МИНИСТЕРСТВО СЕЛЬСКОГО ХОЗЯЙСТВА РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
Иркутский государственный аграрный университет им. А.А. Ежевского
Кафедра информатики и математического моделирования

Учебное пособие

«Архитектура Компьютера и языки программирования»

для аспирантов 09.06.01 Информатика и вычислительная техника

Молодежный 2022

УДК 681.3.066 (075.8)

Печатается по решению методической комиссии Иркутского государственного аграрного университета имени А.А. Ежевского.

Протокол 5 от 28.03.2022г.

Рецензенты:

д.т.н., профессор, профессор кафедры информационные системы и защита информации ИрГУПС Краковский Ю.М.

к.т.н., доцент кафедры информатики и математического моделирования Иркутского ГАУ Белякова А.Ю.

Бендик, Н.В. Архитектура Компьютера и языки программирования: Учебное пособие [Текст]/Н.В. Бендик, М.Н. Полковская - Молодежный: Изд-во Иркутского ГАУ, 2022, - 117 с.

Учебное пособие предназначено для аспирантов направления подготовки 09.06.01 для освоения дисциплины «Архитектура компьютера и языки программирования». В пособии рассмотрены архитектура современного компьютера, организация процессора, основные характеристики процессоров CISC, RISC, MISC, организация процессов вычисления в процессоре, способы повышения их быстродействия за параллелизмом вычислений. В разделе программирования рассмотрены основные парадигмы программирования, характерные для современных языков программирования. Даны определения и характеристики основных парадигм: императивной, функциональной, логической и объектно-ориентированной. Для каждой парадигмы рассмотрены характерные для неё конструкции, языки с элементами синтаксиса и семантики.

ВВЕДЕНИЕ

Целью изучения дисциплины является знакомство со структурой и реализацией современных компьютеров и современными методами проектирования и разработки программного обеспечения

Основные **задачи** освоения дисциплины:

- освоение микро архитектуры, базовых микропроцессоров, коммуникационных сред;
- рассмотрение базовых принципов организации архитектур с параллелизмом на уровне данных, команд, потоков и процессов;
- знакомство с различными парадигмами программирования.

Результатом освоения дисциплины «Архитектура компьютера и языки программирования» является овладение аспирантами по направлению подготовки 09.06.01 Информатика и вычислительная техника следующих видов профессиональной деятельности:

- преподавательская деятельность по образовательным программам высшего образования;
- научно-исследовательская деятельность в области функционирования вычислительных машин, комплексов, компьютерных сетей, создания элементов и устройств вычислительной техники на новых физических и технических принципах, методов обработки и накопления информации, алгоритмов, программ, языков программирования и человеко-машинных интерфейсов, разработки новых математических методов и средств поддержки интеллектуальной обработки данных, разработки информационных и автоматизированных систем проектирования и управления в приложении к различным предметным областям.

Предлагаемое пособие состоит из двух частей. В первой части рассмотрены основные положения, структура и реализация современных процессоров, способы организации вычислительного процесса и средства повышения быстродействия путем параллелизма процессоров и вычислений.

Вторая часть посвящена программированию на языках, отмечены основные парадигмы программирования, и программирование на языках основных парадигм.

1. АРХИТЕКТУРА КОМПЬЮТЕРА

1.1. Структура компьютера

1.1.1. Типовая структура компьютера

В общем случае структуру Компьютера можно представить в следующем виде [1]:

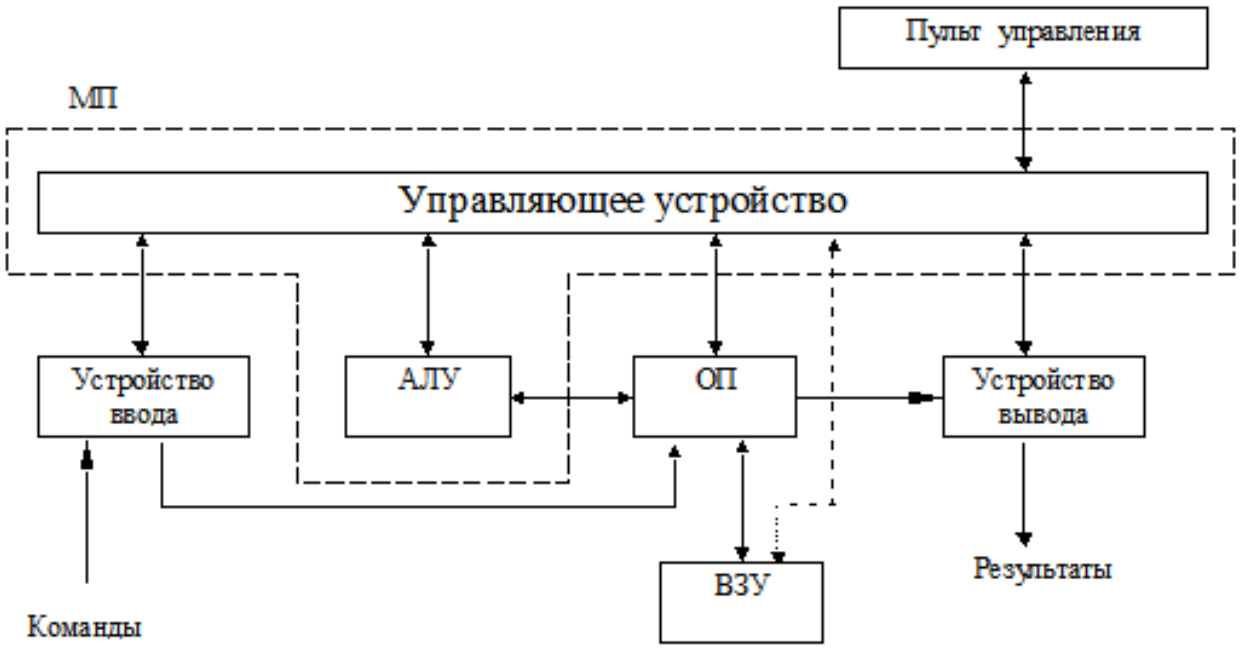


Рисунок 1.1. Типовая структура Компьютера

АЛУ – предназначено для выполнения арифметических и логических преобразований над данными определенной длины.

Память - предназначена для хранения информации (данных и программ). Часто состоит из оперативной памяти и внешнего запоминающего устройства.

Как правило, данные, к которым может обращаться АЛУ находятся в ОП

ВЗУ – используется для долговременного хранения данных

Управляющее устройство (УУ) - автоматически без участия человека управляет вычислительным процессом, посылая сигналы всем устройствам для реализации определенных действий (например, для выполнения определенной операции АЛУ).

УУ в своей работе руководствуется программой.

Программа состоит из команд, каждая из которых, определяет какое либо действие и операнд. Программа в свою очередь основывается на алгоритме решения поставленной задачи. Такой способ управления процессом решения задачи называется принципом программного управления.

Как правило, программы хранятся также в ОП наравне с данными. При этом перед выполнением программы собственно программа и данные должны быть помещены в ОП. Чаще всего это происходит через устройство ввода информации (клавиатура, диск). Команды выполняются в порядке следования в программе кроме команд перехода.

Устройства вывода служат для выдачи информации, результатов (например, на дисплей, принтер).

Пульт управления используется оператором для контроля хода выполнения программ и возможно для его прерывания (в ПКОМПЬЮТЕР - отсутствует).

Решение задач с помощью Компьютера представлено на рисунке:

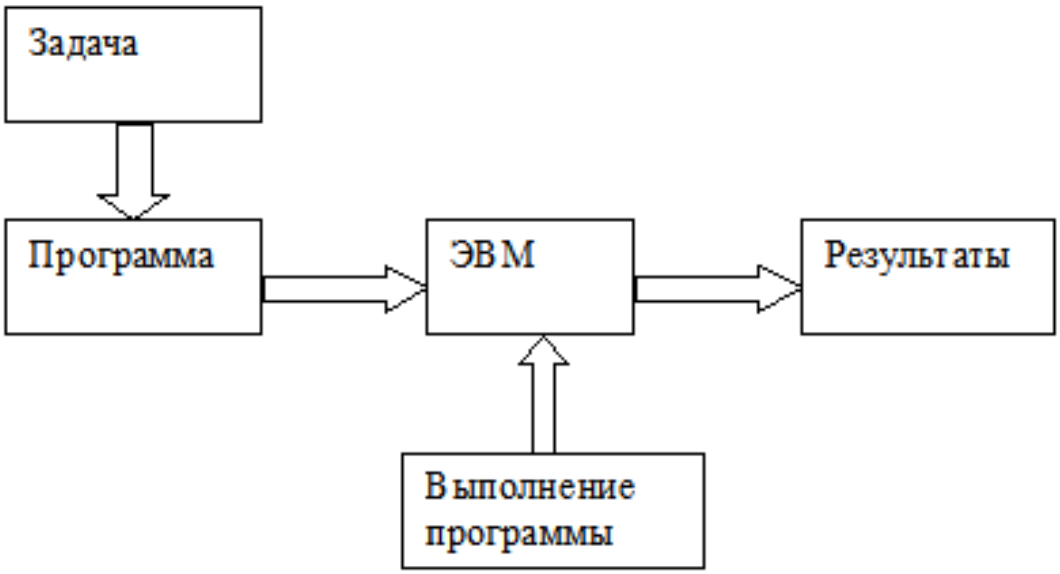


Рисунок 1.2. Схема решения задач на Компьютере

В настоящее время наиболее распространенным видом Компьютеров является персональный Компьютер. Структурная схема персонального Компьютера представлена на рисунке 1.3.

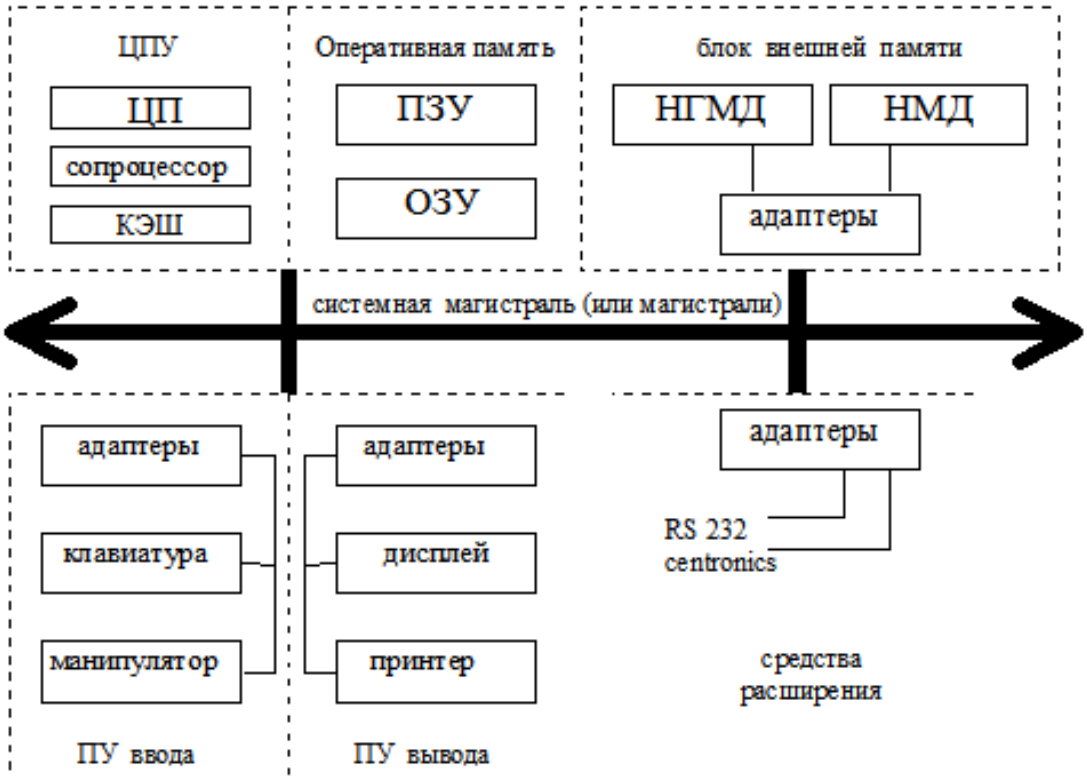


Рисунок 1.3. Структурная схема персонального Компьютера

1.1.2.Взаимосвязь аппаратных и программных средств компьютера

Поскольку в основу принципа работы компьютера положен принцип программного управления, то это означает, что для решения задачи в компьютере используются средства двух видов: аппаратные и программные.

Операционная система - вычислительная и центральная часть ПО: управление вычислительным процессом, планирование работы, распределения ресурсов компьютера, автоматизация процессов подготовки программ и организация их выполнения, облегчение общения оператора с компьютером.

Пользователь не может напрямую общаться с аппаратными ресурсами. Эту задачу берет на себя ОС.

Программа технического обслуживания - для проверки оборудования, диагностики, тестирования. Пакеты прикладных программ - базы данных, текстовые процессоры.

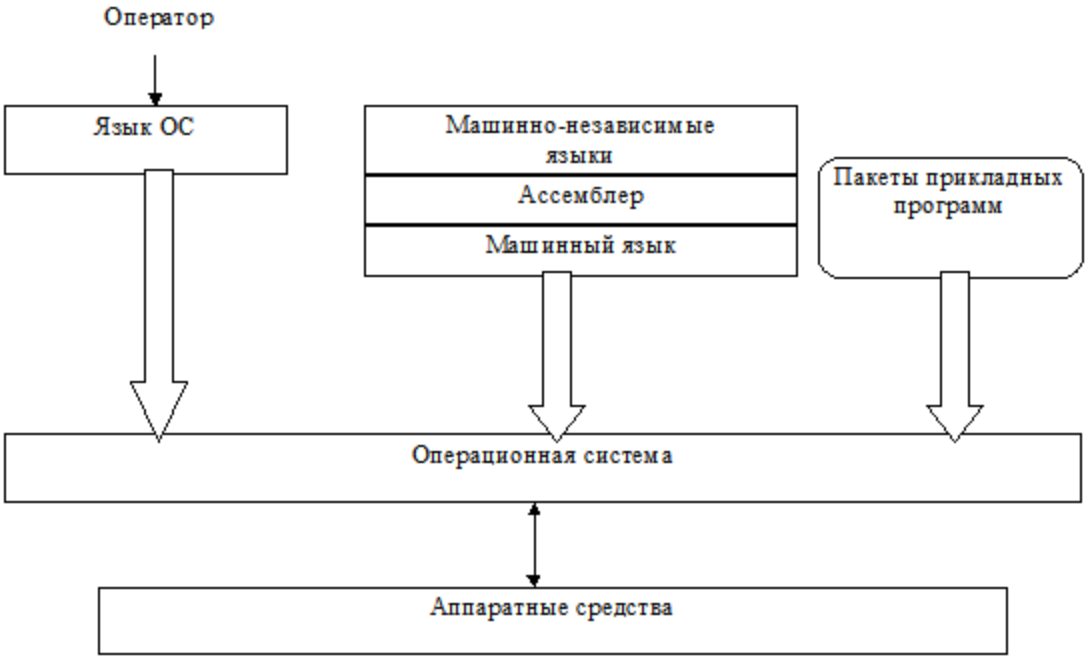


Рисунок 1.4. Структура системного программного обеспечения компьютера.

1.2. Процессор и микропроцессор

1.2.1. Понятие процессора и микропроцессора

При проектировании полупроводниковых автоматов, реализующих некоторую вычислительную функцию, всегда возможно два случая:

- реализовать всю логику автомата (логические функции И, ИЛИ, НЕ, ЕСЛИ... ТО и другие) аппаратно, то есть создать микросборку или микросхему, реализующий данный алгоритм "в самом себе",
- создать некоторый "универсальный вычислитель", который реализует весь алгоритм программно.

При заманчивой простоте первого подхода возникают следующие сложности: проектирование, проверка работоспособности, отладка на опытном образце микросхемы, отладка технологии ее изготовления – требуют много времени и средств, поэтому таким способом реализуются либо очень простые алгоритмы, либо микросхемы, реализующие массовые алгоритмы и поэтому выпускающиеся крупными сериями. Как раз к этим микросхемам относят микропроцессоры, способные не только работать по

строго заданным алгоритмам, но и "эмулировать" множество других алгоритмов.

Вследствие этого, применение "универсального вычислителя" предпочтительней если необходимо:

- при использовании однотипных сложных алгоритмов, требующих вычислительной мощности и для которых не выгодно изготовлять микросхемы с "жесткой" логикой;
- при реализации редко используемых алгоритмов любой сложности, либо в устройствах, для которых требуется частая смена алгоритма (например, в автоматизированных технологических линиях, научных исследованиях и т.д.)

Во всех этих случаях требуется некоторое устройство, являющимся этим вычислителем. Он называется "*процессором*".

Процессор – от латинского "продвижение" – часть вычислительного устройства, включающего АЛУ и УУ, предназначенная для реализации алгоритма (процесса самоуправляющейся последовательности операций для решения некоторой задачи) в соответствии с принципом *программного управления* – автоматическим выполнением последовательности команд составленной и введенной в компьютер программой.

Центральный процессор (ЦП) – основной рабочий элемент компьютера. Осуществляет вычисления алгоритма задачи и управление всем компьютером как единым целом.

Микропроцессор (МП) – полупроводниковый кристалл или комплект кристаллов, на которых реализован центральный процессор компьютера.

Сопроцессор – микропроцессорный элемент, дополняющий возможности основного процессора. Сопроцессор расширяет вычислительную (математическую, символьную, блочную) способность процессора, и одновременно – набор команд, которыми может пользоваться программист.

- *Микроконтроллер* – Можно рассматривать два вида: Микропроцессорная БИС, специально предназначенная для использования в управляющих устройствах, системы передачи данных, системах обработки изображений и системах управлениях технологическими процессами. Обычно микросхема такого контроллера имеет сравнительно небольшую разрядность слова и

богатый набор команд манипулирования отдельными битами, но не способна реализовать некоторые арифметические и строковые операции, характерные для центральных (универсальных) процессоров. Как правило, микроконтроллеры работают совершенно независимо от центрального процессора и часто не управляются им.

- Микропроцессорное устройство или система, предназначенная для использования в системах управления и основанные на микропроцессоре.

Процессор ввода/вывода – специализированный процессор, осуществляющий автономную обработку данных, которыми обменивается устройства ввода-вывода и центральный процессор или центральное ОЗУ компьютера. Может иметь собственные средства программирования. В настоящее время этот термин устаревает.

Связной процессор – специализированный процессор ввода-вывода, используемый для управления некоторым количеством линий и устройств связи. Эти линии работают медленно по сравнению со скоростями вычисления, поэтому один процессор может в режиме разделения времени обслуживать большое число линий (устройств). Связанные процессоры используются для обработки данных, организованных в виде блоков, пакетов, сообщений, дейтаграмм и т.д.

В настоящее время *универсальные* микропроцессорные устройства используются для реализации редко используемых либо часто меняемых алгоритмов с широким набором вычислительных функций, а для обработки сложных алгоритмов, требующих больших однотипных вычислений и которые не надо перепрограммировать, при этом используются *микроконтроллеры* или *микропроцессорные системы с зашитой программой*.

Исторически разделение процессоров на процессоры и микропроцессоры возникло в начале 70-х годов XX века, с началом производства больших интегральных схем. Собственно микропроцессор отличается от обычных процессоров тем, что располагается на одном кристалле БИС.

Если отслеживать поколения компьютеров по элементной базе, можно увидеть, что с первого по третье поколение компьютеров (на лампах, на транзисторах и на малых интегральных схемах) элементы компьютеров (АЛУ, регистры) занимали много места (по площади платы), и поэтому

объединять их в блоки по функциональному признаку не было смысла. Даже в ЭВМ III-го поколения существовали отдельные микросхемы сумматоров, регистров, дешифраторов, и часто было проще заменить отдельную неисправную микросхему, чем весь процессорный блок.

Ситуация стала меняться в конце 60-х годов XX века, когда, во-первых, уменьшились размеры элементов (так появились БИС), а во-вторых, увеличилась надежность и выход готовых микросхем. Тогда стали проводится исследования по построению процессора на одной микросхеме. От слов "микросхема" ("MICROchip") и процессор ("PROCESSOR") возникло понятие "микропроцессор" ("microprocessor"), а вовсе не от "миниатюрный процессор". Тогда микропроцессор считали чудачеством ученых и инженеров. В конце концов в 1968 году группа ученых во главе с Крэгом Барретом (Crag Barrett) выделилась со скандалом из Американского исследовательского центра, занимающегося разработкой технологии интегральных микросхем, в отдельную фирму – Intel. Эта группа стала создавать микропроцессоры. В 1969 году из этого же центра выделилась фирма AMD.

Объединение элементов процессора на одном кристалле повысила не только надежность, но и скорость его работы. После этого скорость работы процессоров увеличивали повышением его разрядности и частоты, о чем будет сказано ниже.

1.2.2. Разрядность процессора

Важным свойством микропроцессора является разрядность его шины данных и адреса. Выясним, почему это так.

С объединением элементов процессора в один кристалл наиболее узким местом в производительности процессора стала не пересылка данных между элементами процессора, а скорость обмена данными между процессором и остальными устройствами по шине. Поскольку любая операция, в том числе и пересылка данных, не может происходить быстрее, чем за такт, логично предположить, что желательно передавать как можно больше информации за один такт. Так как единицей информации является один бит (двоичный разряд), то, чем больше передается разрядов за один такт (по шине данных), тем быстрее работает процессор.

С разрядностью шины адреса немного сложнее. Дело в том, что вся адресуемая память компьютера пронумерована побайтно. Поэтому для обращения процессора к памяти ему необходимо запросить адрес нужных данных по адресной шине. Разрядность шины адреса определяет максимальный номер байта, который может быть затребован процессором. Так, при 8-ми разрядной шине возможна адресация 256 байт, при 16-ти разрядной – 64 Кбайт, при 32-х разрядной – 4 Гбайт . а при 64-х разрядной – 4 Гбайт

Между шиной адреса и шиной данных есть эмпирическое соотношение: чем больше процессор должен адресовать памяти (т.е. чем больше разрядность шины адреса), тем быстрее они должны поступать в процессор. Следовательно, тем шире шина данных. Однако на разрядность шин накладывается технологическое ограничение: чем шире шина, тем сложнее сделать его компоненты.

1.2.3. Тактовая частота.

Важнейшим параметром, определяющим скорость работы любого процессора, является тактовая частота. Она представляет собой импульсы прямоугольной формы, с которой синхронизируются все операции процессора. По другому, тактовая частота называется частотой синхроимпульсов. Тактовой же частотой она называется потому, что любая операция в процессоре не может быть выполнена быстрее, чем за один такт (период) синхроимпульсов.

