Например, есть алгоритм размещения графических объектов на экране. Мы можем реализовывать новые виды графических объектов, а алгоритм будет успешно с ними работать.
Т.е. в расширяемой системе могут появляться компоненты, о которых ничего не было известно на момент разработки системы. Вот в чём основной смысл ООП.
Вирт это понимает и делает самую компактную из возможных поддержку ООП в языке. В то время как другие создатели языков (C++, Objective C, Object Pascal..) просто "тупо" добавляют в обычный язык средства из Simula-67 (классы, объекты...).
Поучительный опыт: если мы видим одну "хорошую штуку" и другую "хорошую штуку", то первое инстинктивное желание - их скомбинировать. Желание "прошито" в механизмах мозга, унаследованных от животных (умение скомбинировать ящик, палку и достать банан в зоне видимости). Увы, большая часть умственной деятельности человечества так и осуществляется. На самом деле нужно не просто комбинировать, а сначала анализировать и критически отсеивать сложные, несовершенные комбинации.
Размер описания Оберона - 16 стр.

Что такое "герметичная система типов"? Это значит, что программист не может написать такой код, который бы разрушил память. Контролируется выход за границы

массивов, нельзя обращаться по произвольным памяти и т.п.

Но в системных задачах иногда нужно напрямую работать с памятью. Для этого есть средства, но они изолированы в отдельной части языка, и всегда можно понять, является ли данный модуль безопасным или он эти средства использует. Прикладным программистам использовать эти средства не разрешается.

Следствия из герметичности:

1) Становится возможным автоматическое освобождение неиспользуемой памяти. Т.е. в старых языках (C, C++, Borland Pascal, Delphi 7...) если мы запросили блок памяти, то потом мы должны его явно вернуть, сказать диспетчеру памяти, что он нам больше не нужен. Существовало два вида ошибок: утечки памяти (программист забыл вернуть память) и "повисшие ссылки" (программист вернул память, но где-то в программе остались адреса этого фрагмента - и программа использует память, уже ей не принадлежащую). Эти ошибки практически неизбежны, их трудно отладить (даже воспроизвести бывает трудно - иногда программа падает, иногда нет). Единственный радикальный способ - запретить ручное освобождение памяти и делать это автоматически. Компонент, который во время работы программы периодически обходит память, находит ненужные уже программе блоки и помечает их как свободные, называется **сборщиком мусора** (garbage collector - GC). GC существовали в интерпретируемых функциональных языках типа LISP с 60-х гг. Но только Вирту в Обероне впервые удалось реализовать автоматическое управление памятью в компилируемом языке. В языках с негерметичной системой типов (C/C++) GC полноценно реализовать невозможно, потому что программист может хитрить и запоминать адреса памяти, например, в целой переменной - и как системе понять, что это адрес?

2) При наличии герметичности и GC язык становится **безопасным**. Это значит, что модули программы не могут испортить память друг друга. Таким образом, сбой в одном из модулей не приводит к непредвиденным нарушениям работы других модулей. (Конечно, если модули B и C пользуются модулем A, то сбой в A повредит их работе; но если модуль D ничего не знает про A, то что бы там ни натворил A, модуль D будет работать).

Прим.: по-английски герметичность типов - type-safety, безопасный язык - safe-language.

Безопасный язык в корне меняет устройство ОС. В ОС становится не нужна защита памяти и виртуальные адресные пространства. Защиту гарантирует язык программирования (своей строгой типизацией) и сборщик мусора.

В ОС Oberon все модули системы и прикладные находятся в общем пространстве памяти. Таким образом они могут свободно взаимодействовать, напрямую через средства языка программирования (обменяться переменными, вызвать процедуры друг друга и т.п.). В традиционных ОС обмен данными между двумя процессами достаточно сложен и ресурсоёмок.

В общем, ОС Oberon - это языково-операционная среда, но впервые она реализована на базе эффективного компилируемого языка.

Если в Smalltalk программирующий пользователь расширяет систему новыми классами и объектами, то в ОС Oberon - новыми модулями.

Также использован графический интерфейс на базе активного текста.

Начиная с системы Oberon начинают говорить о **компонентно-ориентированном программировании (КОП)**.

Идея КОП - в том, что программная система собирается из компонентов, разработанных, возможно, разными разработчиками, и далее может расширяться новыми компонентами, при этом расширяться прямо в процессе работы, без остановки.

Что технически требуется для КОП:

1) Безопасный язык программирования. Компоненты не должны нарушать работу друг друга.

В частности, сборщик мусора совершенно необходим (ни один из компонентов не может быть уверен, что другим компонентам не нужен какой-то фрагмент памяти.

Определить это и освободить данный фрагмент может только система).

2) Система модулей в языке программирования. При этом модули должны загружаться и соединяться динамически, прямо во время работы системы. Для этого модули должны содержать не только машинный код, но и дополнительную информацию о своих типах данных, переменных, процедурах. Такую информацию называют метаинформацией. Таким образом, программа на Обероне может, как и LISP-программа, сама себя изменять и узнавать о самой себе разную информацию.