Все операции в микропроцессоре синхронизируются со специальными синхроимпульсами, вырабатываемой специальной микросхемой – таймером. Синхроимпульсы нужны для того, чтобы все схемы работали с одинаковой скоростью. Дело в том, что разные элементы схемы (триггеры, сумматоры, логические элементы, дешифраторы) по определению работают с разной скоростью. Поэтому в "логику" схемы вводят дополнительный элемент – синхросигнал, и все операции происходят только в момент смены сигнала синхроимпульса с 0 на 1. Конечно, это намного замедляет работу системы, однако появляется гарантия, что операция будет происходить с текущими данными на "текущем" шаге, а не прошлыми или даже позапрошлыми,

поступившими с опозданием в преобразователь данных из-за разной скорости работы элементов схемы.

Кажется очевидным, что чем выше тактовая частота, тем выше скорость работы процессора. Однако не все так просто. Чтобы сделать тактовую частоту выше, необходимо уменьшить элемент схемы (т.е. уменьшить расстояние, проходимое носителями заряда). Это, во-первых, сложно технологически. Во-вторых, увеличивается сопротивления каждого элемента. Это значит (закон Джоуля-Ленца), что процессор будет сильно нагреваться. А это, в свою очередь, приведет к еще большему изменению параметров микросхемы и скорости работы различных участков микросхемы. И мы опять пришли к исходному состоянию. В-третьих, усиливается т.н. "дробовой эффект" в приборе, что может совершенно изменить соотношение "сигнал - шум" в микросхеме и исказить сигнал.

В начале 90-х годов XX века брак при производстве интегральных микросхем в США достигал 95% ! Чтобы цены на микропроцессоры не были астрономическими при таком браке, использовалась и до сих пор используется многоуровневая система контроля, позволяющая уменьшить брак до приемлемых 10%.

Как уже указывалось минимальное время исполнения команды – один такт. Но некоторые операции выполняются медленнее, или включают в себя несколько более простых операций, Такие операции выполняются за несколько тактов. Поэтому самый лучший способ повышения скорости работы компьютера – уменьшение количества тактов для одной сложной операции. Именно по этому пути идут разработчики архитектуры микропроцессоров.

1.2.4. RISC, CISC и MISC процессоры

Аббревиатура RISC (reduced instruction set computer) появилась в середине 80-х годов XX века, когда ученые из Беркли сообщили о создании "компьютера с ограниченным набором команд". С тех пор остальные компьютеры стали называться CISC (complication instruction set computer – компьютеры со сложным (расширенным) набором команд.) К CISC-процессорам относятся процессоры системы IBM 360/370, Intel 80x86 и Pentium, Motorola MC680x0, DEC VAX и некоторые другие. К RISC-

процессорам относятся Sun Ultra SPARC, MIPS, Alpha DEC, PowerPC и некоторые другие.

RISC-процессоры характеризуются следующими особенностями:

1. Из них удалены сложные (типа двоичного умножения) и редко используемые инструкции.
2. Все инструкции имеют одну длину. При этом уменьшается сложность устройства управления процессора и увеличивается скорость дешифрации команд.
3. Отсутствуют инструкции, работающие с памятью напрямую (типа команд "память - память", "регистр - память"). Возможна только загрузка данных из памяти в регистр и наоборот, из регистра в память. Соответственно на порядок увеличивается число регистров.
4. Отсутствуют операции работы со стеком.
5. Возможно использования конвейера и параллельных вычислений. АЛУ, например, одновременно может работать с 2-мя 32-х разрядными, 4-мя 16-ти разрядными, и 8-мью 8-ми разрядными числами. Смысл же конвейера – в накоплении последовательно выполняемых команд программы (т.н. линейных участков) в буфере для их ускоренного дешифрования и выполнения.
6. Почти все операции осуществляются за один такт микропроцессора.
7. Благодаря этим нововведениям тактовая частота RISC-процессоров (при прочих равных условиях) выше.

Более того, в RISC-микропроцессорах появилась возможность работы разных его составляющих на разных тактовых частотах. Например, из-за того, что содержимое памяти обычно дублируется в кэше, частоту работы АЛУ, регистров и дешифратора команд можно повысить, а частоту синхронизации пересылки между кэшем и памятью, предвыборки команд можно уменьшить. Поэтому при указании тактовой частоты процессора выбирают его максимальную частоту.

RISC-процессоры доказали свою состоятельность уже в начале 90-х годов XX века. К этому времени начался перевод большинства высокопроизводительных компьютеров (серверов) на RISC-архитектуру. Однако к тому времени накопилось большое количество программ, написанных для CISC-процессоров. Их перевод (перекомпиляция) для выполнения на RISC-процессоры заняло бы много времени (несколько

десятилетий), поэтому фирмы производители традиционных CISC-процессоров придумали оригинальный способ решения этой проблемы. Они разделили процессор на две части. В первой части, названной ядром, был реализован оригинальный RISC-процессор. Он работал с высокой тактовой частотой, и мог обращаться только к кэш-памяти. Во второй же части располагался блок "перекодировки" CISC-инструкций в RISC-команды, выполняемых ядром микропроцессора. В ней также находились:

- система управления сегментацией и страничного преобразования памяти;
- система управления кэш-памятью;
- конвейер;
- система предсказания ветвлений.

Таким образом, на входе этот процессор получал набор инструкций традиционного CISC-компьютера, а выполнял вычисления как RISC-компьютер. Это, во-первых, позволило увеличить производительность компьютера, а во-вторых, появилась возможность введения дополнительных, потоковых инструкций, благодаря которым контейнер стал работать производительнее и уменьшилось время перевода CISC-инструкций в RISC-команды. Традиционные, "чистые CISC-процессоры" этого не могли достигнуть.

Примером таких процессоров могут служить процессоры Pentium Pro, Pentium II, Celeron, Pentium III, Pentium 4 корпорации Intel, AMD K5, K6 и K7 (Athlon) корпорации AMD, Intel Itanium II и некоторые другие.

MISC (англ. *Minimal Instruction Set Computer*) - процессор, работающий с минимальным набором длинных команд. Дальнейшее развитие идей команды Чака Мура, который полагает, что принцип простоты, изначальный для RISC-процессоров, слишком быстро отошёл на задний план. В пылу борьбы за максимальное быстродействие, RISC догнал и перегнал многие CISC процессоры по сложности. Архитектура MISC строится на стековой вычислительной модели с ограниченным числом команд (примерно 20-30 команд).

Увеличение разрядности процессоров привело к идее укладки нескольких команд в одно большое слово. Это позволило использовать возросшую производительность компьютера и его возможность обрабатывать одновременно несколько потоков данных. Кроме этого MISC использует стековую модель вычислительного устройства и основные

команды работы со стеком Forth языка. MISC принцип может лежать в основе микропрограммы выполнения Java и .Net программ, хотя по количеству используемых команд они нарушают принцип MISC

Процессоры, образующие «компьютеры с минимальным набором команд» MISC, как и процессоры RISC, характеризуются небольшим числом чаще всего встречающихся команд. Вместе с этим, принцип «очень длинных слов команд» VLIW обеспечивает выполнение группы непротиворечивых команд за один цикл работы процессора. Порядок выполнения команд распределяется таким образом, чтобы в максимальной степени загрузить маршруты, по которым проходят потоки данных. Таким образом архитектура MISC объединила вместе суперскалярную и VLIW концепции. Компоненты процессора просты и работают с высокими скоростями.

1.3.Классификация микропроцессоров

Рассматривают следующие виды классификации микропроцессоров

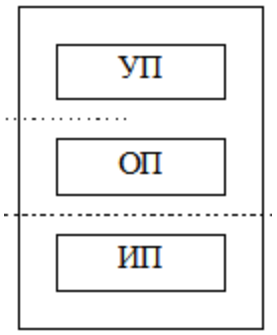
По числу БИС в микропроцессорном комплекте:

- однокристальные МП;
- многокристальные;
- многокристальные секционные.

В первую очередь на такое деление повлияли возможности БИС: ограниченное число элементов, выводов корпуса, в то время как МП довольно сложное устройство, имеющее много логических элементов и требующее большое количество выводов корпуса БИС.

Однокристальный МП получен при реализации всех аппаратных средств МП в виде одной БИС или СБИС. Основные характеристики таких МП зависят от технологии изготовления БИС.

Многокристальные МП получены при разбиении его логической структуры на функционально законченные части и реализация их в виде БИС.



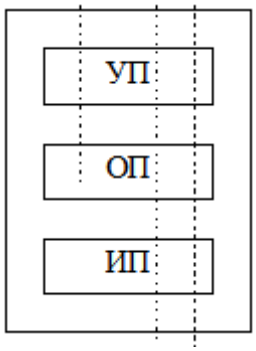
ОП - операционный процессор служит для обработки данных.

УП - управляющий процессор выполняет функции выборки, декодирования и вычисления адресов операндов, а также генерирует последовательность команд, формирует очередь команд.

ИП - интерфейсный процессор позволяет подключить память к МП.

УП, ОП, ИП могут работать автономно (параллельно) и тем самым организовывать конвейер операций.

Многокристальные секционные МП получаются когда в виде БИС реализуются логические структуры МП при функциональном разбиении ее вертикальными плоскостями.



Пример: если невозможно реализовать ОП 16-ти разрядов в одной БИС, его делят на части, реализуемые каждая в своей БИС. Они образуют микропроцессорные секции (например, 4-разрядная микропроцессорная секция).

По назначению:

- универсальные МП;
- специальные МП.

Универсальные МП могут быть применены для решения широкого круга задач. При этом их эффективная производительность мало зависит от проблемной специфики задачи. Как правило, это определяется достаточной широкой универсальной системой команд.

Специальные МП - проблемно ориентированные МП, которые нацелены на ускоренное выполнение определенных функций, что увеличивает эффективную производительность при решении только определенной задачи:

- математические процессоры;
- микроконтроллеры;
- параллельная обработка данных;
- цифровая обработка сигнала - цифровые фильтры и т. д. Пример: сравнение входного сигнала одновременно с несколькими эталонами для выделения нужного сигнала.

По характеру временной организации работы :

- синхронные;
- асинхронные.

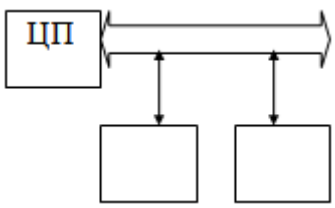
Синхронные - начало и конец выполнения операций задаются устройством управления (время выполнения не зависит от вида команды и операндов).

Асинхронные позволяют начало выполнения следующей операции определить по фактическому окончанию предыдущей. Устройства МП работают асинхронно, и после выполнения операции выдают сигнал о своей готовности. При этом, роль распределителя работ может брать на себя память, которая в соответствии с заранее установленном приоритетом выполняет запросы остальных устройств по обеспечению их командной информацией и данными.

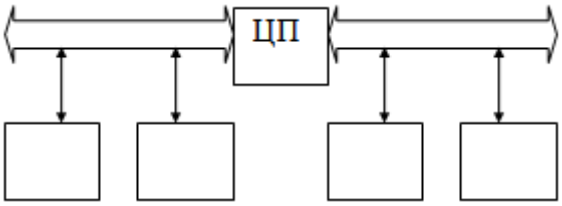
По организации структуры многопроцессорных систем:

- одномагистральные;
- многомагистральные.

Одномагистральные - все устройства имеют одинаковый интерфейс и подключаются к единой информационной магистрали, по которой передаются коды данных, адресов и управляющие сигналов.



Многомагистральные - устройства группами подключаются к своей информационной магистрали, это позволяет осуществить одновременную передачу информационных сигналов по нескольким магистралям. Производительность увеличивается.



По количеству выполняемых программ :

- однопрограммные;
- мультипрограммные.

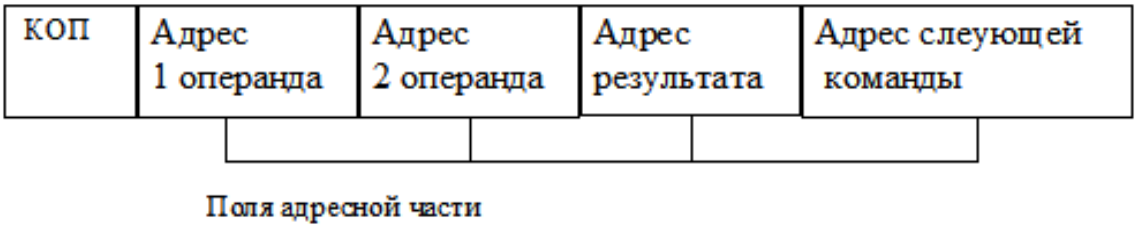
Мультипрограммные либо могут одновременно выполнять несколько программ, либо имеют средства для поддержки виртуальной мультипрограмности.

1.4.Типы структур команд. Способ расширения кодов операций

1.4.1. Типы команд

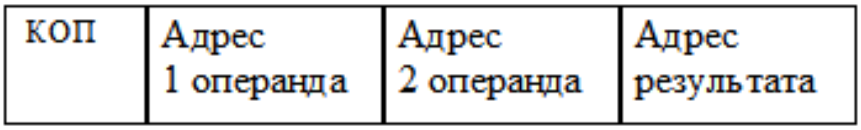
В современных компьютерах можно выделить пять типов команд.

1) четырехадресная команда



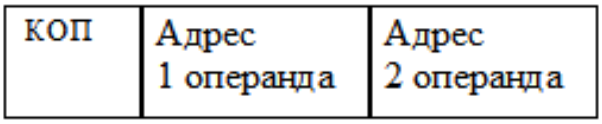
Для данной структуры характерна принудительная выборка команд

2) трехадресная



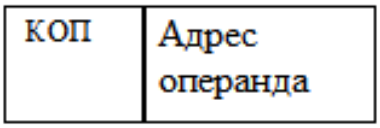
Здесь используется естественная адресация команд

3) двухадресная



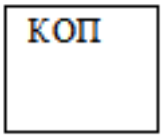
Результат всегда помещается на место одного из операндов (подразумеваемый адрес)

4) одноадресная



Адрес второго операнда и результата подразумевается (например, аккумулятор)

5) безадресная



Адреса всех операндов подразумеваются (например, стековая адресация)

Способ расширения кодов операций.

Если длины частей команды постоянны, то часто невозможно кодировать большое количество различных операций и одновременно иметь гибкую форму адресации данных. Это решается расширением кодов операций.

В данном случае длина КОП не остается постоянной. Часто используемые команды кодируются по возможности меньшим количеством бит, реже используемые – большим. При этом необходимо выделение специальных кодовых комбинаций, несущих информацию о длине КОП. Как правило, данный метод усложняет дешифрацию команд, однако позволяет значительно сократить длину часто используемых команд.

1.4.2.Общая структура команды. Способы адресации операндов

Обработка информации в компьютере осуществляется автоматически путем программного управления. Программа представляет собой алгоритм обработки информации, записанный в виде последовательности команд, которые должны быть выполнены для получения решения задачи.

Команда представляет собой код, определяющий операцию вычислительной машины и данные, участвующие в операции. Команда

содержит также в явной или неявной форме информацию об адресе, по которому помещается результат операции и об адресе следующей команды.

В команде зачастую содержатся не сами операнды, а информация об их расположении.

Общая структура команды имеет следующий вид:

Операционная часть содержит код операции и задает вид операции (+, -, *, /...)

Адресная часть содержит информацию об адресах операндов и результата операции, а иногда и адрес следующей команды.

Формат команды это структура команды с разметкой номеров разрядов, определяющих - границы отдельных полей команды.

Структура команды очень сильно влияет на производительность МП:

- Необходима как можно более короткая длина команды для большего числа операций или адресной информации.
- Чем больше команда, тем больше времени уходит на ее считывает и дешифрирование.

Однако, очень часто система команд достаточно велика, и разрядность ША составляет до 128 разрядов. Все это приводит к появлению очень длинных команд и снижению эффективности.

Если МП может выполнять M операций, то:

$n_{коп} >= \log_2 M$

Количество бит КОП для M=200: $n_{коп}=8$

Если ОП имеет S адресуемых ячеек, то для представления адреса одного операнда необходимо:

$n_A >= \log_2 S$

Кроме того, длина команд должна быть согласована с длиной обрабатываемых МП-ом данных (упрощается аппаратура). Это, как правило, приводит к длине команд, кратным разрядности шины данных

1.4.3. Способы адресации

Часто в командах указывают не сам адрес, а способ его вычисления.

Различают:

- адресный код - информация об адресе операнда, содержащаяся в команде.
- исполнительный адрес - номер ячейки памяти, к которой фактически производится обращение.

Способы адресации:

1. Подразумеваемый операнд.

В команде не содержится явных указаний об адресе операнда, операнд подразумевается и фактически задается кодом операции:

INC INX r (r) <- (r) + 1

2. Подразумеваемый адрес

В команде отсутствует адрес операнда или результата, но этот адрес подразумевается.

ADD B (A)<- (A) + (B)

3. Непосредственная адресация

В команде содержится не адрес, а сам операнд. Используется обычно для констант. Выборка операнда и формирование его адреса не нужно.

4. Прямая адресация

Исполнительный адрес соответствует адресному коду. В команде находится сам адрес.

“-” длинный адрес, необходимо считывать.

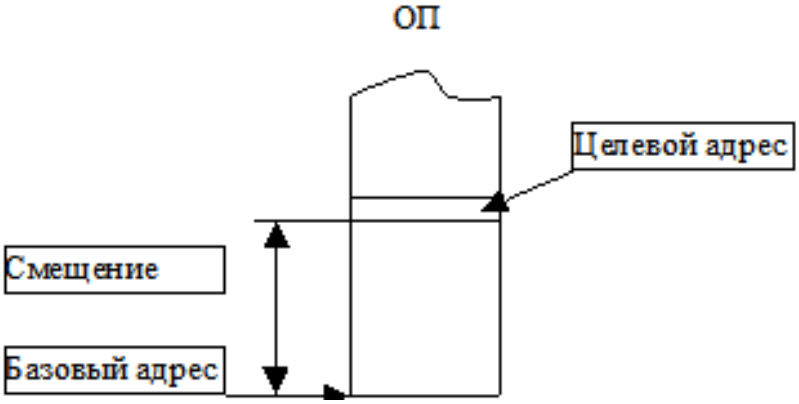
5. Относительная адресация (базирование)

Исполнительный адрес определяется суммой адресного кода команды и некоторого числа (базового адреса)

$A_{effисп}=A_{баз}+A_{ком}$

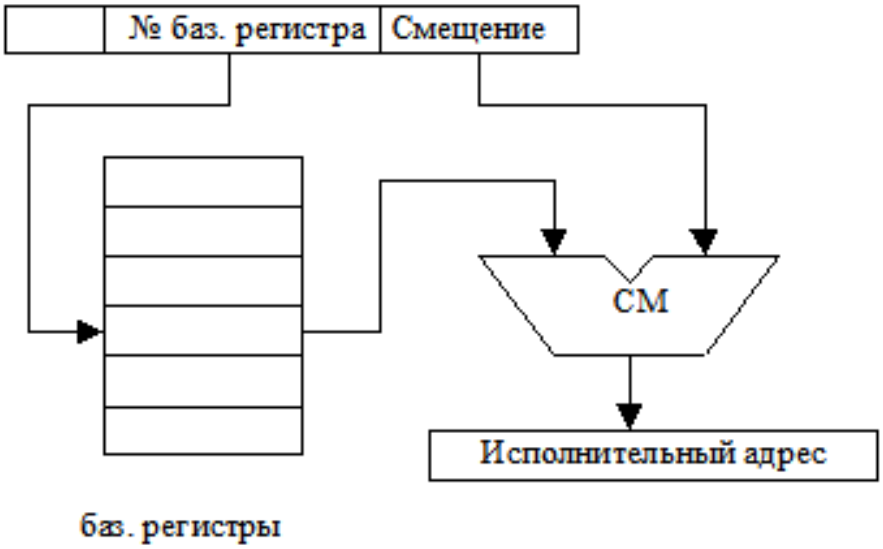
Базовый адрес часто хранится в специальном регистре (базовый регистр). В команде выделяется поле для указания номера базового регистра.

“+” меньшая длина адресного кода, при обращении к любой ячейки памяти. В команде указывается только смещение.

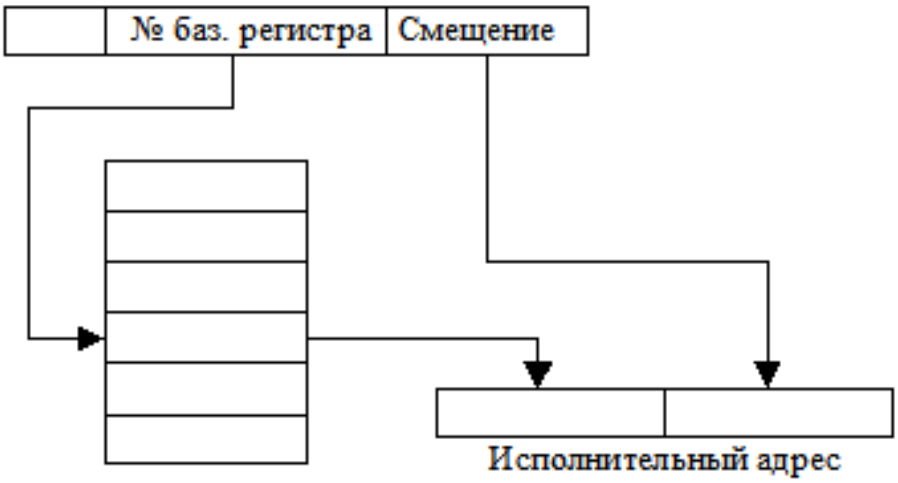


Различают адресацию суммированием и совмещением.

а)



б)



баз. регистры – старшие разряды адреса

а) используется чаще, но сложение - долго.

б) базовый адрес содержит старшие разряды, а следующий младшие разряды.

“-” не возможна адресация всей ОП.

6. Укороченная адресация

Адресный код содержит только часть адреса (младшие или старшие разряды). Остальное подразумевается. Может использоваться только совместно с другими способами адресации.

Используется для уменьшения длины команды.

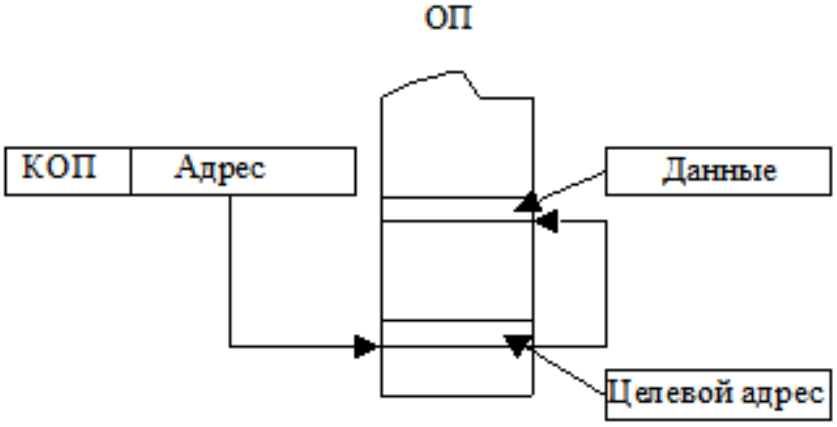
7. Регистровая адресация

Частный случай укороченной адресации

16 РОН - 4 разряда адреса

8. Косвенная адресация

Адресный код команды указывает адрес ячейки памяти, в которой находится адрес операнда.