3) Средства объектно-ориентированного программирования. Как мы помним, суть ООП в том, что в системе могут появляться новые разновидности объектов, а старые модули продолжают с ними работать.

Все эти средства впервые были объединены в языке Оберон, таким образом, стало возможным КОП.

Общая идея КОП - мечта о "рынке компонентов", т.е. о возможности приобретать и продавать отдельные компоненты программных систем, которые потом легко интегрируются между собой.

13. Развитие идей Оберона

На базе Оберона в Швейцарии, Австрии, Германии и др. странах была создана серия ОС (ETH Oberon, Oberon V4, Linz Oberon и др.), которые достаточно долго применялись на ПК.

Сам язык можно рассматривать как "бессмертное ядро" императивного программирования - в нём сконцентрировано самое главное и устареть оно не может (как дифференциальное исчисление в математике).

Принцип построения языка - "Сделай это как можно проще, но не проще, чем позволительно" (А. Эйнштейн).

Но для удобства применения язык всё же немного расширяли.

Так Вирт и Мессенбок добавили в него удобный для прикладного программирования механизм связанных с типом процедур (он похож на методы в Simula-подобных языках), вернули цикл FOR и получили Оберон-2.

Когда язык Ada стали делать объектно-ориентированным, то пошли тем же путём, что и Вирт - получилась Ada-95 (сейчас актуальна Ada-2005).

В Оксфорде Оберон был выбран вместе с функциональным языком Scheme как первый язык для обучения. Был реализован Oxford Oberon Compiler.

В СССР в Новосибирске в конце 80-х группа студентов делает 32-битный процессор и ПК "Кронос" (под влиянием идей Вирта и его Lilith). В это время в Новосибирске идёт проект "ЭВМ 5-го поколения" (начали японцы, все стали гнаться за ними) МАРС ("модульные асинхронные развиваемые системы"), который предполагал разработку многопроцессорной вычислительной архитектуры, которая могла бы масштабироваться от настольных ЭВМ до производительных вычислительных систем. Студенческий "Кронос" пришёлся вовремя - его выбрали в качестве отдельного узла многопроцессорной системы. МАРС использовал Модулу-2 в качестве основного системного языка (выше вводились другие языки - БАРС и ПОЛЯР). Проект МАРС был успешно выполнен, но государство не стало внедрять его результаты.

В то же время ПК "Кронос" развивался самостоятельно. Создатели написали для него UNIX-совместимую ОС на Modula-2. Сделали компилятор для Modula-2 и Oberon-2, который позволял использовать эти языки вместе. Было разработано много ПО: графические редакторы, САПР электронных схем и т.п.; "Кронос" производился и внедрялся как рабочая машина конструкторов на многих предприятиях.

Эти системы были внедрены в НПО Решетнёва для разработки ПО спутников связи, где Модула-2 и новосибирские компиляторы используются и по сей день.

http://kronos.su/

Позднее в Новосибирске была создана компания Excelsior, разработавшая пакет оптимизирующих компиляторов и среду разработки XDS Modula-2/Oberon-2. В 1994 г. это был лучший оптимизирующий компилятор в мире (быстрее лучших компиляторов

С). Сейчас он доступен бесплатно на сайте http://excelsior.ru/.

В 1993 г. Sun покупает лицензию на ОС Оберон, внимательно изучает, приглашает Вирта для консультаций.

В 1994-95 г, как известно, появляется язык Java - безопасный язык, в общем, подходящий для КОП. Правда, синтаксис взят от С, зато вся семантика - от Оберона. При этом есть принципиальные ошибки (нет модулей, всё сделано объектами и т.п.), влияющие на быстродействие программ. Для Оберона Франц (аспирант Вирта) уже в то время сделал динамическую кодогенерацию из промежуточного представления (JIT. Дисертация "Динамическая кодогенерация - ключ к переносимому ПО") и плагин к браузеру Netscape - JUICE. У Sun не хватило времени и квалификации на динамическую кодогенерацию, поэтому они сделали виртуальную машину Java как интерпретатор байт-кода (при этом байт-код - это немного модифицированный P-код Вирта, который был разработан когда-то для Паскаля).

Таким образом, Java можно рассматривать как боковую "гибридную" ветвь Паскалей. От Си там только внешний вид программного кода.

В 1993 г. группа бывших студентов Вирта основывает компанию Oberon Microsystems (http://oberon.ch/) и ставит целью реализовать ЯОС для Оберона поверх популярных ОС (Windows и MacOS). Т.е. "операционку поверх операционки". Проект сначала получил название Oberon/F. Ведущие архитектуры - Клеменс Шиперски и Куно Пфистер. Язык Оберон-2 также был усовершенствован, в него добавлены некоторые модификации для надёжного компонентного программирования; кроме того, система базовых типов (разрядности целых, символов и т.п.) приведена в соответствие с Java, чтобы сделать возможной совместную работу компонентов на двух языках.

Новый язык получил в 1997 г. название Component Pascal, а сама среда Oberon/F название BlackBox Component Builder. КП/ББ можно считать наиболее успешной промышленной Оберон-системой.