9. Автоинкрементная и автодекрементная адресация

Эффективна при работе с массивами. Используется, например, при работе с косвенной адресацией, чтобы для обработки каждого элемента

массива не загружать новое значение адреса, а использовать автоматическое увеличение или уменьшение на 1 содержимого регистра с адресом.

10. Адресация слов переменной длины

Для повышения эффективности необходимо, чтобы имелась возможность выполнять операции над данными переменной длины. Адресация таких данных реализуется путем указания в команде местоположения в памяти начала слова и его длины.

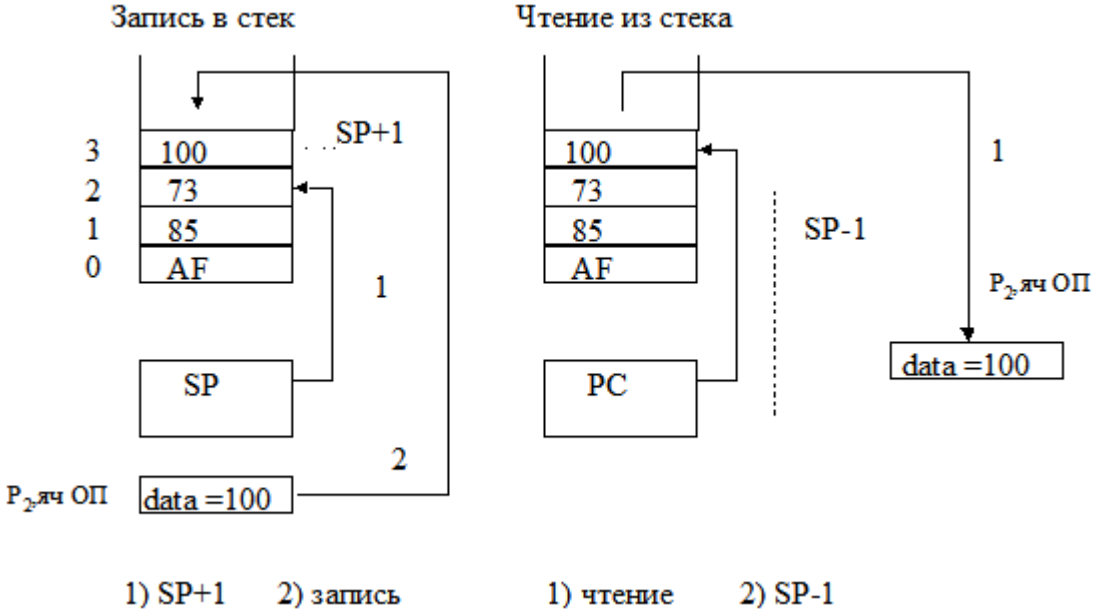
11. Стековая адресация.

Стековая память реализует безадресное задание операндов.

Стек представляет собой группу последовательно пронумерованных регистров или ячеек памяти снабженных указателями стека (регистрами), в которой автоматически поддерживается номер (адрес) последней занятой ячейки (вершина).

При записи заносимое в стек слово помещается в следующую свободную ячейку стека, а при считывании извлекается последнее поступившее в него слово. Принцип LIFO.

Т.к. указатель стека должен изменятся автоматически, то при операциях со стеком используется безадресное задание операнда, т.е. команда не содержит адреса ячейки стека (но содержит адрес 2-го операнда).



С помощью стека при определенных положениях операндов в стеке можно вычислять арифметические выражения. Для этого арифметические выражения записывают с помощью польской записи выражений ПОЛИЗ.

Для этого: читается арифметическое выражение слева направо и, последовательно друг за другом, выписываются встреченные операнды. Как только оказываются выписаны операнды некоторой операции, записывается знак этой операции. Если операция имеет результатом операнд некоторой предыдущей операции, знак которой выписан, то считаем этот операнд выписанным:

(k+l-m)*(p-s)

выписываем: kl+m-ps-x

Адрес K
Адрес L
+
Адрес M
-
Адрес P
Адрес S
-
+

адрес k, l, m - команды засылки в стек операндов.

-, +, * - безадресная команда операций.

Выполнение:

Команды операций выбираются из стека 1 или 2 операнда, выполняют операцию и записывают результат в стек.

Такой способ позволяет уменьшить длину команд и увеличить быстродействие, но является сложным и уменьшает возможность других команд (передача управления, ввод/вывод, ...)

В МП часто используются одновременно несколько способов адресации. Весьма часто в МП используют регистровую архитектуру с аккумулятором, адрес которого подразумевается в арифметических командах, что в совокупности с принципом расширения КОП позволяет получить небольшую длину КОП и достаточно простой управляющий автомат.

1.5. Технологии повышения производительности процессоров

1.5.1. Конвейерная обработка команд (pipelining). Суперскаляризация

Рассмотрим процесс выполнения процессором команды для коротких операций. Как об этом говорилось ранее, обработка команды, или цикл процессора, может быть разделена на несколько основных этапов, которые можно назвать микрокомандами, которых известно пять основных типов.

Каждая операция требует для своего выполнения времени, равного такту генератора процессора. Отметим, что к длинным операциям (ПТ) это не имеет отношения - там другая арифметика. Очевидно, что при тактовой частоте в 100 МГц быстродействие составит 20 млн. операций в секунду.

Все этапы команды задействуются только 1 раз и всегда в одном и том же порядке: одна за другой. Это, в частности, означает, что если первая микрокоманда выполнила свою работу и передала результаты второй, то для выполнения текущей команды она больше не понадобится и, следовательно, может приступить к выполнению следующей команды.

Конвейеризация осуществляет многопоточную параллельную обработку команд, так что в каждый момент одна из команд считывается,

другая декодируется и так далее, и всего в обработке одновременно находится пять команд. Таким образом, на выходе конвейера на каждом такте процессора появляется результат обработки одной команды (одна команда в один такт). Первая инструкция может считаться выполненной, когда завершат работу все пять микрокоманд.

Такая технология обработки команд носит название конвейерной (pipeline) обработки. Каждая часть устройства называется ступенью конвейера, а общее число ступеней - длиной линии конвейера.

С ростом числа линий конвейера и увеличением числа ступеней на линии увеличивается пропускная способность процессора при неизменной тактовой частоте. Процессоры с несколькими линиями конвейера получили название суперскалярных. Pentium - первый суперскалярный процессор Intel. Здесь две линии, что позволяет ему при одинаковых частотах быть вдвое производительней i80486, выполняя сразу две команды за такт.

1.5.2.Блоки операций с плавающей запятой

Начиная с 80486 восемь регистров для ПЗ (именуемых ST(0) - ST (7)) встраивают в центральный процессор. Каждый регистр имеет ширину 80 бит и хранит числа в формате стандарта ПЗ расширенной точности (IEEE floating-point standard).

Эти регистры доступны в **стековом порядке**. Имена (номера) регистров устанавливаются относительно вершины стека (ST (0) - вершина стека, ST(1) - следующий регистр ниже вершины стека, ST (2) - второй после вершины стека и так далее). Вводимые данные таким образом всегда сдвигаются «вниз» от вершины стека, а текущая операция совершается с содержимым вершины стека.

1.5.3. SLMD - процессы (команды)

Г. Флинном в 1966 году была предложена классификация ЭВМ и вычислительных систем (в основном - суперкомпьютеров), основанная на совместном рассмотрении потоков команд и данных. В процессорах таких известных производителей как Intel и AMD все более полно используются некоторые из этих архитектурных наработок.

Таблица 1.Классификации Флинна

Поток данных	**Поток команд**	
	Одиночный	**Множественный**
Одиночный	SISD - Single Instruction stream/Single Data stream (Одиночный поток Команд и	MISD - Multiple Instruction stream/ Single Data stream

	Одиночный поток Данных - ОКОД)	(Множественный поток Команд и Одиночный поток Данных - МКОД)
Множественный	SIMD - Single Instruction stream/Multiple Data stream (Одиночный поток Команд и Множественный поток Данных - ОКМД)	MIMD - Multiple Instruction stream/Multiple Data stream (Множественный поток Команд и Множественный поток Данных -МКМД)

Например, в общем случае архитектура SIMD (ОКМД) предполагает создание структур векторной или матричной обработки. Под эту схему хорошо подходят задачи обработки матриц или векторов (массивов), задачи решения систем линейных и нелинейных, алгебраических и дифференциальных уравнений, задачи теории поля и другое

В микропроцессорах массового выпуска при обработке мультимедийных данных также целесообразно применять подобные решения.

1.5.4.Динамическое исполнение (dynamic execution technology)

Динамическое исполнение - технология обработки данных процессором, обеспечивающая более эффективную работу процессора за счет манипулирования данными, а не просто линейного исполнения списка инструкций.

Предсказание ветвлений

С большой точностью (более 90 %) процессор предсказывает, в какой области памяти можно найти следующие инструкции. Это оказывается возможным, поскольку в процессе исполнения инструкции процессор просматривает программу на несколько шагов вперед.

Внеочередное выполнение (выполнение вне естественного порядка - out-of-order execution

Процессор анализирует поток команд и составляет график исполнения инструкций в оптимальной последовательности независимо от порядка их следования в тексте программы, просматривая декодированные инструкции и определяя, готовы ли они к непосредственному исполнению или зависят от результата других инструкций. Далее процессор определяет оптимальную последовательность выполнения и исполняет инструкции наиболее эффективным образом.

Переименование (ротация) регистров (register rename)

Чтобы избежать пересылок данных между регистрами в соответствующей команде изменяется адрес регистра, содержащего данные, участвующие в следующей операции. Поэтому вместо пересылки данных в регистр-источник осуществляется трактовка регистра с данными как источника.

Выполнение по предположению (спекулятивное – speculative)

Процессор выполняет инструкции (до пяти инструкций одновременно) по мере их поступления в оптимизированной последовательности (спекулятивно). Поскольку выполнение инструкций происходит на основе предсказания ветвлений, результаты сохраняются как предположительные («спекулятивные»). На конечном этапе порядок инструкций восстанавливается.

Существуют следующие варианты спекулятивного выполнения:

- предикация (predication) - одновременное исполнение нескольких ветвей программы вместо предсказания переходов (выполнения наиболее вероятного);
- опережающее чтение данных (speculative loading), то есть загрузка данных в регистры с опережением, до того, как определилось реальное ветвление программы (переход управления).

Эти возможности осуществляются комбинированно - при компиляции и выполнении программы.

Предикации

Обычный компилятор транслирует оператор ветвления (например, if-then-else) в блоки машинного кода, расположенные последовательно в потоке. Обычный процессор в зависимости от исхода условия исполняет один из этих базовых блоков, пропуская все другие. Более развитые процессоры пытаются прогнозировать исход операции и предварительно выполняют предсказанный блок. При этом в случае ошибки много тактов тратится впустую. Сами блоки зачастую весьма малы - две или три команды, - а ветвления встречаются в коде в среднем каждые шесть манд. Такая структура кода делает крайне сложным его параллельное выполнение.

При использовании предикации компилятор, обнаружив оператор ветвления в исходной программе, анализирует все возможные ветви (блоки) и помечает их метками или предикатами (predicate). После этого он определяет, какие из них могут быть выполнены параллельно (из соседних, независимых ветвей).

В процессе выполнения программы центральный процессор выбирает команды, которые взаимно независимы и распределяет их на параллельную обработку. Если центральный процессор обнаруживает оператор ветвления, он не пытается предсказать переход, а начинает выполнять все возможные ветви программы.

Таким образом, могут быть обработаны все ветви программы, но без записи полученного результата. В определенный момент процессор наконец «узнает» о реальном исходе условного оператора, записывает в память результат «правильной ветви» и отменяет остальные результаты.

В то же время, если компилятор не «отметил» ветвление, процессор действует как обычно - пытается предсказать путь ветвления и так далее Испытания показали, что описанная технология позволяет устранить более половины ветвлений в типичной программе, и, следовательно, уменьшить более чем в 2 раза число возможных ошибок в предсказаниях.

Опережающее чтения

Опережающее чтение (предварительная загрузка данных, чтение по предположению) разделяет загрузку данных в регистры и их реальное использование, избегая ситуации, когда процессору приходится ожидать прихода данных, чтобы начать их обработку.

Прежде всего, компилятор анализирует программу, определяя команды, которые требуют приема данных из оперативной памяти. Там, где это возможно, он вставляет команды опережающего чтения и парную команду контроля опережающего чтения (speculative check). В то же время компилятор переставляет команды таким образом, чтобы центральный процессор мог их обрабатывать параллельно.

В процессе работы центрального процессора встречает команду опережающего чтения и пытается выбрать данные из памяти. Может оказаться, что они еще не готовы (результат работы блока команд, который еще не выполнился). Обычный процессор в этой ситуации выдает сообщение об ошибке, однако система откладывает «сигнал тревоги» до момента прихода процесса в точку «команда проверки опережающего чтения». Если к этому моменту все предшествующие подпроцессы завершены и данные считаны, то обработка продолжается, в противном случае вырабатывается сигнал прерывания.

1.5.5. Многократное декодирование команд

В то время как традиционный процессор линейно переводит команды в тактовые микрокоманды и последовательно их выполняет, центральный процессор с многократным декодированием сначала преобразует коды исходных команд программы в некоторые вторичные псевдокоды (предварительное декодирование, или предекодирование), которые затем более эффективно исполняет ядро процессора. Эти преобразования могут содержать несколько этапов. В качестве примеров рассмотрим 2-3-ступенчатые декодеры.

Многократные декодеры и смежные технологии

- преобразование CISC/RISC в VLIW;
- преобразование VLIW/CICS в RISC;
- макрослияние;
- микрослияние

Декодирование команд CISC/RISC в VLIW

Декодирование команд CISC/RISC в VLIW. Эти технологии использованы в мобильных процессорах Crusoe (фирма Transmeta) и

некоторых центральных процессорах Intel - архитектуры IA-64 и EPIC (Explicitly Parallel Instruction Computing - вычисления с явной параллельностью инструкций).

В частности, в Crusoe на входе процессора - программы, подготовленные в системе команд Intel x86, однако внутренняя система команд VLIW не имеет ничего общего с командами x86 и разработана для быстрого выполнения при малой мощности, используя обычную CMOS-технологию. Окружающий уровень программного называют программным обеспечением модификации кодов (Code Morphing software - CMS, или CM), здесь осуществляется динамический перевод команд x86 в команды VLIW.

Аналогичные приемы используются в процессорах Intel Itanium - здесь при компиляции готовятся пакеты (связки, bundles) команд (по 3 команды в 128-битовом пакете). Тем самым, компилятор выполняет в данном случае функции CM.

Декодирование команд CISC VLIW в RISC

Указанные выше достоинства RISC-архитектуры привели к тому, что во многих современных CISC-процессорах используется RISC-ядро, выполняющее обработку данных. При этом поступающие сложные и разноформатные команды предварительно преобразуются в последовательность простых RISC-операций, быстро выполняемых этим процессорным ядром. Таким образом, работают, например, современные модели процессоров Pentium и K7, которые по внешним показателям относятся к CISC-процессорам. Использование RISC-архитектуры является характерной чертой многих современных процессоров.

Макрослияние (macrofusion)

В процессорах предыдущих поколений каждая выбранная команда отдельно декодируется и выполняется. Макрослияние позволяет объединять типичные пары последовательных команд (например, сравнение, сопровождающееся условным переходом) в единственную внутреннюю команду-микрооперацию (МкОП, micro-op) в процессе декодирования. В дальнейшем две команды выполняются как одна МкОП, сокращая полный объем работы процессора.

Микрослияние (micro-op fusion)

В современных доминирующих процессорах команды x86 (macro-ops) обычно расчленяются на МкОП прежде, чем передаются на конвейер процессора. Микрослияние группирует и соединяет МкОП, уменьшая их число. Исследования показали, что слияние МкОП вкупе с выполнением команд в измененном порядке может уменьшить число МкОП более чем на 10 процентов. Данная технология использована в системах Intel Core, а ранее апробировалась в ПЦ мобильных систем Pentium M.

В процессорах AMD K8 конвейер строится на том, что работа с потоком МкОП происходит тройками инструкций (AMD называет их линиями - line). Конвейер K8 обрабатывает именно линии, а не x86-инструкции или отдельные микрооперации.

Технология Hyper-Threading (HT)

Здесь реализуется разделение времени на аппаратном уровне, Разбивая физический процессор на два логических процессора, каждый из которых использует ресурсы чипа - ядро, кэш память, шины, исполнительное устройство.

Благодаря HT многопроцессная операционная система использует один процессор как два и выдает одновременно два потока команд. Смысл технологии заключается в том, что в большинстве случаев исполнительные устройства процессора далеки от полной загруженности. От передачи на выполнение вдвое большего потока команд повышается загрузка исполнительных устройств.

Технологии «невыполнимых битов»

Технологии «невыполнимых битов» (No-eXecute bit). Бит «NX» (63-й бит адреса) позволяет операционной системе определить, какие страницы адреса могут содержать исполняемые коды, а какие - нет. Попытка обратиться к NX-адресу как к исполняемой программе вызывает событие «нарушение защиты памяти», подобное попытке обратиться к памяти «только для чтения» или к области размещения операционной системы. Этим может быть запрещено выполнение программного кода, находящегося в некоторых страницах памяти, таким образом предотвращая вирусные или хакерские атаки. С теоретической точки зрения, здесь осуществляется виртуальное назначение «Гарвардской архитектуры» - разделение памяти для команд и для данных. Обозначение «NX-bit» используется AMD, Intel использует выражение «XD-bit» (eXecute Disable bit).

1.6.Технологии повышения производительности компьютеров

1.6.1. Методология повышения производительности компьютеров

По мере того как компьютеры становятся все более быстрыми, может возникнуть соблазн предположить, что компьютеры, в конечном счете, станут "достаточно быстрыми", и что аппетит увеличения вычислительной мощности будет постепенно уменьшаться. Однако история развития компьютеров показывает, что по мере того как новая технология удовлетворяет уже известные приложения, появляются новые приложения, интерес к которым был вызван этой технологией и которые теперь требуют разработки еще более новой технологии и так далее. Так, например, первые исследования рынка сбыта фирмой Cray Research предсказывали рынок в десяток супер-ЭВМ, однако с тех пор было продано много сотен супер-ЭВМ.

Традиционно, увеличение вычислительной мощности мотивировалось числовыми моделированиями сложных систем, таких как погода, климат, производственные процессы, физические и химические процессы, механические и электронные устройства. Однако, на сегодняшний день наиболее существенными силами, требующими разработки более быстрых

компьютеров, становятся коммерческие приложения, для которых необходимо, чтобы компьютер был способен обработать огромные объемы данных, причем используя многообразие сложных методов. Эти приложения включают базы данных (особенно, если они используются при принятии решений), видеоконференции, совместные рабочие среды, автоматизацию диагностирования в медицине, развитую графику и виртуальную реальность (особенно для промышленности развлечений).

Хотя коммерческие приложения могут в достаточной мере определить архитектуру большинства будущих параллельных компьютеров, традиционные научные приложения будут оставаться важными потребителями параллельной вычислительной технологии. Действительно, поскольку нелинейные эффекты затрудняют понимание чисто теоретических исследований, эксперименты становятся все более и более дорогостоящими, непрактичными или невозможными по политическим или каким-либо другим причинам (например, США проводит ядерные испытания, используя лишь суперкомпьютеры), то вычислительные исследования сложных систем становятся все более и более важными. Вычислительные издержки, обычно, увеличиваются как четвертая степень и даже более от точности вычислений. Научные исследования часто характеризуются большими требованиями к объему памяти, повышенными требованиями к организации ввода-вывода. Так что для научных исследований будет необходима все возрастающая мощность компьютера.

Потребность во все более мощных компьютерах будет определятся запросами как интенсивных по данным коммерческих приложений, так и интенсивных по вычислениям научных и технических приложений. Требования этих областей человеческой деятельности постепенно сближаются - научные и технические приложения вовлекают все большие объемы данных, а коммерческие приложения начинают применять все более сложные вычисления.

До настоящего времени эффективность самых быстрых компьютеров возросла почти по экспоненте. Первые компьютеры выполнили несколько десятков операций с плавающей запятой в секунду, а производительность параллельных компьютеров середины девяностых достигает десятков и даже сотен миллиардов операций в секунду, и, скорее всего этот рост будет продолжаться. Однако архитектура вычислительных систем, определяющих этот рост, изменилась радикально - от последовательной до параллельной. Эра однопроцессорных компьютеров продолжалась до появления семейства CRAY X-MP/ Y-MP - слабо параллельных векторных компьютеров с 4 - 16 процессорами, которых в свою очередь сменили компьютеры с массовым параллелизмом, то есть компьютеры с сотнями или тысячами процессоров.

Эффективность компьютера зависит непосредственно от времени, требуемого для выполнения базовой операции и числа базовых операции, которые могут быть выполнены одновременно. Время выполнения базовой операции ограничено временем выполнения внутренней элементарной операции процессора (тактом процессора). Уменьшение такта ограничено

физическими пределами, такими как скорость света. Чтобы обойти эти ограничения, производители процессоров пытаются реализовать параллельную работу внутри чипа - при выполнении элементарных и базовых операций. Однако теоретически было показано, что стратегия Сверхвысокого Уровня Интеграции (*Very Large Scale Integration - VLSI*) является дорогостоящей, что время выполнения вычислений сильно зависит от размера микросхемы. Наряду с VLSI для повышения производительности компьютера используются и другие способы: конвейерная обработка (различные стадии отдельных команд выполняется одновременно), многофункциональные модули (отдельные множители, сумматоры, и т.д., управляются одиночным потоком команды).

Все больше и больше в ЭВМ включается несколько "компьютеров" и соответствующая логика их соединения. Каждый такой "компьютер" имеет свои собственные процессор, память. Успехи VLSI технологии в уменьшении число компонент компьютера, облегчают создание таких ЭВМ. Кроме того, поскольку, хотя и очень приблизительно, стоимость ЭВМ пропорциональна числу имеющихся в ней компонент, то увеличение интеграции компонент позволяет увеличить число процессоров в ЭВМ при не очень значительном повышении стоимости.

Другая важная тенденция развития вычислений - это огромное увеличение производительности сетей ЭВМ. Еще недавно сети имели быстродействие в 1.5 Мбит/с, но в конце девяностых, сети с быстродействием в 1000 Мбит/с будут повсюду. Наряду с увеличением быстродействия сетей увеличивается надежность передачи данных. Это позволяет разрабатывать приложения, которые используют физически распределенные ресурсы, как будто они являются частями одного многопроцессорного компьютера. Например, коллективное использование удаленных баз данных, обработка графических данных на одном или нескольких графических компьютерах, а вывод и управление в реальном масштабе времени на рабочих станциях.