Сферы применения ББ - сложное ПО в разных отраслях (система управления крупнейшей в мире ГЭС в Бразилии, система проектирования радаров для истребителя Eurofighter (более 1 млн. строк кода), система машинного перевода для Евросоюза - разрабатывает университет Женевы. моделирование стрелкового оружия в DuPont; на сайте oberon.ch в числе клиентов названы автомобильные и авиационные корпорации. Компания Borland для своего JBuilder (Java-среда) заказывала JIT-компилятор у Oberon Microsystems, он был разработан в Блэкбоксе. В России применяется в Институте ядерных исследований РАН.)

Кроме того, Oberon Micr-s разработали в 90-х ОС реального времени JBed, которая поддерживает исполнение компонентов на КП и Java. Позднее от них отделилась компания Esmertec, которая продаёт Java-машину для мобильных устройств. На более чем 50% мобильных установлена Java-машина от Esmertec, а она написана на КП/ББ.

С применениями Оберон-технологий можно познакомиться на сайте http://oberoncore.ru/, раздел "Применения".

В конце 90-х Microsoft решает создать "свой ответ на Java" - среду .NET Framework. Идея в том, чтобы ввести промежуточный код, динамическую кодогенерацию, а сверху "разводить" компиляторы разных языков. Но базовый язык делается похожим на Java - это C#.

Microsoft для работы на .NET приглашает Клеменса Шиперски. В .NET использованы внутренние механизмы, задолго до этого испытанные в Блэкбоксе.

Кроме того, MS пригласила Андреса Хейлсберга - ведущего архитектора Borland Pascal и Delphi. Графические библиотеки .NET похожи на Delphi VCL.

Но .NET больше похож на традиционные компиляторы и среды разработки. Он не является расширяемой и гибкой языковой средой. Кроме того, совершенно проигнорирована идея "текст как интерфейс".

В Цюрихском университете (ETH) в 2000-х продолжалась работа над оригинальными ОС, теперь уже с прицелом на многоядерность и параллелизм. Есть язык Active Oberon и ОС A2 (раньше называлась BlueBottle). В ней поддерживается мощная графика и

параллелизм. Предназначена для встраивания в разные графические устройства (терминалы, автоматы, плееры). Поскольку нет адресных пространств и виртуальной памяти, то параллелизм очень эффективный (могут существовать десятки тысяч параллельных активных объектов, процессор без затрат времени переключается между ними).

Также в ETH создан экспериментальный язык и среда выполнения для него (работает как ОС поверх железа) Composita. Вместо традиционных средств ООП в ней иерархия параллельных процессов и передача сообщений ними. Ориентирована на имитационное моделирование. Т.е. вы описываете каждую сущность своим алгоритмом, соединяете сущности между собой - они начинают параллельно выполняться и обмениваться сообщениями. В Composita могут существовать сотни тысяч параллельных сущностей.

MS, посмотрев на A2 и Composita, провела силами MS Research проект "ОС нового поколения" Singularity. Её идеи взяты из Оберонов: написать ОС на безопасном языке (Sing#), без необходимости аппаратной защиты памяти, сделать поддержку параллелизма и проч. Кроме того, используются средства математической верификации (доказательство правильности). Singularity сейчас доступна в открытых исходных текстах (для некоммерческого использования). Её опыт использован в ядре Windows 7. MS утверждает, что при разработке новых ОС они активно используют математические методы верификации и т.п.

Кроме того, в MS Research был проект, подражающий Composita - MS Axum.

При финансировании MS в ETH разрабатывался язык для .NET - Zonnon (некоторые вариации на тему Оберона и Ады; автор Евгений Зуев, ранее он делал Интерстрон-C++).

Одно из преимуществ Оберона и его "лёгких" модификаций типа КП - из них нечего убрать, они как "автомат Калашникова". Просто, надёжно и эффективно.

Вирт выпустил модификацию первого Оберона - Oberon-07 (последние его уточнения - 2011 г.)

Это, видимо, лучший язык для программирования встроенных процессоров, типа ARM. Есть компилятор и среда такого программирования Astrobe.

В программировании есть некий "опимальный универсальный уровень", когда язык уже машинно-независимый и безопасный, но при этом сохранена полная прозрачность - мы точно знаем, какой машинный код получится и "сколько будет стоить" каждая строка нашей программы.

Оберон и Компонентный Паскаль как раз находятся на этом уровне. Переизобретать их бесполезно - будет то же самое.

Страуструп хорошо понимает необходимость в языке такого уровня, но его C++ небезопасен и крайне сложен.

Java и C# безопасны, но также очень сложны и уже не столь прозрачны, т.е. дальше от машины.

Доп. факты:

- в итоге стало ясно, что без модулей не обойтись, и в 2010 г. в очередной версии Java появились модули в том виде, в каком они были в Обероне.
- Google в 2010 г. опубликовал язык Go (планируют его как основной язык для своих серверных приложений). Авторы Go - создатели UNIX. Но по качествам он похож на Оберон (в описании авторы явно указывают Оберон-2 как образец).