Рассмотренные тенденции развития архитектуры и использования компьютеров, сетей позволяют предположить, что в скором будущем параллельность не будет уделом только суперкомпьютеров, она проникнет и на рынок рабочих станций, персональных компьютеров и сетей ЭВМ. Программы будут использовать не только множество процессоров компьютера, но и процессоры, доступные по сети. Поскольку большинство существующих алгоритмов предполагают использование одного процессора, это потребуются новые алгоритмы и программы способные выполнять много операций одновременно. Наличие и использование *параллелизма* будет становиться основным требованием при разработке алгоритмов и программ.

Число процессоров в компьютерах будет увеличиваться, следовательно, можно предположить, что в течение срока службы программного обеспечения оно будет эксплуатироваться на все увеличивающимся числе процессоров. В этой связи можно предположить, что для защиты капиталовложений в программное

обеспечение, *маштабируемость* (*scalability*) программного обеспечения (гибкость по отношению к увеличению/изменению числа используемых процессоров) будет столь же важна как переносимость. Программа, способная использовать только фиксированное число процессоров, будет такой же плохой, как и программа, способная работать только на одном типе компьютеров.

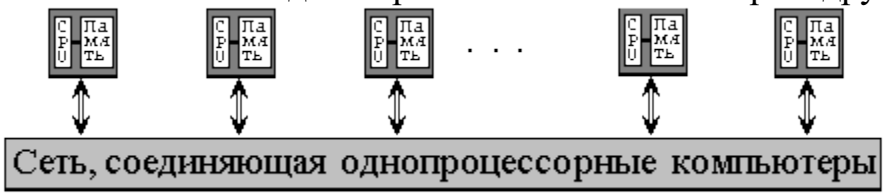
Отличительных признаком многих параллельных архитектур является то, что доступ к локальной памяти процессора дешевле, чем доступ к удаленной памяти (памяти других процессоров сети). Следовательно, желательно, чтобы доступ к локальным данным был более частым, чем доступ к удаленным данным. Данное свойство программного обеспечения называют *локальностью* (*locality*). Наряду с параллелизмом и маштабируемостью, он является основным требованием к параллельному программному обеспечению. Важность этого свойства определяется отношением стоимостей удаленного и локального обращений к памяти. Это отношение может варьироваться от 10:1 до 1000:1 или больше, что зависит от относительной эффективности процессора, памяти, сети, и механизмов, используемых для помещения данных в сеть и извлечения их из сети.

Теперь рассмотрим модели параллельных компьютеров, параллельных вычислений, проблемы, возникающие при разработке, написании и отладке параллельных программ.

1.6.2. Модели параллельных компьютеров

С самого начала компьютерной эры существовала необходимость во все более и более производительных системах. В основном это достигалось в результате эволюции технологий производства компьютеров. Наряду с этим имели место попытки использовать несколько процессоров в одной вычислительной системе в расчете на то, что будет достигнуто соответствующее увеличение производительности. Первой такой попыткой, осуществленной в начале 70-х годов, является ILLIAC IV. Сейчас имеется масса параллельных компьютеров и проектов их реализации.

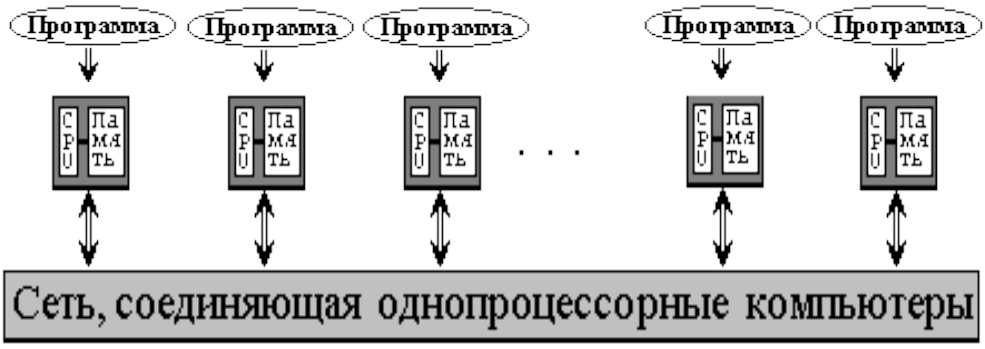
Архитектуры параллельных компьютеров могут значительно отличаться друг от друга. Рассмотрим некоторые существенные понятия и компоненты параллельных компьютеров. Параллельные компьютеры состоят из трех основных компонент: *процессоры*, *модули памяти*, и *коммутирующая сеть*. Можно рассмотреть и более изощренное разбиение параллельного компьютер на компоненты, однако, данные три компоненты лучше всего отличают один параллельный компьютер от другого.



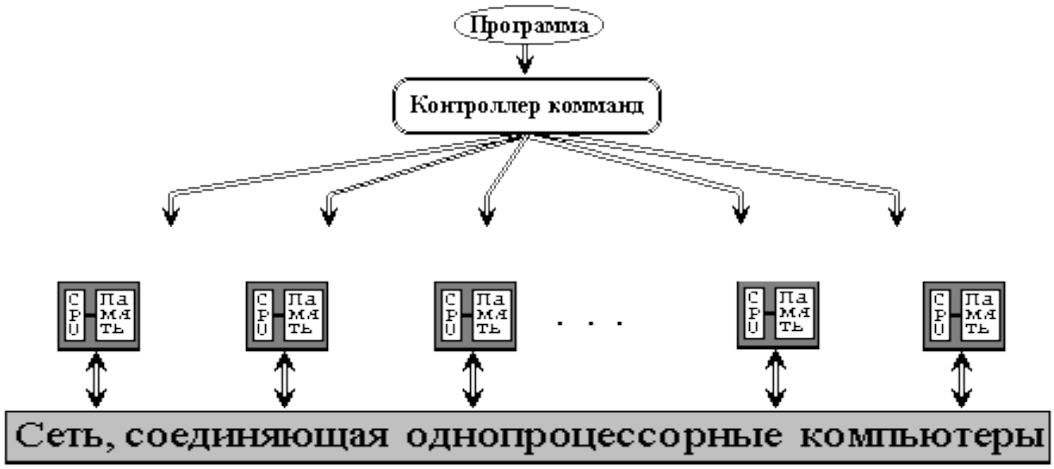
Коммутирующая сеть соединяет процессоры друг с другом и иногда также с модулями памяти. Процессоры, используемые в параллельных компьютерах, обычно точно такие же, что и процессоры однопроцессорных систем, хотя современная технология, позволяет разместить на микросхеме не только один процессор. На микросхеме вместе с процессором могут быть расположены те компоненты или их составляющие, которые дают наибольший эффект при параллельных вычислениях. Например, микросхема транспьютера наряду с 32-разрядным микропроцессором и 64-разрядным сопроцессором плавающей арифметики содержит внутри кристальное ОЗУ емкостью 4Кбайт, 32-разрядную шину памяти, позволяющую адресовать до 4Гбайт внешней по отношению к кристаллу памяти, четыре последовательных двунаправленных линии связи, обеспечивающих взаимодействие транспьютера с внешним миром и работающих параллельно с ЦПУ, интерфейс внешних событий.

Одним из свойств различающих параллельные компьютеры является *число возможных потоков команд*. Различают следующие архитектуры:

- *MIMD* (*Multiple Instruction Multiple Data* - множество потоков команд и множество потоков данных). MIMD компьютер имеет *N* процессоров, *N* потоков команд и *N* потоков данных. Каждый процессор функционирует под управлением собственного потока команд.



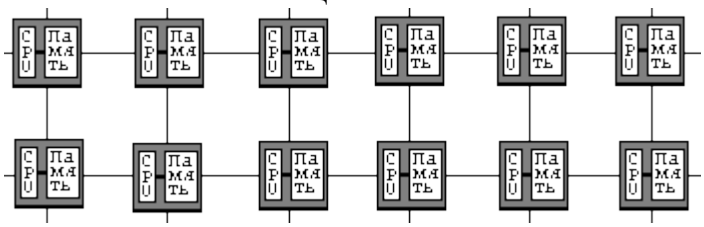
- *SIMD* (*Single Instruction Multiple Data* - единственный поток команд и множество потоков данных). SIMD компьютер имеет *N* идентичных синхронно работающих процессоров, *N* потоков данных и один поток команд. Каждый процессор обладает собственной локальной памятью. Сеть, соединяющая процессоры, обычно имеет регулярную топологию.



Другим свойством, различающим параллельные компьютеры, является *способ доступа к модулям памяти*, то есть имеет ли каждый процессор локальную память и обращается к другим блокам памяти, используя коммутирующую сеть, или коммутирующая сеть соединяет все процессоры с общей памятью. Исходя из способа доступа к памяти, различают следующие (довольно условные) типы параллельных (MIMD) архитектур:

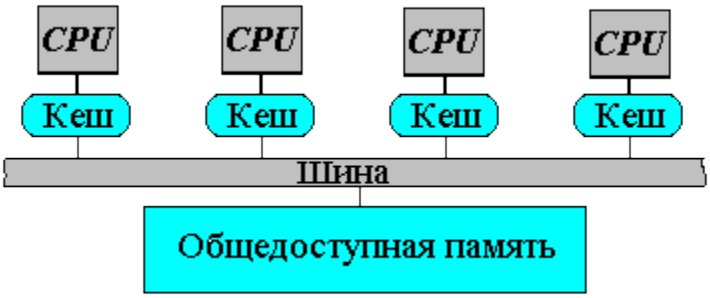
Компьютеры с распределенной памятью (*Distributed memory*)

Каждый процессор имеет доступ только к локальной собственной памяти. Процессоры объединены в сеть. Доступ к удаленной памяти возможен только с помощью системы обмена сообщениями.

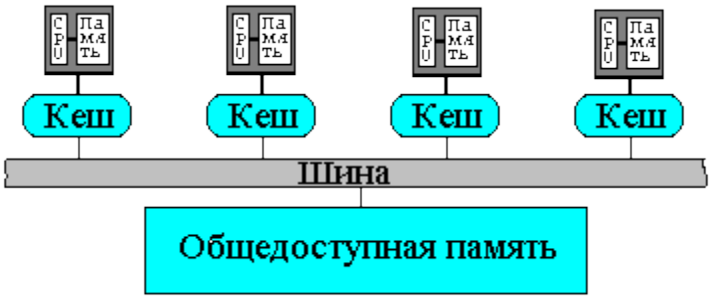


Компьютеры с общей (разделяемой) памятью (*True shared memory*)

Каждый процессор компьютера обладает возможностью прямого доступа к общей памяти, используя общую шину (возможно, реализованную на основе высокоскоростной сети). В таких компьютерах нельзя существенно увеличить число процессоров, поскольку при этом происходит резкое увеличение числа конфликтов доступа к шине.



В некоторых архитектурах каждый процессор имеет как прямой доступ к общей памяти, так и собственную локальную память.

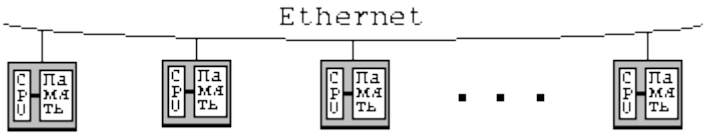


Компьютеры с виртуальной общей (разделяемой) памятью (*Virtual shared memory*)

В таких системах общая память как таковая отсутствует. Каждый процессор имеет собственную локальную память. Он может обращаться к локальной памяти других процессоров, используя "глобальный адрес". Если "глобальный адрес" указывает не на локальную память, то доступ к памяти реализуется с помощью сообщений с малой задержкой, пересылаемых по сети, соединяющей процессоры.

Отметим два класса компьютерных систем, которые иногда используются как параллельные компьютеры:

- *локальные вычислительные сети* (*LAN*), в которых компьютеры находятся в физической близости и соединены быстрой сетью,
- *глобальные вычислительные сети* (*WAN*), которые соединяют географически распределенные компьютеры.



Хотя системы этого сорта вводят дополнительные свойства, такие как *надежность* и *защита*, во многих случаях они могут рассматриваться как MIMD компьютеры, хотя и с высокой стоимостью удаленного доступа.

1.6.3.Модели параллельных вычислений

Параллельное программирование представляет дополнительные источники сложности - необходимо явно управлять работой тысяч процессоров, координировать миллионы межпроцессорных взаимодействий. Для того решить задачу на параллельном компьютере, необходимо распределить вычисления между процессорами системы, так чтобы каждый процессор был занят решением части задачи. Кроме того, желательно, чтобы как можно меньший объем данных пересылался между процессорами, поскольку коммуникации значительно больше медленные операции, чем вычисления. Часто, возникают конфликты между степенью распараллеливания и объемом коммуникаций, то есть чем между большим числом процессоров распределена задача, тем больший объем данных

необходимо пересылать между ними. Среда параллельного программирования должна обеспечивать адекватное управление распределением и коммуникациями данных.

Из-за сложности параллельных компьютеров и их существенного отличия от традиционных однопроцессорных компьютеров нельзя просто воспользоваться традиционными языками программирования и ожидать получения хорошей производительности. Рассмотрим основные модели параллельного программирования, их абстракции адекватные и полезные в параллельном программировании:

Процесс/канал (*Process/Channel*)

В этой модели программы состоят из одного или более процессов, распределенных по процессорам. Процессы выполняются одновременно, их число может измениться в течение времени выполнения программы. Процессы обмениваются данными через каналы, которые представляют собой однонаправленные коммуникационные линии, соединяющие только два процесса. Каналы можно создавать и удалять.

Обмен сообщениями (*Message Passing*)

В этой модели программы, возможно различные, написанные на традиционном последовательном языке исполняются процессорами компьютера. Каждая программа имеют доступ к памяти исполняющего е§ процессора. Программы обмениваются между собой данными, используя подпрограммы приема/передачи данных некоторой коммуникационной системы. Программы, использующие обмен сообщениями, могут выполняться только на MIMD компьютерах.

Параллелизм данных (*Data Parallel*)

В этой модели единственная программа задает распределение данных между всеми процессорами компьютера и операции над ними. Распределяемыми данными обычно являются массивы. Как правило, языки программирования, поддерживающие данную модель, допускают операции над массивами, позволяют использовать в выражениях целые массивы, вырезки из массивов. Распараллеливание операций над массивами, циклов обработки массивов позволяет увеличить производительность программы. Компилятор отвечает за генерацию кода, осуществляющего распределение элементов массивов и вычислений между процессорами. Каждый процессор отвечает за то подмножество элементов массива, которое расположено в его локальной памяти. Программы с параллелизмом данных могут быть оттранслированы и исполнены как на MIMD, так и на SIMD компьютерах.

Общей памяти (*Shared Memory*)

В этой модели все процессы совместно используют общее адресное пространство. Процессы асинхронно обращаются к общей памяти как с запросами на чтение, так и с запросами на запись, что создает проблемы при выборе момента, когда можно будет поместить данные в память, когда можно будет удалить их. Для управления доступом к общей памяти используются стандартные механизмы синхронизации - семафоры и блокировки процессов.

Модель процесс/канал характеризуется следующими свойствами:

Параллельное вычисление состоит из одного или более одновременно исполняющихся процессов, число которых может изменяться в течение времени выполнения программы.

Процесс - это последовательная программа с локальными данными. Процесс имеет входные и выходные порты, которые служит интерфейсом к среде процесса.

В дополнение к обычным операциям процесс может выполнять следующие действия: послать сообщение через выходной порт, получить сообщение из входного порта, создать новый процесс и завершить процесс.

Посылающаяся операция асинхронная - она завершается сразу, не ожидая того, когда данные будут получены. Получающаяся операция синхронная: она блокирует процесс до момента поступления сообщения.

Пары из входного и выходного портов соединяются очередями сообщений, называемыми каналами (channels). Каналы можно создавать и удалять. Ссылки на каналы (порты) можно включать в сообщения, так что связность может измениться динамически.

Процессы можно распределять по физическим процессорам произвольным способами, причем используемое отображение (распределение) не воздействует на семантику программы. В частности, множество процессов можно отобразить на одиночный процессор.

Понятие процесса позволяет говорить о местоположении данных: данные, содержащихся в локальной памяти процесса - расположены ``близко '', другие данные ``удалены''. Понятие канала обеспечивает механизм для указания того, что для того, чтобы продолжить вычисление одному процессу требуются данные другого процесса (зависимость по данным).

Модель *обмен сообщениями*

На сегодняшний день модель ***обмен сообщениями*** (*message passing*) является наиболее широко используемой моделью параллельного программирования. Программы этой модели, подобно программам модели *процесс/канал*, создают множество процессов, с каждым из которых ассоциированы локальные данные. Каждый процесс идентифицируется уникальным именем. Процессы взаимодействуют, посылая и получая сообщения. В этом отношение модель *обмен сообщениями* является разновидностью модели *процесс/канал* и отличается только механизмом, используемым при передаче данных. Например, вместо отправки сообщения в канал "channel 2" можно послать сообщение процессу "process 3".

Модель *обмен сообщениями* не накладывает ограничений ни на динамическое создание процессов, ни на выполнение нескольких процессов одним процессором, ни на использование разных программ для разных процессов. Просто, формальные описания систем *обмена сообщениями* не рассматривают вопросы, связанные с манипулированием процессами, Однако, при реализации таких систем приходится принимать какое-либо решение в этом отношении. На практике сложилось так, что большинство систем *обмена сообщениями* при запуске параллельной программы создает

фиксированное число идентичных процессов и не позволяет создавать и разрушать процессы в течение работы программы.

В таких системах каждый процесс выполняет одну и ту же программу (параметризованную относительно идентификатора либо процесса, либо процессора), но работает с разными данными, поэтому о таких системах говорят, что они реализуют ***SPMD*** (*single program multiple* data - одна программа много данных) модель программирования. SPMD модель приемлема и достаточно удобна для широкого диапазона приложений параллельного программирования, но она затрудняет разработку некоторых типов параллельных алгоритмов.

Модель *параллелизм данных* также является часто используемой моделью параллельного программирования. Название модели происходит оттого, что она эксплуатирует параллелизм, который заключается в применении одной и той же операции к множеству элементов структур данных. Например, "умножить все элементы массива *M* на значение *x"*, или "снизить цену автомобилей со сроком эксплуатации более 5-ти лет". Программа с *параллелизмом данных* состоит из последовательностей подобных операций. Поскольку операции над каждым элементом данных можно рассматривать как независимые процессы, то степень детализации таких вычислений очень велика, а понятие "локальности" (распределения по процессам) данных отсутствует. Следовательно, компиляторы языков с параллелизмом данных часто требуют, чтобы программист предоставил информацию относительно того, как данные должны быть распределены между процессорами, другими словами, как программа должны быть разбита на процессы. Компилятор транслирует программу с *параллелизмом данных* в SPMD программу, генерируя коммуникационный код автоматически.

Модель *общая память.* В модели программирования с ***общей памяти***, все процессы совместно используют общее адресное пространство, к которому они асинхронно обращаются с запросами на чтение и запись. В таких моделях для управления доступом к общей памяти используются всевозможные механизмы синхронизации типа семафоров и блокировок процессов. Преимущество этой модели с точки зрения программирования состоит в том, что понятие собственности данных (монопольного владения данными) отсутствует, следовательно, не нужно явно задавать обмен данными между производителями и потребителями. Эта модель, с одной стороны, упрощает разработку программы, но, с другой стороны, затрудняет понимание и управление локальностью данных, написание детерминированных программ. В основном эта модель используется при программировании для архитектур с общедоступной памятью.

2. ЯЗЫКИ ПРОГРАММИРОВАНИЯ

2.1. Принципы построения языков программирования

2.1.1. Определение языков программирования

Прежде чем анализировать конкретные парадигмы программирования, рассмотрим задачу определения систем программирования. Обычно при разработке системы программирования различают три уровня: синтаксис, семантика и прагматика реализуемого языка. Разграничение уровней носит условный характер, но можно констатировать, что синтаксис определяет внешний вид программ, семантика задает класс процессов, порождаемых программами, а прагматика реализует процессы в рамках конкретных условий применения программ. При изучении парадигм программирования центральную роль играет семантика, а синтаксис и прагматика лишь используются как вспомогательные построения на примерах.

Венский метод (ВМ) определения языков программирования был разработан в 1968 году в Венской лаборатории IBM под руководством П. Лукаса на основе идей, восходящих к Дж. Маккарти. Венская методика определения языков программирования с помощью операционной семантики, использующей концепции программирования при спецификации систем программирования, удобна для сравнения парадигм программирования.

Благодаря хорошо разработанной концепции абстрактных объектов, позволяющей концентрировать внимание лишь на существенном и игнорировать второстепенные детали, Венский метод годится для описания и машин, и алгоритмов, и структур данных, особенно при обучении основам системного программирования. По мнению ведущих программистов, Венский метод вне конкуренции в области описания абстрактных процессов, в частности, процессов интерпретации программ на языках программирования. Согласно концепции абстрактных объектов (абстрактный синтаксис, абстрактная машина, абстрактные процессы) интерпретирующий автомат содержит в качестве компоненты состояния управляющую часть, содержимое которой может изменяться с помощью операций, подобно прочим данным, имеющимся в этом состоянии.

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

Определение языка программирования (ЯП) по Венской методике начинается с четкого отделения существа семантики от синтаксического разнообразия определяемого языка.

С этой целью задается отображение конкретного синтаксиса (КС) ЯП в абстрактный синтаксис (АС), вид которого инвариантен для семейства эквивалентных языков.

КС => АС

Семантика ЯП определяется как универсальная интерпретирующая функция (ИФ), дающая смысл любой программе, представленной в терминах абстрактного синтаксиса. Такая функция использует определенную языком схему команд – языково ориентированную абстрактную машину (АМ), позволяющую смысл программ формулировать без конкретики компьютерных архитектур.

ИФ: АС => АМ

Выбор команд абстрактной машины представляет собой компромисс между уровнем базовых средств языка и сложности их реализации на предполагаемых конкретных машинах (КМ), выбор которых осуществляется на уровне прагматики.

АМ => КМ

Способ такого определения семантики языка программирования с помощью транслятора над языково ориентированной абстрактной машиной называют операционной семантикой языка. Если рассматривать в качестве транслятора интерпретатор, то принятая в операционной семантике динамика управления обладает большей гибкостью, чем это принято в стандартных системах программирования (СП) компилирующего типа. Это показывает резервы развития СП навстречу современным условиям разработки и применения информационных систем с повышенными требованиями к надежности и безопасности.

Синтаксис программ в языке программирования сводится к правилам представления данных, операторов и выражений языка. Начнем с выбора абстрактного синтаксиса на примере императивного языка Pascal.

Конкретный синтаксис языков программирования принято представлять в виде простых БНФ (Формул Бекуса-Наура). БНФ – это формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. БНФ используется для описания контекстно-свободных формальных грамматик. Существует расширенная форма Бэкуса-Наура, отличающаяся лишь более ёмкими конструкциями. В правой части таких грамматик допускается использование следующих «регулярных операций»:

A B - последовательно A, за тем B.

A | B - альтернатива. Читается: «A или B».

A* - произвольное количество повторений (в том числе - 0 раз) A. Читается: «последовательность A».

A # B - эквивалентно A (B A)*. Читается: «последовательность A через B».

Синтаксис описания алгоритмов в простейшем языке, поддерживающем императивную модель программирования, мог бы быть таким:

Оператор ::= Простой оператор | Структурный оператор;

Простой оператор ::= Оператор присваивания | Оператор вызова | Оператор возврата;

Структурный оператор ::= Оператор последовательного исполнения | Оператор ветвления | Оператор цикла

Оператор присваивания ::= Переменная := Выражение ;

Оператор вызова ::= Имя подпрограммы (Список параметров) ;

Оператор возврата ::= return [Выражение] ; Оператор последовательного исполнения ::= begin Оператор* end;

Оператор ветвления ::= if Выражение then Оператор* (elseif Выражение then Оператор*)* [else Оператор*] end;

Оператор цикла ::= while Выражение do Оператор* end.

Такой синтаксис позволяет сделать описание оператора *if* языка Pascal в БНФ:

<условный оператор if> ::= if <булево выражение> then <оператор> [else <оператор>]

<булево выражение> ::= <булево выражение> <булево выражение>| <выражение> <логический оператор> <выражение>

<логический оператор> ::= < | > | =

<выражение> ::= <переменная> | <строка> | <символ>

Для языка функциональной парадигмы Лисп данные языка - это атомы, списки и более общие структуры из бинарных узлов – пар, строящихся из любых данных, и синтаксис можно представить:

<атом> ::= <БУКВА> <конец_атома>

<конец_атома> ::= <пусто>

| <БУКВА> <конец_атома>

| <цифра> <конец_атома>

<данное> ::= <атом>

Это правило констатирует, что данные - это или атомы, или списки из данных.

Согласно такому правилу есть допустимое данное. Оно в языке Лисп по соглашению эквивалентно атому Nil.

Такая **единая структура данных** вполне достаточна для представления сколь угодно сложных программ. Дальнейшее определение языков Pascal и Лисп можно рассматривать как восходящий процесс генерации семантического каркаса, по ключевым позициям которого распределены **семантические действия** по обработке программ. Позиции распознаются как вызовы соответствующих семантических функций.

Другие правила представления данных нужны лишь при расширении и специализации **лексики** языка (числа, строки, имена особого вида и т.п.). Они не влияют ни на общий синтаксис языка, ни на строй его понятий, а лишь характеризуют разнообразие сферы его конкретных приложений.

1.1.2. Синтаксические конструкции языков программирования

Абстрактный синтаксис программ является уточнением синтаксиса данных, а именно – выделением подкласса **вычислимых выражений** (форм), т.е. данных, имеющих смысл как выражения языка и приспособленных к вычислению. Внешне это выглядит как объявление объектов, заранее известных в языке, и представление разных форм, вычисление которых обладает определенной спецификой.

Операционная семантика языка определяется как **интерпретация абстрактного синтаксиса**, представляющего выражения, имеющие значение. Учитывая исследованность проблем синтаксического анализа и существование нормальных форм, гарантирующих генерацию оптимальных распознавателей программными инструментами, в качестве абстрактного синтаксиса для Лиспа выбрано не текстовое, а списочное представление программ. Такое решение снимает задачу распознавания конструкций языка – она решается простым анализом первых элементов списков. Одновременно решается и задача перехода от конкретного синтаксиса текстов языка к абстрактному синтаксису его понятийной структуры. Переход получается просто чтением текста, строящим древовидное представление программы.

Ниже приведены синтаксические правила для обычных конструкций, к которым относятся идентификаторы, переменные, константы, аргументы, формы и функции.

<идентификатор> ::= <атом>

Идентификатор - это подкласс атомов, используемых при именовании неоднократно используемых объектов программы - функций и переменных. Предполагается, что идентифицируемые объекты размещаются в памяти так, что по идентификатору их можно найти.

Переменная - это подкласс идентификаторов, которым сопоставлено многократно используемое значение, ранее вычисленное в подходящем контексте. Подразумевается, что одна и та же переменная в разных контекстах может иметь разные значения.

Форма - это выражение, которое может быть вычислено.

Форма, представляющая собой **константу**, выдает эту константу как свое значение. В таком случае нет необходимости в вычислениях, независимо от вида константы. **Константные значения** могут быть любой сложности, включая вычислимые выражения. Чтобы избежать двусмысленности, в функциональной и логической парадигмах предлагается константы изображать как результат специальной функции **QUOTE, блокирующей вычисление**. Представление констант с помощью QOUTE устанавливает границу, далее которой вычисление не идет. Константные значения аргументов характерны при тестировании и демонстрации программ.

Если форма представляет собой переменную, то ее значением должно быть данное, связанное с этой переменной до момента вычисления формы.

Третья ветвь определения формы гласит, что можно написать **функцию, затем перечислить ее аргументы,** и все это как общий список заключить в скобки.

Аргументы представляются формами. Это означает, что допустимы **композиции функций.** Обычно **аргументы вычисляются в порядке вхождения** в список аргументов. **Позиция** «аргумент» выделена для того, чтобы было удобно в дальнейшем локализовать разные схемы обработки аргументов в зависимости от категории функций. Аргументом может быть любая форма, но метод вычисления аргументов может варьироваться. Функция может не только учитывать тип обрабатываемого данного, но и управлять временем обработки данных, принимать решения по глубине и полноте анализа данных, обеспечивать продолжение счета при исключительных ситуациях и т.п.

Последняя ветвь определяет формат условного выражения. Согласно этому формату **условное выражение** строится из синтаксически различимых позиций для **предиката** и двух обычных форм. Значение условного выражения определяется выбором по значению предиката первой или второй формы. Первая форма вычисляется при истинном значении предиката, иначе вычисляется вторая форма. Любая форма может играть роль предиката.

Название - это подкласс идентификаторов, определение которых хранится в памяти, но оно может не подвергаться влиянию **контекста вычислений**.

Функция может быть представлена просто именем. В таком случае ее смысл должен быть заранее известен. Функция может быть введена с помощью **выражения**, устанавливающего соответствие между аргументами функции и **связанными переменными**, упоминаемыми в теле ее определения (в определяющей ее форме).

Можно показать, что полученная система правил сводима к нормальным формам, гарантирующим возможность автоматического построения как автомата распознавания текстов, принадлежащих заданному этими формулами языку, так и автомата генерации списочных структур, эквивалентных этому языку. Поэтому приведенное определение может одновременно рассматриваться как определение и множества текстов языка – конкретный синтаксис, и множества структурных представлений программ – абстрактный синтаксис. Это снимает необходимость здесь рассматривать задачу перехода от конкретного синтаксиса к абстрактному.

Введя понятие универсальной функции можно считать, что интерпретация функций осуществляется как **взаимодействие** четырех подсистем:

- обработка структур данных;
- конструирование функциональных объектов (тело функции);

- идентификация объектов (имена переменных и названия функций);
- управление логикой вычислений.

Определение универсальной функции является важным шагом, показывающим **одновременно и механизмы реализации, и технику программирования на любом языке**.

1. В **строгой теории языка программирования** все функции следует определять всякий раз, когда они используются. На практике это неудобно. Реальные системы имеют большой запас встроенных функций, известных языку, и возможность присоединения такого количества новых функций, которые могут понадобится.
2. Для языков программирования характерно большое разнообразие условных форм, конструкций выбора, ветвлений и циклов, практически без ограничений на их комбинирование.
3. В реальных системах программирования **обычно поддерживается работа с целыми, дробными и вещественными числами в предельно широком диапазоне**, а также работа с кодами и строками. Такие данные, как и атомы, являются минимальными объектами при обработке информации, но отличаются от атомов тем, что их смысл задан непосредственно их собственным представлением.

2.2.Обзор парадигм программирования

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

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

Ведущая парадигма прикладного программирования на основе **императивного** управления и **процедурно-операторного стиля** построения программ получила популярность более пятидесяти лет назад в сфере узкопрофессиональной деятельности специалистов по организации вычислительных и информационных процессов. 21-й век резко расширил

географию информатики, распространив ее на сферу массового общения и досуга. В связи с этим меняются критерии оценки информационных систем и предпочтения в выборе средств и методов обработки информации.

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

Прикладное программирование подчинено проблемной направленности, отражающей компьютеризацию информационных и вычислительных процессов численной обработки, исследованных задолго до появления компьютеров. Именно здесь быстро проявился явный практический результат. Естественно, в таких областях программирование мало отличается от кодирования, для него, как правило, достаточно операторного стиля представления действий. В практике **прикладного программирования** принято доверять проверенным шаблонам и библиотекам процедур, избегать рискованных экспериментов.

Ценится точность и устойчивость научных расчетов. Язык Фортран - ветеран прикладного программирования. Лишь в последнее десятилетие он стал несколько уступать в этой области Паскалю-Си, а на суперкомпьютерах - языкам параллельного программирования, таким как Sisal.

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

Основные средства и методы программирования сложились по мере возрастания сложности решаемых задач. Произошло расслоение парадигм программирования в зависимости от глубины и общности проработки технических деталей организации процессов компьютерной обработки информации. Выделились разные стили программирования, наиболее зрелые из которых – низкоуровневое (машинно-ориентированное), системное, декларативно-логическое, оптимизационно-трансформационное, и высокопроизводительное/параллельное программирование.

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

Системное программирование долгое время развивалось под прессом сервисных и заказных работ. Свойственный таким работам производственный подход опирается на предпочтение воспроизводимых процессов и стабильных программ, разрабатываемых для многократного использования. Для таких программ оправдана компиляционная схема обработки, статический анализ свойств, автоматизированная оптимизация и контроль. В этой области доминирует императивно-процедурный стиль программирования, являющийся непосредственным обобщением операторного стиля прикладного программирования. Он допускает некоторую стандартизацию и модульное программирование, но обрастает довольно сложными построениями, спецификациями, методами тестирования, средствами интеграции программ и т.п. Жесткость требований к эффективности и надежности удовлетворяется разработкой профессионального инструментария, использующего сложные ассоциативно семантические эвристики наряду с методами синтаксически-управляемого конструирования и генерации программ. Бесспорный потенциал такого инструментария на практике ограничен трудоемкостью освоения - возникает квалификационный ценз.

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

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

для разных категорий функций, свобода рекурсивных построений, доверие интуиции математика и исследователя, уклонение от бремени преждевременного решения непринципиальных проблем распределения памяти, отказ от необоснованных ограничений на область действия определений – все это увязано Джоном Мак-Карти в идее языка Лисп [8]. Продуманность и методическая обоснованность первых реализаций Лиспа позволила быстро накопить опыт решения новых задач, подготовить их для прикладного и теоретического программирования. В настоящее время существуют сотни функциональных языков программирования, ориентированных на разные классы задач и виды технических средств.

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

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

Дальнейшее развитие парадигмы программирования отражает изменение круга лиц, заинтересованных в применении информационных систем. Формирование экстенсивных подходов к программированию – естественная реакция на радикальное улучшение эксплуатационных характеристик оборудования и компьютерных сетей. Происходит переход вычислительных средств из класса технических инструментов в класс бытовых приборов. Появилась почва для обновления подходов к программированию, а также возможность реабилитации старых идей, слабо развивавшихся из-за низкой технологичности и производительности компьютеров. Представляет интерес развитие исследовательского, эволюционного, когнитивного и адаптационного подходов к программированию, создающих перспективу рационального освоения реальных информационных ресурсов и компьютерного потенциала.

Исследовательский подход с учебно-игровым стилем профессионального, обучающего и любительского программирования может дать импульс изобретательности в совершенствовании технологии

программирования, не справившейся с кризисными явлениями на прежней элементной базе.

Эволюционный подход с мобильным стилем уточнения программ достаточно явно просматривается в концепции **объектно-ориентированного программирования**, постепенно перерастающего в субъектно-ориентированное и даже эго-ориентированное программирование. Повторное использование определений и наследование свойств объектов могут удлинить жизненный цикл отлаживаемых информационных обстановок, повысить надежность их функционирования и простоту применения. Когнитивный подход с интероперабельным стилем визуально-интерфейсной разработки открытых систем и использование новых аудио-видео средств и нестандартных устройств открывают пути активизации восприятия сложной информации и упрощения ее адекватной обработки.

Характерные свойства основных парадигм программирования представлены в Таблице 2.

Таблица 2.
Свойства парадигм программирования

Парадигма	**Ключевой концепт**	**Программа**	**Выполнение программы**	**Результат**
Императивная	Команда	Последова-тельность команд	Исполнение команд	Итоговое состояние памяти
Функциональная	Функция	Набор функций	Вычисление функций	Значение главной функции
Логическая	Предикат	Логические формулы	Логическое доказательство	Результат доказательства
Объектно-ориентированная	Объект	Набор классов объектов	Обмен сообщениями между объектами	Результирующее состояние объектов

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

2.3. Высокоуровневое программирование

Для императивного программирования характерно четкое разделение понятий «программа» и «данное» и учет в процессе обработки данных

средств контроля типов данных. Кроме того идеи структурного программирования налагают на стиль программирования ряд ограничений, способствующих удобству отладки программ:

- дисциплина логики управления с исключением переходов по меткам;
- минимизация использования глобальных переменных в пользу параметров процедур;
- полнота условий в ветвлениях, отказ от отсутствия «else»;
- однотипность результатов, полученных при прохождении через разные ветви.

Общий механизма интерпретации стандартной программы естественно представить как автомат с отдельными таблицами имен для переменных, значения которых подвержены изменениям, и для меток и процедур, определения которых неизменны. Наиболее распространенные языки программирования, такие как Фортран, Pascal, Си, Basic, придерживаются примерно такой семантики при слегка варьируемой строгости контроля типов данных. Семантика стандартных императивных языков допускает применение общей абстрактной машины, что объясняет существование многоязыковых систем программирования, поддерживающих общие библиотеки процедур, компонентов и модулей, а также интеграцию с ассемблером.

Рассмотрим основные конструкции языков императивного программирования и их краткое описание и представление с использованием псевдокода и блок-схем, и их воплощения на наиболее популярных языках программирования, таких как Pascal/Delphi, Си, Basic, Java . Псевдокод – это псевдоязык программирования, на синтаксис которого стандартов не существует. Псевдокод лишен несущественных для понимания сути алгоритма деталей, без которых невозможно обойтись при написании программ на реальных языках программирования. Единственная цель псевдокода – формализовать описание алгоритма. Задачи, решения которых описаны на псевдокоде, очень легко переносятся на любой язык программирования, поскольку псевдокод и есть язык программирования с той лишь разницей, что для него не существует компилятора, а единственным интерпретатором для него является человеческий мозг.

2.3.1.Выражения

Выражения не являются самостоятельными конструкциями языка программирования. Чаще всего, они являются аргументами оператора присваивания, условного оператора или цикла.

Математическое выражение

Назначение: описание выражения, результатом которого является числовое значение. В состав математического выражения могут входить

константы (конкретные числовые значения), переменные элементарных типов и функции. Практически все языки программирования поддерживают основные математические операции: + (сложение), - (вычитание), * (умножение), / (деление). Некоторые также поддерживают на базовом уровне возведение в степень (^), остаток от деления и другие. Более сложные операции доступны, как правило, в виде встроенных функций, например, sqrt(x) – квадратный корень или exp(x) – значение экспоненты. Естественно, что синтаксис математических выражений поддерживает скобки и унарный минус.

Пример математического выражения:

(вычисление длины стороны треугольника по теореме косинусов)
*SQRT(B^2 + C^2 - 2*B*C*COS(alpha))*

Логическое выражение

Назначение: Описание выражения, результатом которого является истина (true) или ложь (false). В состав логического выражения также могут входить константы, переменные, функции и скобки. Плюс ко всему, могут использоваться операции сравнения: «больше», «меньше», «равно», «неравно» и логические операции: «И» (AND), «ИЛИ» (OR) и «НЕ» (NOT).

Пример логического выражения:

(условие существования решения квадратного уравнения)
*(A=0) ИЛИ (C=0) ИЛИ ((B^2-4*A*C)>=0)*
Си, Java
*(A==0) || (C==0) || ((B*B-4*A*C)>=0)*
Pascal /Delphi, Visual Basic
*(A=0) or (C=0) or ((B*B-4*A*C)>=0)*

Строковое выражение

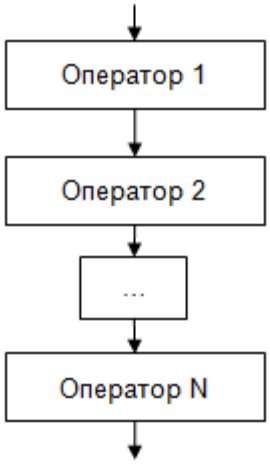
Назначение: манипуляция со строками. По виду напоминает математическое выражение, но результатом его является новая строка. Операция + (сложение) в контексте строкового выражения означает объединение (конкатенацию) двух и более строк.

2.3.2.Операторы

Оператор – это основная конструкция в языке программирования, которая представляет собой команду вычислительной машине выполнить некоторое действие или организует выполнение других операторов в определенном порядке.

Составной оператор

Назначение: организация нескольких операторов в один составной оператор. Используется в структурированных операторах.



Блок-схема составного оператора.

Псевдокод:

Начало

Список операторов

Конец

В языках Си, Java и некоторых Си-подобных языках составной оператор заключается в фигурные скобки {}. В языке Pascal /Delphi составной оператор начинается ключевым словом *begin* и заканчивается ключевым словом *end*, а в Visual Basic начало и конец составного оператора могут выглядеть по-разному, например *If/End If* или *Sub/End Sub*.

Объявление переменной

Назначение: объявление и выделение памяти под переменную заданного типа. В случае с объектами, объявление и выделение памяти может выполняться отдельно. В случае с переменными элементарных типов и экземплярами структур память под них выделяется сразу по факту объявления. Объявление переменной, помимо типа и имени, может быть дополнено инициализирующим значением и модификатором константы. Наличие модификатора константы запрещает изменять значение переменной после ее инициализации.

Псевдокод:

[КОНСТАНТА] имя_типа Имя_переменной [={инициализирующее_значение|выражение|функция}]

Примеры определения переменных:

ЦЕЛЫЙ A

СИМВОЛЬНЫЙ C=’A’

КОНСТАНТА ВЕЩЕСТВЕННЫЙ PI= 3.1415926535

Си

```
int A;
char C = 'A';
```

const double PI = 3.1415926535;

Java

int A;
char C = 'A';
final double PI = 3.1415926535;

Pascal /Delphi

var A: Integer;
var C: char = 'A';
const PI: Double = 3.1415926535;

Visual Basic

Dim A As Integer
Dim C As Char = "A"
Const PI As Double = 3.1415926535

Оператор присваивания

Назначение: изменение значения переменной. Аргументом оператора присваивания может быть математическое, логическое, строковое выражение или функция. Во многих языках программирования оператор присваивания можно совместить с объявлением переменной.

Псевдокод:

Имя переменной = {значение|выражение|функция}

Пример оператора присваивания:

Си, Java, Visual Basic

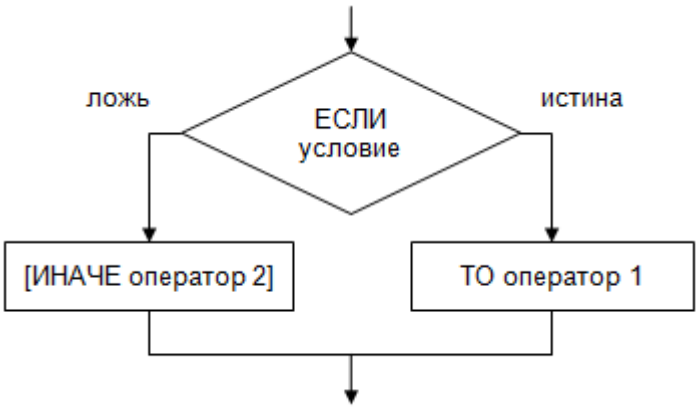
*C=sqrt(B*B+C*C-2*B*C*cos(alpha));*

Pascal /Delphi

*C:=sqrt(sqr(B)+sqr(C)-2*B*C*cos(alpha));*

Условный оператор или оператор IF

Назначение: организация ветвления хода вычислений в зависимости от условия. **ЕСЛИ** условие истинно, то выполняется основная часть условного оператора **ТО**. Также есть возможность определить альтернативный ход вычислений **ИНАЧЕ**, наличие которого не является обязательным.



Блок-схема условного оператора.

Псевдокод:

ЕСЛИ (Логическое выражение)
ТО Составной оператор
[ИНАЧЕ Составной оператор]

Пример условного оператора:

```
(Решение квадратного уравнения)
ЕСЛИ (b^2 >= 4 * a * c)
  ВЕЩЕСТВЕННЫЙ d = SQRT(b^2 - 4 * a * c)
  ВЕЩЕСТВЕННЫЙ x1 = (-b + d) / (2 * a)
  ВЕЩЕСТВЕННЫЙ x2 = (-b - d) / (2 * a)
ИНАЧЕ
  ВЫВОД('Решения нет!')
КОНЕЦ ЕСЛИ
```

Си, Java

```
if (b * b >= 4 * a * c)
{
  double d = Sqrt(b * b - 4 * a * c);
  double x1 = (-b + d) / (2 * a);
  double x2 = (-b - d) / (2 * a);
}
else //Решения нет!
```

Pascal /Delphi

```
program example;
 var d,x1,x2:real;
begin
 Writeln ("Введите a,b,c:");
 Read (a,b,c);
 if  b * b >= 4 * a * c
  then
    begin
      d:= Sqrt(b * b - 4 * a * c);
```

```
        x1:= (-b + d) / (2 * a);
        x2:= (-b - d) / (2 * a);
      end
    else Writeln("Решения нет!");
  end;
```

Visual Basic

```
  Sub example()
    If b * b >= 4 * a * c Then
      Dim d As Double = Math.Sqrt(b * b - 4 * a * c)
      Dim x1 As Double = (-b + d) / (2 * a)
      Dim x2 As Double = (-b - d) / (2 * a)
    Else 'Решения нет!
    End If
  End Sub
```

Оператор выбора или оператор SELECT (SWITCH)

Назначение: организация ветвления хода вычислений в соответствии со значением критерия выбора. Оператор выбора предполагает передачу управления одному из вариантов выбора, с которым ассоциировано значение, равное заданному критерию.

Псевдокод:

```
ВЫБОР (Критерий)
ВАРИАНТ Значение 1: Составной оператор
ВАРИАНТ Значение 2: Составной оператор
...
ВАРИАНТ Значение N: Составной оператор
[ИНАЧЕ: Составной оператор]
```

Пример:

```
ВЫБОР (Key)
ВАРИАНТ 1: Вывод("1")
ВАРИАНТ 2: Вывод("2")
ИНАЧЕ: Вывод("Key<>1 И Key<>2")
```

Си, Java

```
switch (key)
{
  case 1://key == 1
       break;
  case 2://key == 2
       break;
  default://key != 1 && key != 2
       break;
```

```
}
```

Pascal /Delphi

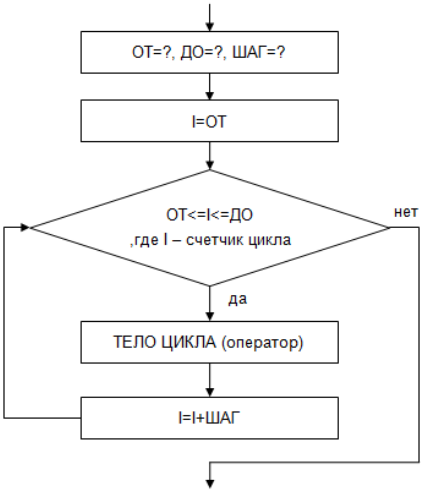
```
Case key of
  1:  WriteLn("1");
  2:  WriteLn("2");
 Else WriteLn("key<>1 and key<>2");
End;
```

Visual Basic

```
Select Case key
 Case 1 '1
 Case 2 '2
 Case Else 'key<>1 and key<>2
End Select
```

Цикл с определенным количеством итераций или цикл FOR

Назначение: организация определенного заголовком цикла числа итераций. Чаще все число итераций задается начальным и конечным значением счетчика цикла и шагом его изменения с каждой итерацией. Счетчик цикла – числовая переменная, значение которой может использоваться в теле цикла, например, в качестве индекса текущего элемента в массиве в задачах последовательной обработки массивов. Явное определение величины шага обязательным не является. Значением шага по умолчанию, обычно, является единица.



Блок-схема цикла FOR.

Псевдокод:

ЦИКЛ Счетчик={Начальное значение|Математическое выражение} ДО {Конечное значение|Математическое выражение} [ШАГ {Величина шага|Математическое выражение}]

Составной оператор

СЛЕДУЮЩЕЕ ЗНАЧЕНИЕ

Пример:

(Вычисление факториала числа N)

```
ЦЕЛЫЙ N=10
ЦЕЛЫЙ F=1
ЦИКЛ I=2 ДО N
 F=F*I
СЛЕДУЮЩЕЕ ЗНАЧЕНИЕ
```

Си, Java

```
int N=10;
int F=1;
for (int i = 2; i <= N; i++)
   {
    F = F * i;
   }
```

Pascal /Delphi

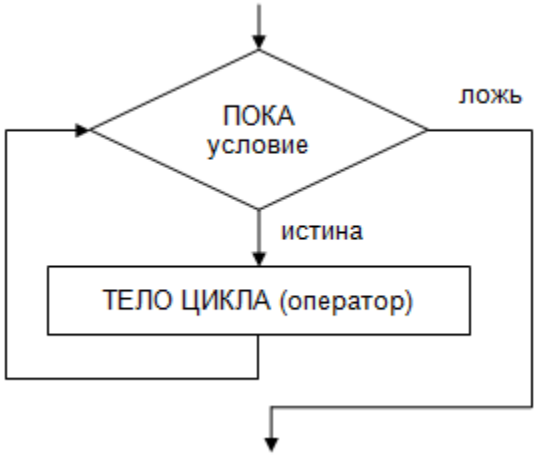
```
program fact;
 var N, I:Integer; F:LongInt;
begin
  F:=1;
 Writeln ("Введите N:");
 Read (N);
 for i:=2 to N do
  begin
   F:=F*I;
  end;
end;
```

Visual Basic

```
Dim N As Integer = 10
Dim F As Integer = 1
For I As Integer = 2 To N
   F = F * I
Next
```

Цикл с предусловием или цикл WHILE

Назначение: организация повторения (зацикливания) определенной части программы. Условием выполнения тела цикла является истинность заданного логического выражения. Цикл с предусловием характерен тем, что сначала проверяется логическое выражение и только потом выполняется тело цикла. Таким образом, тело цикла с предусловием имеет все шансы не выполниться ни одного раза.



Блок-схема цикла с предусловием.

Псевдокод:

ПОКА (Логическое выражение)
Составной оператор
КОНЕЦ ЦИКЛА ПОКА

Пример:

(Вычисление факториала числа N)
ЦЕЛЫЙ N=10
ЦЕЛЫЙ F=1
ЦЕЛЫЙ I=1
ПОКА I<N
I=I+1
*F=F*I*
КОНЕЦ ЦИКЛА ПОКА

Си, Java

```
int N=10;
int F=1;
int I=1;
while (I<N)
{
  I++;
  F=F*I;
}
```

Pascal /Delphi

```
program fact;
 var N, I:Integer; F:LongInt;
begin
 Writeln ("Введите N:");
 Read (N);
 F:=1;
 I:=1;
 while I<N do
```

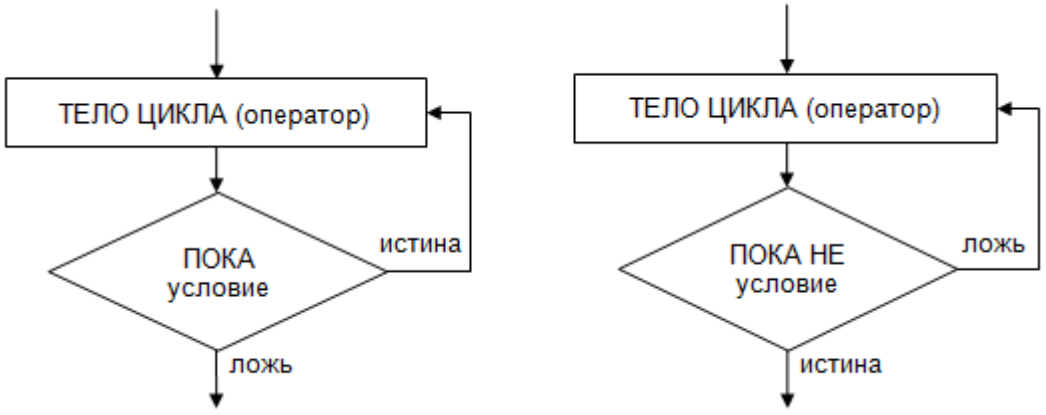
```
  begin
   I:=I+1;
   F:=F*I;
  end;
end;
```

Visual Basic

```
Dim N As Integer = 10
Dim F As Integer = 1
Dim I As Integer = 1
While I < N
    I = I + 1
    F = F * I
End While
```

Цикл с постусловием или цикл DO (UNTIL)

Назначение: все аналогично циклу с предусловием с той лишь разницей, что сначала всегда выполняется тело цикла, а потом проверяется условие повторного его выполнения. Цикл с постусловием выполняется минимум один раз! С помощью цикла с постусловием, к примеру, программируют всем известную задачу «угадай число», критерием завершения цикла в которой является факт совпадения введенного пользователем варианта с «загаданным» машиной значением. Существует две разновидности цикла с постусловием: цикл выполняется **ПОКА** условие истинно или цикл выполняется **ПОКА НЕ** условие истинно, т.е. цикл завершается, как только условие становится истинным и выполняется, пока оно ложно.



Блок-схема цикла с постусловием.

Псевдокод:

```
ВЫПОЛНЯТЬ
Составной оператор
ПОКА (Логическое выражение)
или
ВЫПОЛНЯТЬ
Составной оператор
```

ПОКА НЕ (Логическое выражение)

Пример:
(Вычисление факториала числа N)

```
ЦЕЛЫЙ N=10
ЦЕЛЫЙ F=1
ЦЕЛЫЙ I=1
ВЫПОЛНЯТЬ
 I=I+1
 F=F*I
ПОКА I<N
```

или

```
ЦЕЛЫЙ N=10
ЦЕЛЫЙ F=1
ЦЕЛЫЙ I=1
ВЫПОЛНЯТЬ
 I=I+1
 F=F*I
ПОКА НЕ I>=N
```

Си, Java

```
int N=10;
int F=1;
int I=1;
do
{
  I++;
  F = F * I;
} while (I < N);
```

Pascal /Delphi

```
program fact;
 var N, I:Integer; F:LongInt;
begin
 Writeln ("Введите N:");
 Read (N);
 F:=1;
 I:=1;
 repeat
  I:=I+1;
  F:=F*I;
 until I>=N
end;
```

Visual Basic

```
Dim N As Integer = 10
Dim F As Integer = 1
Dim I As Integer = 1
Do
   I = I + 1
   F = F * I
Loop While I < N
```

или

```
Dim N As Integer = 10
Dim F As Integer = 1
Dim I As Integer = 1
Do
   I = I + 1
   F = F * I
Loop Until I >= N
```

Выход из подпрограммы или оператор RETURN/EXIT

Назначение: досрочный выход из процедуры или функции. В языках Си, Java оператор *return* в контексте процедуры просто досрочно завершает ее работу, а в контексте функции еще и определяет ее значение: *return result_value*;. В языках Pascal /Delphi оператор *exit* просто завершает выполнение и процедуры и функции, а значение функции присваивается по ее имени: *function_name:=result_value*;.

Выход из составного оператора или операторы BREAK и CONTINUE

Назначение: досрочный выход из составного оператора, например, досрочное завершение цикла. В языках Си, и Java и Pascal /Delphi оператор *break* позволяет завершить работу текущего цикла или оператора *switch*. В Visual Basic для этих целей используются инструкции *Exit For, Exit While, Exit Do* и т.п. В языках Си, Java и Pascal есть еще и оператор *continue*, который в отличие от *break* не выходит совсем из цикла, а завершает только текущую итерацию, т.е. переходит к анализу заголовка цикла.

Метка и переход к метке или операторы LABEL и GOTO

Назначение: выделение определенного места в программе (установка метки) с возможностью безусловного перехода к этой метке из другой части программы. Переход чаще всего возможен только в границах одной процедуры или функции. *Есть мнение, что использование меток – это плохой стиль в программировании*. Чаще всего оператором безусловного перехода к метке является команда *goto*.

Псевдокод:
(Пример установки метки и перехода к метке)

Имя_метки:
…
ПЕРЕХОД Имя_метки

В большинстве языков программирования работа с метками будет выглядеть примерно так:

[label] Имя_метки:
…
goto Имя_метки

2.3.3. Декомпозиция программного кода

Процедуры и функции являются ключевыми конструкциями в процедурном программировании, поскольку позволяют разбить весь программный код на фрагменты, решающие локальные задачи и пригодные для многократного использования в процессе основной обработки данных.

Процедура

Назначение: выделить часть программы, пригодную для повторного использования. Процедурам дают имена, отражающие суть того, что они выполняют. Процедуры могут иметь набор входных параметров – аргументов процедуры. Для вызова процедуры в программе достаточно указать ее имя и список значений аргументов.

Псевдокод:

ПРОЦЕДУРА Имя (СПИСОК ПАРАМЕТРОВ)
Составной оператор
[КОНЕЦ ПРОЦЕДУРЫ],

где СПИСОК ПАРАМЕТРОВ – это список объявлений формальных параметров (для каждого аргумента процедуры), разделенных запятыми.

Функция

Назначение: аналогично процедуре с той разницей, что функция возвращает результат определенного для нее типа. Вызов функции может быть не только частью составного оператора, но и аргументом оператора присваивания, частью математического, логического или строкового выражения.

Псевдокод:

ФУНКЦИЯ Имя (СПИСОК ПАРАМЕТРОВ) ТИП
Составной оператор

[КОНЕЦ ФУНКЦИИ]

Рассмотрим, как используются функции в императивном программировании на примере задачи вычисления последовательности простых чисел.

Си, Java

```
public class Application
{
    public static void Main()
    {
        Console.Write("Формируем последовательность простых чисел не более N = ");
        int _MaxValue = 0;
        try
        {
           _MaxValue = System.Convert.ToInt32(Console.ReadLine());
        }
        catch (Exception ex)
        {
           Console.WriteLine("Произошла ошибка ввода: "+ex.Message);
              Console.WriteLine("Нажмите любую кнопку для выхода...");
              Console.ReadKey();
              return;
        }
          //Перебираем все числа от 2, до _MaxValue
        for (int i = 2; i <= _MaxValue; i++)
        {
           if (i == 2 || IsPrime(i))
           {
              if (i > 2)
                 Console.Write(",");

              Console.Write(i.ToString());
           }
        }
        Console.WriteLine();
        Console.WriteLine("Нажмите любую кнопку для выхода...");
        Console.ReadKey();
    }
    //Определение, является ли число d простым?
    static bool IsPrime(int d)
    {
        int _sqrt_int = (int)Math.Sqrt(d); //Целая часть от квадратного корня из d
```

```
        for (int i = 2; i <= _sqrt_int; i++)
           if ((d % i) == 0) //Проверка: остаток от деления d/i==0?
              return false;

        return true;
     }
}
```

Pascal /Delphi

```
program PrimeNumbers;
  uses   crt;
  var   MaxValue, i : Integer;
  function IsPrime(d:Integer):boolean
    var sqrt_int,i : Integer;
 begin
  sqrt_int:=round(sqrt(d));
  for i:= 2 to sqrt_int do
     if (d mod i = 0) then
       begin
         IsPrime:=false;
        Exit;
       end;
  IsPrime:=true;
 end;

begin
Write('N = ');
 Read (MaxValue);
 for i:=1 to MaxValue do
    if IsPrime(i) then writeln(i);
 ReadLn;
end.
```

2.3.4.Типы данных

Типы данных – это то, без чего любой вид программирования не возможен в принципе. Эти конструкции предназначены для описания формата обрабатываемой программой информации: от элементарных данных до сложноорганизованных структур.

Элементарные (базовые) типы данных

Назначение: описание неделимой части информации, например, числовой переменной, символа или результата логического выражения. Большинство языков программирования поддерживают одни и те же элементарные или базовые типы: целочисленный, вещественный, символьный, логический, дата и время. Диапазон значений для

целочисленного и вещественного типа определяется количеством занимаемых им байт. Один байт – это объем информации, способный хранить 256 различных значений и состоящий из 8 бит. Бит, в свою очередь, способен хранить лишь два значения: 0 и 1. Комбинаций из 8 нулей и единиц ровно 256 (2^8). В этом контексте целочисленные и вещественные типы делятся на подтипы, которые в каждом языке программирования могут обозначаться по-разному. Также, к элементарным типам часто относят **строковый тип** данных (последовательность символов - строка), который неделимым не является, но также относится к базовым типам. Ниже приведена таблица с основными элементарными типами наиболее известных языков программирования.

Таблица 3.

Основные типы данных

Описание типа	**Размер (биты)**	**Диапазон**	**Си**	**VB**	**Java**	**Pascal /Delphi**
Байт со знаком	8	от -128 до 127	sbyte	SByte	byte	shortint
Байт без знака	8	от 0 до 255	byte	Byte	-	byte
Короткое целое число со знаком	16	от -32 768 до 32 767	short	Short	short	integer
Короткое целое число без знака	16	от 0 до 65 535	ushort	UShort	-	word
Среднее целое число со знаком	32	от -2147483648 до 2147483647	int	Integer	int	longint
Среднее целое число без знака	32	от 0 до 4294967295	uint	UInteger	-	-
Длинное целое число со знаком	64	от -9223372036854775808 до 9223372036854775807	long	Long	long	comp
Длинное целое число без знака	64	от 0 до 18446744073709551615	ulong	Ulong	-	-
Вещественное число одинарной точности с плавающей запятой	32	$\pm 1{,}5*10^{-45}$ до $\pm 3{,}4*10^{33}$	float	Single	float	single
Вещественное число двойной точности с плавающей запятой	64	$\pm 5*10^{-324}$ до $\pm 1{,}7*10^{306}$	double	Double	double	double

Символ (8 бит)	8	ASCII	-	-	-	char
Символ (16 бит)	16	UNICODE	char	Char	char	-
Логический тип	8	{true, false}	bool	Boolean	boolean	boolean
Строка	-	-	string	String	String	string
Пустой тип	-	-	void	-	void	-

Структура

Назначение: описание сложных (составных) типов данных. Структура может в себе содержать определения переменных элементарных типов (числовые, символьные, логические), а также определения массивов, записей, коллекций, классов, других структур и т.д. Все составные части структуры называют *полями структуры*. Экземпляры структур создаются точно так же, как экземпляры простых типов. Доступ к любому полю структуры, как правило, осуществляется по имени соответствующей переменной – экземпляра структуры и имени поля, разделенных между собой точкой.

Псевдокод:

СТРУКТУРА Имя
Объявление поля 1
Объявление поля 2
…
Объявление поля N
КОНЕЦ СТРУКТУРЫ

Pascal /Delphi

```
{объявление структуры Record (запись)}
Type
   Student = Record
    Fam, name: String [20]; {объявление полей записи}
    Pol: (men, women);
    Bday, dinp: Word;
    Fac: String [15];
  End;
```

Дальнейшее развитие императивной парадигмы программирования отражает изменение круга лиц, заинтересованных в применении информационных систем. Формирование экстенсивных подходов к программированию – естественная реакция на радикальное улучшение эксплуатационных характеристик оборудования и компьютерных сетей. Появилась почва для обновления подходов к программированию, а также возможность реабилитации старых идей, слабо развивавшихся из-за низкой технологичности и производительности ЭВМ. Представляет интерес развитие исследовательского, эволюционного, когнитивного и адаптационного подходов к программированию, создающих перспективу

рационального освоения реальных информационных ресурсов и компьютерного потенциала.

2.4. Функциональное программирование

2.4.1. Введение в функциональное программирование

Несмотря на богатую историю развития традиционных языков программирования, они несут на себе родовую печать фон-неймановской архитектуры, использующую императивную парадигму. В то же время передовые программисты второй половины 20-го века считали, что программирование может и должно быть интеллектуально благодарной деятельностью; что хороший язык программирования является мощным инструментом абстракций - инструментом, пригодным для организации, выражения, экспериментирования и даже общения в процессе интеллектуальной деятельности; что взгляд на программирование как на кодирование - рутинный, интеллектуально тривиальный, но очень трудоемкий и утомительный заключительный этап решения задачи при использовании компьютеров, - возможно, и является глубинной причиной явления, называемого кризисом программирования.

Иными словами, главная мысль здесь заключается в том, чтобы не человека вынудить при создании программ мыслить в категориях машины, а компьютер заставить понимать язык с более высоким уровнем абстракции, удобный для формулировки и планирования решения задач человеком. Более того, в математике известны такие математические формализации понятия алгоритма и модели вычислений, как рекурсивные функции, нормальные алгоритмы Маркова, лямбда-исчисление, не привязанные к понятиям состояния, шага программы и записи значения в ячейку, характерным для машины Тьюринга. Они могут служить мощным теоретическим фундаментом при создании языков программирования, которые можно с определенной натяжкой назвать нетрадиционными.

Функциональное программирование - одна из основных нетрадиционных парадигм программирования, называемая также аппликативной. Причиной этого является то, что в ней вычислительный процесс сводится к применению функции к аргументу, а теоретическим базисом служит λ-исчисление (лямбда-исчисление). В качестве важного преимущества данной модели вычислений можно выделить, в частности, отсутствие так называемых побочных эффектов, что означает, что результат выполнения программы полностью определяется ею самой и исходными данными. Пионером в этой области стал Джон Мак-Карти, который в 1958 году во время работы в Массачусетском технологическом институте разработал язык программирования LISP (от англ. LISt Processing — обработка списков). Это, вероятно, первый из языков, который основывался на серьезном теоретическом фундаменте и пытался поднять практику

программирования до уровня концепций, а не опустить концепции до уровня существовавшей на момент создания языка практики [12]. В русскоязычной литературе используются различные варианты написания названия — LISP, Lisp, ЛИСП, мы в основном будем называть его Лисп. Лисп был изобретен как формализм для рассуждений об определенном типе логических выражений, называемых уравнениями рекурсии, как о модели вычислений.

2.4.2. Язык программирования Лисп. Общие сведения

Программа на Лиспе состоит из выражений. При этом каждое выражение является либо атомом, либо списком. Здесь атом — некоторый объект, представленный последовательностью алфавитно-цифровых символов. Примеры атомов:

7
A
ПРИВЕТ
S17

Списки — базовая составная структура данных языка. Под списком подразумевается упорядоченное множество некоторых элементов, например: *((КОРОЛЬ ФЕРЗЬ) ((ЛАДЬЯ) (СЛОН)) ПЕШКА)*

Элементы могут быть как атомами, так и списками. Более того, в отличие от списков в ряде других языков программирования, в списках в Лиспе элементы могут иметь разную природу, что видно и в приведенном примере. Внутренняя часть списка заключается в круглые скобки, элементы обычно разделяются пробелами. Важной особенностью языка Лисп является то, что сама программа в нем представляет собой список. Это дает возможность программам выполнять довольно интересные манипуляции над самими собой, а также передавать функции в качестве аргументов функций, иными словами, использовать функции высших порядков. Несмотря на то что Лисп - язык, созданный почти сразу после Фортрана, на заре эпохи языков программирования высокого уровня, в нем воплощены несколько идей, лишь значительно позднее получивших распространение в наиболее широко используемых языках. Среди них помимо функций высших порядков:

- ссылочная организация памяти;
- удобство обработки символьных данных;
- сборка мусора (автоматическое освобождение системой не используемой более памяти);
- использование замыканий (локально определяемых в некотором контексте функций);
- функции проверки (RTTI) типа значения в ходе выполнения программы.

К достоинствам Лиспа относят точность, определенность, лаконичность (на многих неофитов производит впечатление краткость

программ на Лиспе, выполняющих те же действия, что и значительно более длинные программы, например, на Паскале). Он является хорошим средством для представления древовидных структур. Интересно, что, начиная с первых реализаций, транслятор Лиспа принято разрабатывать методом раскрутки (простейшие операции реализуются в машинном коде, а остальные - на самом языке программирования). В составе системы сразу предусматривались и интерпретатор, и компилятор. Оба эти инструмента были весьма точно описаны на самом Лиспе, причем основной объем описаний не превышал пары страниц. Все это привело к тому, что к середине 1970-х годов на Лиспе решались наиболее сложные практические задачи, связанные с принятием решений, построением систем искусственного интеллекта и пр. До сих пор Лисп остается языком № 1 в мире при создании интеллектуальных систем.

Некоторые атомы являются именами функций. Заострим внимание на том, что в соответствии с принятой в лямбда-исчислении, являющемся теоретической базой Лиспа, записью принято, что применение функции к аргументам записывается как список

(<функция> <арг1> <арг2> ... <аргn>)

в отличие от принятой в классической математике и других языках программирования записи

<функция> (<арг1> <арг2> ... <аргn>).

Чтобы определить, как воспринимать атом - как константу или как имя функции, значение которой надо вычислить, в Лиспе используется специальное ключевое слово QUOTE. Если перед атомом идет QUOTE, Лисп-система воспринимает его как не требующий вычисления. Часто вместо QUOTE перед именем, которое не нужно немедленно вычислять ставится апостроф: ‘L.

Некоторые символы имеют специальное назначение. Например, символ T означает логическую истину, NIL наряду с пустым списком - ложь.

Используя пустой список, структуру данных список можно определить рекурсивно. Итак, списком является или пустой список, или некоторый элемент, называемый головой списка, за которым следует сам список (так называемый хвост).

2.4.3. Функции обработки списков

Взятие головы и хвоста некоторого списка является парой наиболее известных встроенных в Лисп функций - CAR и CDR. Любопытно, что свои названия они получили от двух ассемблерных команд ЭВМ IBM 704 - взятия значения по адресу и взятия значения из памяти с автоматическим декрементом (уменьшением на единицу) адреса. Приведенные далее примеры иллюстрируют применение CAR и CDR (справа после стрелки записан результат вычисления функции, немедленно выдаваемый Лисп-

интерпретатором после ввода соответствующей строки в диалоговом режиме):

(CAR '(A B C)) → A
(CAR '(A)) → A
(CAR 'A) → ошибка, A — атом, а не список
(CAR '(NIL)) → NIL
(CAR NIL) → NIL
(CAR '(NIL A)) → NIL

(CDR '(A B C)) → (B C)
(CDR '(A)) → NIL
(CDR 'A) → ошибка, A — атом, а не список
(CDR '(A (B C))) → ((B C))
(CDR '(NIL A)) → (A)
(CDR ())→ NIL

Для удобства головой пустого списка считается NIL. Поскольку в программах на Лиспе довольно часто можно встретить цепочки последовательных вызовов функций CAR и CDR, в языке разрешается использовать удобные сокращенные обозначения вида CADDR, CADR и пр.:
(CADR S) ; то же самое, что (CAR (CDR S))
(CADR '(A B C)) ; получим B
(CADDAR S) ; то же самое, что (CAR(CDR (CDR (CAR S))))

Символы в программе на Лиспе, стоящие после точки с запятой и до конца строки, считаются комментарием. Глядя на примеры, можно обратить внимание еще на одну особенность Лиспа. Это - язык без строгой системы типов. Из-за этого при вычислении функции она может быть применена к аргументам различного типа, что, в свою очередь, может привести к ошибке. Зато в Лиспе имеются функции проверки типа аргумента при выполнении программы. Некоторые функции Лиспа являются предикатными, иными словами, возвращают в качестве значения или ИСТИНУ - T, или ЛОЖЬ в виде NIL. Именно к ним относятся такие функции, как ATOM, проверяющая, является ли аргумент атомом, и NULL, определяющая, не является ли аргумент пустым списком:

(ATOM 'A) → T
(ATOM '(A B C)) → NIL
(ATOM ())→ T ; пустой список атомарен
(NULL NIL) → T
(NULL 'A) → NIL
(NULL (CDR '(ATOM))) → T
(NULL T) → NIL

На основании этих примеров можно сделать вывод: функцию NULL можно использовать и как логическое отрицание. Впрочем, для этих же целей в Лиспе существует и предикатная функция NOT. Помимо нее есть и функции AND (логическое И, если среди последующих аргументов хоть раз встретится NIL, значением будет NIL, в противном случае — значение последнего из аргументов) и OR (логическое ИЛИ, если среди аргументов

встречается выражение со значением, отличным от NIL, возвращается оно, иначе — NIL).

Функция CONS противоположна по действию CAR и CDR, она конструирует список из двух своих аргументов, которые должны быть элементом — будущей головой и списком — хвостом получаемого списка:

(CONS 'A '(B C)) → (A B C)
(CONS 'A NIL) → (A)
(CONS '(A B) '(C)) → ((A B) C)
(CONS (CAR '(A B C)) (CDR '(A B C))) → (A B C)

Помимо CONS списки создают функции LIST и APPEND. APPEND получает на вход набор списков — в общем случае произвольное число, что является, в частности, иллюстрацией допустимости в Лиспе функций произвольного количества аргументов, — и формирует из их элементов единый список, удаляя внешние скобки для каждого из исходных списков:

(APPEND '(A) NIL '((B)(C))) → (A (B)(C))

В отличие от этого функция LIST формирует общий список из своих аргументов (которыми могут быть и атомы), оставляя скобки:

(LIST '(A) NIL '((B)(C))) → ((A) ((B)(C))) (LIST '(A) NIL) → ((A) NIL)

Функция SETQ позволяет связать некоторый атом со значением выражения, после чего он становится его именем:

(SETQ G '(A B(C D)E))

Теперь можно использовать это имя вместо исходного выражения:

(CAR G) → A

Важно подчеркнуть, что не следует использовать SETQ в качестве аналога присваивания в императивных языках программирования. Разрушающее присваивание, когда в ходе исполнения программы переменная изменяет свое значение, нехарактерно для аппликативного программирования. Вместо этого лучше думать, что SETQ в некотором смысле наклеивает ярлык на выражение. Для работы со списками в Лиспе есть еще ряд полезных функций, назовем лишь некоторые из них.

LENGTH вычисляет длину списка (количество элементов, число «братьев первого уровня») и возвращает целое число:

(LENGTH '((AB) C (D))) → 3

REVERSE обращает список, записывая элементы в обратном порядке:

(REVERSE '(A (BC) D)) → (D (BC) A)

Для определения того, является ли нечто «братом» первого уровня в определенном списке, используется предикатная функция MEMBER:

(SETQ M '(AB (CD) E)) (MEMBER 'F M) → NIL
(MEMBER 'C M) → NIL
(MEMBER 'B M) → T

Функция SUBST позволяет выполнить подстановку и имеет три аргумента. Результатом выполнения функции является значение третьего аргумента, в котором все вхождения второго аргумента заменены первым, например:

(SUBST 'A 'B '(A B C)) → (A A C)

Функция RPLACA имеет два аргумента. Первый из них — список. В качестве результата функция возвращает его с заменой первого элемента вторым :

(SETQ G '(A B C)) (RPLACA (CDR G) 'D) → (D C)

Функция RPLACD похожа на функцию RPLACA, но при ее использовании и второй аргумент должен быть списком. Он заменяет собой в результирующем списке хвост списка, являвшегося аргументом:

(RPLACD G '(1 2 3)) → (A 1 2 3)

2.4.4. Функции для работы с числами

В Лисп встроен набор функций для обработки чисел. Как нужно их использовать, ясно из следующих примеров:

(+ 2 3) → 5
(+ 1 2 3 4 5) → 15
(- 5 2) → 3
(4 5) → 20*
(/ 8 2) → 4
(MAX 5 8 7 6) → 8
(MIN 5 8 7 6) → 5
(SQRT 16) → 4 ; квадратный корень из числа
(ABS -3) → 3 ; модуль числа

Функция REM позволяет найти остаток от деления:

(REM 7 2) → 1

Для сравнения чисел можно использовать функции <, >, >=, <=:

(> 2 3) → NIL
(>= 2 3) → NIL
(>= 3 3) → T

Для проверки на равенство (не только чисел) в Лиспе применяются функции EQUAL и EQ:

(EQ 'A 'B) → NIL
(EQ 'A 'A) → T
(EQ 'A (CAR '(A B)) → T
(EQ () NIL) → T

Следует помнить, что предикат EQ применим лишь к атомарным аргументам, и не может быть использован для списков. Отличным в этом плане от него и более общим для списков является предикат EQUAL, позволяющий сравнивать два списка:

(EQUAL 'A 'A) → T (EQUAL '(A B) '(A B)) → T

Предикатные функции NUMBERP и ZEROP проверяют свои аргументы. NUMBERP возвращает истину, если аргумент - число, ZEROP - если аргумент равен нулю (применим лишь числам!):

(NUMBERP 1) → T
(NUMBERP 'A) → NIL
(ZEROP 0) → T
(ZEROP 11) → NIL

2.4.5. Функции высших порядков

Весьма интересными возможностями обладают функции MAPCAR и APPLY. Функция MAPCAR применяет свой первый аргумент, который должен являться унарной функцией, ко всем элементам второго аргумента, как в следующем примере:

(MAPCAR 'ABS '(1 -2 3 -4 5)) → (1 2 3 4 5)

Функция APPLY применяет свой первый аргумент — функцию — ко второму:

(SETQ A '(4 5 8 1)) (APPLY '+ A) → 18

С помощью APPLY и MAPCAR, являющихся функциями высших порядков, в программах на Лиспе можно легко и элегантно переносить действия любых функций на элементы списка.

2.4.6. Написание программ на языке Лисп

Написание программы на Лиспе сводится к определению новых функций. Сделать это позволяет особая функция DEFUN следующего вида:

(DEFUN <атом-имя> (<a_1>...<a_n>) <выражение-тело>)

Здесь первый аргумент — имя создаваемой функции, затем идет список, содержащий формальные параметры, и после завершающей список скобки — выражение, вычисление которого будет приниматься в качестве результата вычисления данной функции. Определим для примера функцию вычисления квадрата суммы двух чисел:

(DEFUN SUMKVAD (X Y) (+ (X X) (* Y Y)))*

Обратите внимание на стиль написания. Аналогично программам на Си, программы на Лиспе можно сделать более удобными для восприятия человеком, применяя «лесенку», соответствующую логической структуре программы. Теперь мы можем использовать построенную функцию:

(SUMKVAD 2 5) → 29

Однако эта функция не включает возможность обработки разных случаев при выполнении программы. В императивных языках программирования подобная обработка обычно реализуется с помощью операторов ЕСЛИ - ТО - ИНАЧЕ или операторов выбора. Лисп дает возможность реализации обработки разных ситуаций, в том числе с помощью мощной конструкции COND. Это функция, записываемая особым образом:

(COND (<t_1> <v_1>)
(<t_2> <v_2>)
.
.
.
(<t_n> <v_n>)

)

При ее использовании применяются пары (<t_i> <v_i>). Значение COND вычисляется по следующей схеме. Сначала вычисляется значение выражения t_1. Если оно отличается от NIL, вычисляется значение выражения v_1 и возвращается в качестве результата выполнения всей функции COND. Если результат вычисления t_1 — NIL, вычисляется значение t_2. Если оно отлично от NIL, вычисляется и возвращается в качестве итогового результата значение v_2 и т. д. Если все t_i будут иметь значение NIL, оно будет окончательным значением COND. Некоторые выражения v_i могут отсутствовать. В этом случае, если t_i будет первым отличающимся от NIL, в качестве значения COND будет возвращено это значение. В качестве t_i обычно используют предикатные функции, возвращающие значение ИСТИНА или ЛОЖЬ. Часто в последней паре в качестве t_n используют просто константу T, чтобы, если не сработала ни одна из вышележащих пар, в любом случае нечто выполнить — аналог *else* или *default* в языке Си. Следующая программа иллюстрирует использование COND. Это аналог подобной программы на Паскале, переводящей оценку, выраженную в баллах, в ее словесное обозначение. Кроме того, программа замечательно иллюстрирует возможности и особенности Лиспа, связанные с обработкой символьной информации и нестрогой динамической типизацией, — она производит и обратное преобразование словесного выражения оценки в баллы:

```
; функция "Оценка"
; 2- неуд/ 3- уд/ 4- хор/ 5- отл
(DEFUN MARK (A)
 (COND
((EQUAL A 5) 'ОТЛИЧНО)
((EQUAL A 4) 'ХОРОШО)
((EQUAL A 3) 'УДОВЛЕТВОРИТЕЛЬНО)
((EQUAL A 2) 'НЕУДОВЛЕТВОРИТЕЛЬНО)
; перевод из строки в баллы
((EQUAL A 'ОТЛИЧНО) 5)
((EQUAL A 'ХОРОШО) 4)
((EQUAL A 'УДОВЛЕТВОРИТЕЛЬНО) 3)
((EQUAL A 'НЕУДОВЛЕТВОРИТЕЛЬНО) 2)
(T 'НЕПОНЯТНО)
 )
)
```

Итак, функция COND применяется как аналог условного оператора и оператора выбора императивных языков программирования. Что, если в программе на Лиспе необходимо неоднократно выполнить некоторые действия? В императивном языке для этого наиболее часто применяется итерация, реализуемая с помощью того или иного оператора цикла. В чистом функциональном программировании циклы не используются. Возникает законный вопрос: что же применяется вместо них? Базовый механизм повторения в функциональной парадигме программирования - рекурсия. Напомним, что она сводится к вызову, прямому или косвенному, функции из

самой этой функции. Использование рекурсии при программировании требует определенной перестройки мышления. Необходимо забежать вперед и представить, что мы уже имеем функцию, которая выполняет нужное нам, но не совсем для полной совокупности исходных данных - например, за исключением одного элемента списка. Далее нужно представить, как мы можем использовать ее для решения всей задачи. Кроме этого, при написании рекурсивных программ нужно помнить о необходимости избегать бесконечного повторения. В той или иной ситуации нужно остановить рекурсию. Для этого следует использовать соответствующее условие (например, с помощью функции COND языка Лисп). При обработке списков таким условием часто будет проверка входного списка на пустоту.

Рассмотрим в качестве примера программу подсчета длины списка. Подобная функция, естественно, имеется в стандартной поставке Лиспа, но мы ее повторим в учебных целях:

```
;подсчет длины списка
(DEFUN DLINA (S)
(COND
((NULL S) 0)
(T (+ 1 (DLINA (CDR S))))
)
)
```

Поскольку для подсчета количества элементов в списке нужно их перебрать, а их несколько, нам потребуется повторение одних и тех же действий. Следовательно, программа будет рекурсивной. Что можно использовать для останова рекурсии, какое условие? Видимо, в данном случае это будет ситуация, когда на вход поступает пустой список. Проверка этого организуется с помощью функций COND и NULL. В этом случае дальнейшие вычисления не проводятся, а в качестве результата возвращается нуль. Далее длину всего списка можно определить путем прибавления единицы к результату вычисления длины списка без одного элемента (например, головы). Рекурсивные программы на Лиспе писать легко и просто за счет того, в частности, что базовая структура данных языка — список — рекурсивна изначально, по своей природе.

Рассмотрим еще один пример — программу, обращающую список, иными словами, записывающую элементы исходного списка в обратном порядке:

```
(DEFUN ZERKALO (S)
    (COND
        ((NULL S) S)
        (T (APPEND (ZERKALO (CDR S))(LIST (CAR S))))
    )
)
```

Программа подобно предыдущей будет являться рекурсивной в силу необходимости обработки нескольких элементов исходного списка. Останов рекурсии производится в случае наличия на входе пустого списка — его можно считать собственным обращением. Результат получаем приписыванием первого элемента исходного списка к обращенной остальной

части этого списка. Поскольку все аргументы APPEND должны являться списками, используем функцию LIST для преобразования первого элемента исходного списка в список из одного этого элемента. Следующая программа решает задачу генерации числовой последовательности, начиная с заданного числа, по закону

$$u_{n+1} = \begin{cases} u_n / 2 & \text{если } u_n \text{ четное} \\ 3u_n + 1 & \text{иначе} \end{cases}$$

```
; PIMP — ГЕНЕРАЦИЯ ЧИСЛОВОЙ ПОСЛЕДОВАТЕЛЬНОСТИ
(DEFUN PIMP (U)
(PRINT U)
 (COND
((EQUAL U 1) NIL);останов рекурсии
((ZEROP (REM U 2)) (PIMP (/ U 2)))
(T (PIMP (+ 1 (* U 3))))
 )
 )
```

В программе используется еще одна функция Лиспа - PRINT, печатающая значение своего аргумента. Таким образом, при очередном рекурсивном вызове функции PIMP она сначала печатает свой аргумент. В качестве условия останова применяется равенство аргумента единице.

Проверка четности осуществляется с помощью вызова функции REM, подсчитывающей остаток от деления на 2.

Следующий пример программы - функция объединения множеств на Лиспе. Множества представляются списками, подаваемыми на вход в качестве аргументов. Особенностью множеств по отношению к спискам является то, что они, во-первых, в отличие от списков, не упорядочены, а во-вторых, элементы в списке могут повторяться, а в множество в общем случае элемент может входить лишь один раз. Поэтому программа должна выявлять элементы, входящие и в первый, и во второй список. В связи с этим реализуем сначала вспомогательную функцию APP, проверяющую вхождение элемента в список. Затем она используется в функции UNI, объединяющей аргументы E и F, с помещением результата в F:

```
 ; Объединение множеств на Лиспе
; вспомогательная функция — поиск атома A в списке X
(DEFUN APP (A X)
(COND
      ((NULL X) NIL)
      ((EQUAL A (CAR X)) T)
      (T (APP A (CDR X)))
 )
 )
 ; собственно объединение множеств
(DEFUN UNI (E F)
(COND
    ((NULL E) F)
    ((APP (CAR E) F) (UNI (CDR E) F))
    (T (UNI (CDR E) (CONS (CAR E) F)))
```

```
)
 )
```

В следующем примере разберем уже приводившуюся программу подсчета суммы нечетных элементов списка:

```
; подсчет суммы нечетных элементов списка
(DEFUN SUMNECH (X)
(COND
     ((NULL X) 0)
    ((ZEROP (REM (CAR X) 2)) (SUMNECH (CDR X)))
    (T (+ (CAR X) (SUMNECH (CDR X))))
)
 )
```

Поскольку необходимо перебирать несколько элементов списка, программа является рекурсивной. Условие останова — пустой список на входе: ясно, что сумма нечетных в нем равна нулю. Для проверки четности используются встроенные функции Лиспа ZEROP и REM. Если первый элемент списка четный, он просто отбрасывается и продолжается выполнение для хвоста списка, в противном случае он добавляется к результату обработки хвоста списка. Впрочем, программа подсчета суммы нечетных на Лиспе может быть написана и по-другому.

В Лиспе присутствует возможность использования так называемых лямбда-функций, или безымянных функций. Лямбда-функция определяется следующим образом:

((LAMBDA (<пер>) <выр1>) <выр2>)

и возвращает значение выражения *<выр1>*, в котором все вхождения формального параметра *<пер>* замещены значениями выражения *<выр2>*. Первое выражение — тело лямбда-функции, второе выражение — фактический аргумент. Примеры:

((LAMBDA (X) (ATOM X)) 123) → T

(MAPCAR (LAMBDA (I) (+ I 10)) '(1 2 3 4 5)) → (11 12 13 14 15)

Для примера — совсем короткая программа подсчета суммы нечетных элементов числовой последовательности на Лиспе с телом функции, состоящим из одной строки. Программа использует трюк со свойствами умножения и сложения, лямбда-функцию и функции высших порядков:

```
 (DEFUN SUMN (S)
(APPLY '+ (MAPCAR (LAMBDA (X) (* X (REM X 2))) S))
 )
```

В данной программе в явном виде рекурсия не используется. Следующий пример программы относится к классу программ сортировки, а именно сортировке перестановкой. Идея, воплощенная в этой программе, заключается в том, что вначале на первую позицию в списке перемещается минимальный элемент, а затем процедура повторяется для оставшейся части списка:

```
;Сортировка перестановкой на Лиспе
;вспомогательная функция возвращает минимальное значение в списке (DEFUN MINS(X)
(COND
((NULL (CDR X)) (CAR X)) ;В списке один элемент — он же min
```

```
((< (CAR X) (MINS (CDR X))) (CAR X)); рекурсия
(T (MINS (CDR X)))
 )
 )
;вспомогательная функция удаляет первое вхождение элемента в список
(DEFUN REMV (EL S)
(COND
    ((NULL S) NIL)
    ((EQ EL (CAR S)) (CDR S)); если первый-возвращается хвост
    (T (CONS (CAR S) (REMV EL (CDR S))));рекурсия
)
 )
 ;собственно функция сортировки списка по возрастанию перестановкой
(DEFUN SORTS(X)
(COND
    ((NULL (CDR X)) X)
    (T (CONS (MINS X) (SORTS (REMV (MINS X) X))))
 )
 )
```

Еще один вариант программы сортировки основан на идее вставки, начиная с первого, элемента в список на подобающее ему место в зависимости от величины, своего рода выполнение команды:

```
;Сортировка вставкой на Лиспе
;вспомогательная функция вставки элемента
;в список на подобающее место
(DEFUN INS(X S)
(COND
   ((NULL S) (LIST X)); если был пустой список
   ((< X (CAR S)) (CONS X S) );если меньше первого, вставляется перед ним
    (T (CONS (CAR S) (INS X (CDR S))))   ) )
 ;функция сортирует список по возрастанию
 (DEFUN SORTS(X)
(COND
    ((NULL X) X)
   (T (INS (CAR X) (SORTS (CDR X))))
)
 )
```

Следует упомянуть также о том, что в функциональном программировании используется концепция «ленивых вычислений», когда значение функции вычисляется лишь в момент, когда (и если) оно потребуется. Порядок вычисления сложных выражений в Лиспе следующий:

1. Аргументы функции вычисляются в порядке перечисления. Пример: (FUN A B C).

2. Композиции функций вычисляются от внутренней к внешней. Пример: (FUN1 (FUN2 A) B).

3. Представление функции анализируется до того, как начинают вычисляться аргументы, так как в некоторых случаях аргумент можно и не вычислять.

4. При вычислении лямбда-выражений связи между именами переменных, а также между именами функций и их определениями

накапливаются в так называемом ассоциативном списке, пополняемом при вызове функции и освобождаемом при выходе из функции.

2.4.7. Современное состояние и перспективы функционального программирования

Подчеркнем, что в настоящее время функциональное направление в языках программирования активно развивается, и, несмотря на наибольшую известность Лиспа, он далеко не единственный — и не самый концептуально чистый — представитель семейства аппликативных языков, то есть языков, в которых главным действием остается применение функции к аргументу. Среди наиболее известных — Hope, Miranda, целое семейство языков, порожденных Haskell (назван в честь математика и логика Хаскелла Карри), — Curry, Clean, Gofel. Еще одно семейство функциональных языков, ведущих начало от языка ML, включает Standard ML, Objective CAML, Harlequin's MLWorks, F# и др. Языки Nemerle и F# построены на платформе .Net. Язык Joy базируется на композиции функций, а не на лямбда-исчислении. В настоящее время Joy считается каноническим примером языка конкатенативного программирования. Все в Joy является функциями, принимающими стек как аргумент и возвращающими стек в качестве результата. Бывает, что создатели изначально функционального языка программирования стремятся позволить использовать те или иные возможности других подходов. Так, языки Scala и Oz относятся к так называемым мультипарадигмальным, или языкам, в которых поддерживается более одной систем взглядов на программирование, но в них обоих функциональная парадигма занимает достойное место.

Среди отечественных разработок функциональных языков упомянем Пифагор (параллельный информационно-функциональный алгоритмический (Красноярск, 1995)) и Фактор. Все же наиболее многочисленным остается Лисп-семейство, включающее сегодня многочисленные ответвления (MuLisp, Common Lisp, Scheme, Closure, Sisal, FP, FL и др.). В качестве примеров применения Лиспа в индустрии помимо систем искусственного интеллекта назовем AutoLisp — диалект, используемый как встроенный базовый язык программирования в системе автоматизации проектирования AutoCAD. На Лиспе написан популярный в UNIX-сообществе текстовый редактор Emacs. Встроен язык и в свободно распространяемый аудиоредактор Audacity, и в не менее свободный графический редактор Gimp.

2.5. Логическое программирование

2.5.1. Введение в логическое программирование

Логическое программирование — наряду с функциональным — еще одна из наиболее известных альтернатив традиционному императивному программированию. Можно сказать, что парадигма логического программирования еще более повышает уровень абстракции. Достаточно сказать, что языки логического программирования, в частности Пролог, заслуженно могут быть отнесены к декларативным. Иными словами, Пролог-система может находить решение задачи самостоятельно, без четкого предписания со стороны программиста, как это сделать. Программисту необходимо лишь указать, что мы хотим получить, и четко описать предметную область на некотором формальном языке. Например, мы знаем, что x · 15 – 1 = 224. Как найти x? На языке Пролог-Д для этого достаточно написать:

?УМНОЖЕНИЕ(x,15,-1,224).

Система дает немедленный ответ: x=15. Но самое замечательное здесь то, что мы можем использовать одну и ту же программу для решения другой задачи: сколько нужно отнять от произведения x · 15, чтобы получить 224? Для этого следует лишь переставить переменную:

?УМНОЖЕНИЕ(15,15,x,224).

и в ответ получим: –1. Не правда ли необычно? В традиционных языках программирования мы привыкли к тому, что программа имеет четко определенные входные и выходные данные. Более того, мы можем использовать Пролог, например, для нахождения всех пар аргументов, дающих заданный ответ. Например, мы можем спросить у системы GNU Prolog, как сформировать список [a,b,d] слиянием двух подсписков, следующим образом (на особенностях синтаксиса читателю предлагается пока не останавливаться, хотя все довольно прозрачно):

?- append(X,Y,[a,b,d]).

и получить набор ответов:

X = [] Y = [a,b,d] ? ;
X = [a] Y = [b, d] ? ;
X = [a,b] Y = [d] ? ;
X = [a,b,d] Y = [] ? ;

Нажатие на клавишу {;} используется для получения очередного варианта ответа.

В основу языков логического программирования, как следует из самого названия, положена логика. А логика, как известно, это «наука о правильном мышлении», «правила рассуждения». С точки зрения приверженцев этого стиля, язык программирования должен быть инструментом мышления. Л. Стерлинг и Э. Шапиро [14] пишут: «Мы склонны думать, что программирование может и должно быть частью процесса собственно решения задачи; что рассуждения можно организовать в виде программ, так что следствия сложной системы предположений можно исследовать, “запустив” предположения; что умозрительное решение задачи должно… сопровождаться работающей программой, демонстрирующей правильность решения и выявляющей различные аспекты проблемы».

Сходство и различие языков программирования Лисп и Пролог приведены в табл. 2.

Таблица 4

Сходство и различие языков программирования Лисп и Пролог

Сходство	**Различие**
Оба ориентированы на символьную обработку	В Прологе теоретической основой является логика предикатов, а в Лиспе — лямбда-исчисление
Широко применяется рекурсия	Лисп активнее используется в США, а Пролог — в Европе и Японии
Активно используются при разработке систем искусственного интеллекта	В Лиспе вычисляется единственное значение функции, а в Прологе система может самостоятельно искать множество значений, удовлетворяющих наложенным ограничениям, причем как от аргументов к значению функции, так и наоборот, от значения к приводящим к его получению аргументам

Непредубежденному человеку, не знакомому с кухней создания ЭВМ, машина фон Неймана покажется причудливым существом. Рассуждения в терминах машины требуют значительных мыслительных усилий. Эта особенность программирования на компьютерах фон Неймана приводит к разделению создателей программ на изобретающих методы решения задач *алгоритмистов* и реализующих алгоритм на ЭВМ знатоков особенностей машинного языка - *кодировщиков*. При этом в программировании, как и в логике, требуется явное выражение знаний с помощью некоторого формализма.

Подобная формализация утомительна, однако в логике полезна, поскольку обеспечивает понимание задачи. Для машины фон Неймана это вряд ли приводит к подобному эффекту. Компьютерам все еще далеко до того, чтобы стать равными партнерами человека в интеллектуальной деятельности. Однако использование логики при программировании представляется естественным и плодотворным, поскольку она сопровождает процесс мышления.

Логическое программирование сильно отклоняется от основного пути развития языков программирования. Здесь используется не некоторая последовательность преобразований, отталкивающихся от архитектуры фон Неймана и присущего ей набора операций, а теоретическая модель, никак не связанная с каким либо типом машины.

Логическое программирование базируется на следующем убеждении: не человека надо учить мышлению в терминах компьютера, а напротив, компьютер должен уметь выполнять действия, свойственные человеку.

Логическое программирование подразумевает, что конкретные инструкции не задаются, вместо этого используются логические аксиомы и правила вывода, с помощью которых формулируются сведения о задаче и

предположения, достаточные для ее решения. Постановка задачи формализуется в виде логического утверждения, подлежащего доказательству. Такое утверждение называется целевым утверждением или вопросом.

Программа - это множество аксиом и правил вывода, а исполнение - попытка логического вывода целевого утверждения из программы. При этом используется конструктивный метод доказательства целевого утверждения, то есть в процессе доказательства находятся аргументы, которые делают истинным данное целевое утверждение. Покажем, как можно в логическом языке задать некоторое правило рассуждений, например: «любит — значит, дарит цветы». На Прологе-Д можно записать:

дарит_цветы(x,y):-любит(x,y). %логическое правило
любит(Вася, Маша). %факт
любит(Коля, Даша). %факт
?дарит_цветы(x,y). %вопрос Пролог-системе.
Будет сгенерирован набор ответов:
x=Вася y=Маша

x=Коля y=Даша

Пролог можно «научить» и другим правилам, например: «любит - значит, дарит брильянты» или, согласно поговорке, «бьет - значит, любит».

Здесь мы подходим к тому, что логика бывает разной. Это действительно так. Кроме классической, существуют различные виды логик. Наиболее бурное развитие логика получила с начала XX века, когда математикам удалось логику формализовать. Примерами неклассических логик являются (приводятся характерные для них формулировки):

- темпоральные («когда-нибудь пойдет снег», «снег будет идти всегда»);
- многозначные (пример трехзначной: истина/ложь/неопределенность, пятизначная — система школьных оценок и т. п.);
- нечеткая (размытая);
- вероятностная («с вероятностью 0,7 Вася любит Машу»);
- логика возможных миров («X истинно в возможном мире К» (варианты развития событий)).

Изначально логика считалась частью философии. Среди ученых, фиксировавших законы правильного мышления, можно назвать Аристотеля, Авиценну, П. Абеляра, И. Канта, Г. В. Лейбница. Со временем логика становилась все более точной. Значительный вклад в этот процесс внесли такие ученые, как Дж. Буль и де О. Морган. Можно сказать, что, начиная с трудов Дж. Пеано, а также классической книги Б. Рассела и А. Уайтхеда «Основания математики», формальная логика выделилась в самостоятельную дисциплину.

Здесь, однако, возникают некоторые вопросы, например:

- В какой степени уместно распространение математических методов, принятых в теории чисел и алгебре, на логику?

- Каким образом переходить от логической символики к конкретной интерпретации, связанной с той или иной предметной областью? Иначе говоря, истинна ли некоторая формальная логика применительно к данной предметной области?

Математика всегда считалась областью знаний с точной системой рассуждений. В основаниях самой математики лежит логика. Леммы, теоремы, доказательства — неотъемлемые составляющие математического знания. Математика, в свою очередь, лежит в основе множества наук, более того, по степени значимости применения математического аппарата мы выделяем точные науки. Происхождение информатики довольно сильно связано с математической логикой. Неудивительно, что с появлением ЭВМ и созданием для них первых программ возникало желание «скрестить» логику и программирование. К 1969 году относятся первые получившие довольно широкую известность попытки создания логических языков программирования — это Absys, созданный в университете Абердина в Шотландии, и Planner — фунционально-логический язык, разработка К. Хьюита из лаборатории искусственного интеллекта Массачусетского технологического института (США). Planner впоследствии породил целый ряд потомков, среди которых QA4, Popler, Conniver, QLisp и Ether.

Одним из ранних языков логического программирования стал эквациональный язык Golux в котором программа представляет собой совокупность равенств (1973). Его автор П. Хайес сформулировал идею «вычисления = контролируемая дедукция». Однако наибольшую популярность завоевал Пролог и его диалекты [14], разработанный в 1972 году в университете Марсель-Экс. Авторами Пролога являются А. Кольмрауэр и Р. Ковальский.

Среди наиболее известных диалектов Пролога, поддерживаемых соответствующими трансляторами и средами разработки, отметим GNU Prolog, Turbo Prolog, Prolog-10, SW-Prolog, IC Prolog, Edinburgh Prolog, Visual Prolog, MUProlog, P# для платформы .Net. Были созданы версии для параллельного программирования, например Parlog и Parallel Prolog.

Отметим реализации Пролога на базе русского синтаксиса встроенных предикатов, например уже упоминавшийся Пролог-Д. Впоследствии были разработаны и другие языки логического программирования, среди которых в первую очередь можно назвать Mercury, Strand, ALF, Fril, Gödel, XSB, KLO, ShapeUp, Hayes. Весьма популярными стали и так называемые гибридные языки — языки, поддерживающие сразу несколько парадигм программирования. Функционально-логическим можно считать широко используемый для создания многопоточных приложений в телекоммуникационной отрасли Erlang, разработанный фирмой Ericsson.

Поскольку Пролог распространен наиболее широко, именно он рассматривается здесь как пример языка логического программирования.

2.5.2. Язык программирования Пролог

Как уже отмечалось, язык Пролог был создан во французском городе Марселе. Целью работы было создание языка, который мог бы делать логические заключения на основе заданного текста. Название Prolog является сокращением от Programming in logic. Главными разработчиками являются Р. Ковальский и А. Кольмрауэр. Перед этим Ковальский работал над программой на Фортране, предназначенной для доказательства теорем. Эта программа должна была обрабатывать фразы на естественном языке. Первая реализация языка Пролога с использованием компилятора Вирта ALGOL-W была закончена в 1972 году, а основы современного языка были заложены в 1973 году. Пролог постепенно распространялся среди ученых, занимавшихся логическим программированием, однако недостаток эффективных реализаций сдерживал его распространение.

Следует отметить, что, как и в случае с Лиспом, прогресс в области производительности средней ЭВМ позволил в значительной степени снять проблемы Пролога, связанные с не столь высокой скоростью обработки, как, например, в Си.

Программа на языке Пролог состоит из предложений (выражений). Предложения строятся из атомарных элементов — переменных, констант, а также термов. Для обозначения переменных в GNU Prolog используются латинские буквы и цифры, начиная с прописной латинской буквы. В некоторых реализациях допускается использование особой безымянной переменной, обозначаемой символом подчеркивания _. Константы обозначаются с помощью латинских букв и цифр.

Пролог-Д для этих же целей помимо латинских позволяет использовать русские буквы. В качестве констант большинство реализаций Пролога допускает также использование чисел.

Терм определяется рекурсивно:

- константы и переменные — термы;
- термами являются также составные конструкции, содержащие имя функтора и последовательность из одного или более заключенных в скобки аргументов, также являющихся термами. Функтор задается своим именем и арностью — количеством аргументов.

Примеры термов:

- *петя — простой терм, состоящий из одной константы петя;*
- *J(0) — терм с арностью 1 с функтором J и аргументом константой 0;*
- *Горячий (молоко) — терм с функтором Горячий арности 1 и аргументом молоко;*
- *Имя (Ваня,Коля) — терм с функтором Имя арности 2 и аргументами Ваня и Коля;*

- *List(a,li(b,n)) — терм с функтором List арности 2 с первым аргументом константой a и вторым аргументом составным термом li арности 2 и аргументами b и n.*

Термы, в которые не входят переменные, называются основными. Подразумевается, что функтор позволяет получить некоторое значение. Функторы могут быть разными. Фундаментальную роль в языке Пролог играют предикаты. Напомним математическое определение: предикат — это функция, принимающая значения из двухэлементного множества, например: *ИСТИНА и ЛОЖЬ, 0 и 1, ДА и НЕТ.*

Существует тесная связь между предикатами и отношениями. Если некоторые предметы A_1, A_2, …, A_n вступают между собой в отношение P, можно сказать, что, будучи использованными в качестве аргументов соответствующего предиката арности n, они дают в качестве результата значение ИСТИНА и ЛОЖЬ — в противном случае.

Программа на языке Пролог может включать несколько видов выражений:

- факты;
- правила вывода;
- вопросы.

Также допустимы комментарии. В качестве комментария воспринимается любая последовательность символов, начинающаяся с символа % и заканчивающаяся символом конца строки. Факты в Прологе — это предикаты, заканчивающиеся точкой. Примеры фактов в программе на Прологе:

```
Отец (Алексей,Иван).     %Алексей — отец Ивана
 Плюс(2,3,5).   %2+3=5
любит(Вася,Маша).
любит(Коля,Маша).
любит(Вася,яблоки).
```

Как правило, тот или иной факт из жизни (предметной области) можно сформулировать разными способами. Искусство программирования на Прологе заключается, в частности, в их правильном выборе. Примеры записи одного и того же факта из жизни в программе на Прологе:

```
дарит_цветы(Вася,Маша).
дарит(Вася,Маша,цветы).
дарит(цветы,Вася).
получатель(цветы,Маша).
```

Вопрос записывается так же, как и факт, то есть обычно представляет собой предикат. Как правило, вопрос, в отличие от факта, предваряется символом ?. Допускаются так называемые конъюнктивные вопросы, содержащие несколько имен предикатов, разделенных запятой. В конце вопроса ставится точка, как и при оформлении факта. Вопрос является целью логического вывода для Пролог-системы. Система должна попытаться найти ответ на вопрос в виде ДА или НЕТ.

В программе может формулироваться несколько вопросов-целей. Примеры вопросов в программе на Прологе:

?любит (Вася,Маша). % а любит ли Вася Машу? Да или нет?
?любит (Вася,X). % перечислить все X, которых любит Вася
?любит (Вася,_). % любит ли Вася хоть кого-нибудь? Да или нет?
?любит (X,яблоки) % выдать всех, кто любит яблоки
?любит (X,Y) % вывести всех на чистую воду
?любит (X,Маша),любит (X,яблоки). %найти любящего Машу яблокоеда

Правило - это утверждение, включающее две части, разделенные символами **:-** (в некоторых реализациях Пролога может использоваться <- или иная комбинация символов):

A**:-**$B_1,B_2,..,B_n$**.**

В конце правила вывода также присутствует точка. Выражение A называется заголовком, или заключением правила, в правой части идет его тело, состоящее из B_n — посылок, или гипотез. Должно соблюдаться условие $n \geq 0$, и заголовок, и посылки должны представлять собой предикаты. Примеры правил вывода языка Пролог:

сын (X,Y):-отец (Y,X),мужчина(X).
дочь (X,Y):-отец (Y,X),женщина(X).

Допускается несколько правил вывода с одним и тем же заголовком. В простейшем случае программа на Прологе может состоять из одного лишь вопроса (цели). Можно поинтересоваться: а о чем спрашивать систему, если мы не задали ей ни одного факта? Дело в том, что большинство реализаций Пролога поставляются с заранее определенными встроенными предикатами, иногда простейшими арифметическими и логическими, а иногда формирующими довольно богатую библиотеку (например, в GNU Prologe).

2.5.3. Написание баз данных и знаний на Прологе

Весьма хорошо Пролог подходит для написания баз данных и знаний. Попробуем проделать это на примере базы знаний о родственных связях. Побудительным мотивом для написания этой программы может служить то, что родственная связь - тоже отношение, соответственно, легко переводится в базовые для Пролога термины предикатов. Для начала необходимо определиться, от каких базовых родственных связей (отношений) мы будем отталкиваться, задавая более сложные отношения. Логично выделить самую близкую кровную связь, а именно «быть родителем» («быть ребенком»). Сразу договоримся для определенности, что первым аргументом будет идти родитель, а вторым - его ребенок, и станем придерживаться этого же принципа для всех последующих предикатов. Весьма важной в человеческой цивилизации является половая принадлежность. Поэтому, можно выделить также предикаты для «быть мужчиной» и «быть женщиной». Мы здесь не имеем в виду переносный смысл этих понятий, а подходим к делу чисто формально. Немаловажен для нас и институт брака. Введем базовый предикат для обозначения отношения «быть супругами».

Предположим, у нас имеются факты, описывающие некоторую семью:

родитель(Коля,Петя).
родитель(Коля,Маша).
родитель(Коля,Володя).
мужчина(Коля).
мужчина(Володя).
женщина(Маша).
супруги(Коля,Нина).

Программа на Прологе, содержащая одни лишь факты, является прямым аналогом базы данных - информационной системы, в которой можно запоминать некоторую информацию и из которой по запросу пользователя извлекать ее. Например, в данном случае мы можем задать системе вопрос: кто является детьми Коли?

? родитель(Коля,X).

Иными словами, формулируем к нашей базе следующий запрос: «выдай все X такие, для кого Коля является родителем». В настоящее время наиболее распространены так называемые реляционные базы данных, опирающиеся на понятие «отношения». Пролог построен на базе предикатов, а предикаты неразрывно связаны с отношениями. Поэтому неудивительны явные аналогии между реляционной базой данных и программой на Прологе. Имя предиката является аналогом имени отношения (таблицы). Каждый факт соответствует строке отношения (таблицы) в реляционной базе. Однако Пролог обладает более богатыми возможностями. На нем мы легко можем создавать не просто базы данных, а базы знаний.

Определение базы знаний подразумевает активный характер ее содержимого, возможность на основании одной имеющейся в базе информации получать (выводить) другую. Поддерживающий этот процесс механизм в Прологе основан на использовании правил вывода. Попробуем научить программу определять, кто кому приходится матерью, отцом, дочерью, сыном, братом и сестрой. Можно сделать безусловно верный вывод о том, что если некто является родителем и при этом мужчиной, он будет отцом.

Правило вывода на Прологе может быть сформулировано следующим образом (в примере, поскольку для установления факта отцовства несущественно, кто именно является ребенком, используется безымянная переменная _):

отец(A,B):-родитель(A,_),мужчина(A).

Аналогичным образом формулируется правило вывода для матери. Для получения на основании фактов о том, что B является женщиной и некоторый A является родителем B, заключения о том, что B — дочь, можно использовать правило вывода

дочь(B,A):-родитель(A,B),женщина(B).

Подобным же образом можно построить правила вывода для получения информации о сыне, братьях и сестрах. Получится следующий набор правил вывода:

отец(A,B):-родитель(A,_),мужчина(A).

мать(A,B):-родитель(A,_),женщина(A).
сын(A,B):-родитель(B,A),мужчина(A).
дочь(B,A):-родитель(A,B),женщина(B).
брат(A,B):-родитель(Y,A),родитель(Y,B),мужчина(A).
сестра(A,B):-родитель(Y,A),родитель(Y,B),женщина(A).

На следующем этапе перейдем к другому поколению — опишем родственные связи вида «бабушка», «дедушка», «внук» и «внучка»:

внук(A,B):-родитель(Z,Y), родитель(Y,A),мужчина(A).
внучка(A,B):-родитель(Z,Y),родитель(Y,A),женщина(A).
бабушка(A,B):-родитель(A,Y),родитель(Y,B),женщина(A).
дедушка(A,B):-родитель(A,Y), родитель(Y,B),мужчина(A).

Перейдем к тетям и дядям, племянникам и племянницам:

племянник(A,B):-родитель(Y,A),брат(Y,B),мужчина(A).
племянник(A,B):-родитель(Y,A),сестра(Y,B),мужчина(A).
племянница(A,B):-родитель(Y,A),брат(Y,B),женщина(A).
племянница(A,B):-родитель(Y,A),сестра(Y,B),женщина(A).
тетя(A,B):-племянник(B,A),женщина(A).
тетя(A,B):-племянница(B,A),женщина(A).
дядя(A,B):-племянник(B,A),мужчина(A).
дядя(A,B):-племянница(B,A),мужчина (A).

В этих примерах продемонстрированы такие особенности Пролога, как существование нескольких правил вывода для одного и того же заключения (Пролог-система перебирает их все в процессе попытки вывода цели и, если какое-то правило подходит по посылкам, использует его) и получение симметричного отношения из ранее определенного, например, «тетя» на базе «племянник» или «племянница». Следующие правила позволяют описать некоторые нетривиальные родственные связи:

стрый(x,y):-папа(z,y),брат(x,z). % дядя по отцу (брат отца)
уй(x,y):-мама(z,y),брат(x,z). % дядя по матери (брат матери)
свекор(x,y):-муж(z,y),папа(x,z). % отец мужа
свекоровь(x,y):- муж(z,y), мама(x,z). % мать мужа
тесть(x,y):-жена(z,y),папа(x,z). % отец жены
теща(x,y):-жена(z,y),мама(x,z). % мать жены

зять(x,y):- муж(x,z),дочь(z,y). % муж дочери
невестка(x,y):-жена(x,z),сын(z,y). % жена сына
сноха(x,y):- жена(x,z),папа(y,z). % жена сына для его отца

деверь(x,y):-муж(z,y),брат(x,z).% брат мужа
золовка(x,y):-муж(z,y),сестра(x,z). % сестра мужа
шурин(x,y):-жена(z,y),брат(x,z). % брат жены
свояченица(x,y):-жена(z,y),сестра(x,z). % сестра жены

Наконец, сформулируем более глубокое понятие, а именно предок. Правда, в русской разговорной речи не принято относить к предкам собственно родителей, а также бабушек и дедушек. Однако в молодежном жаргоне это принято: кто не слышал выражения «предки на даче»? Мы последуем этому примеру:

предок(A,B):-родитель(A,B). %граничное условие
предок(A,B):-предок(A,C),родитель(C,B).

2.5.4. Введение арифметики через логику в Прологе

Весьма интересным является использование Пролога для демонстрации мощи логики. Например, мало кто задумывается о том, что обычная школьная арифметика может быть определена логическим путем. Мы будем рассматривать это в несколько упрощенном виде. Для начала необходимо логически определить понятие «натуральное число». Примем, что ноль является натуральным числом. На Прологе это может быть записано, например, как

nat(0).

Однако натуральный ряд с нуля всего лишь начинается, а уходит он в бесконечность! Что мы можем сделать для того, чтобы вместить понятие натурального числа в компьютер? Как уже отмечалось, не все понимают, что ЭВМ с конечной памятью просто не может работать с, если можно так выразиться, «настоящими» математическими числами. Имеют дело с конечными приближениями чисел — как действительных, так и дробных. Именно отводимое под стандартное представление числа в компьютере количество бит называется его разрядностью. Если при выполнении вычислений должно быть получено большее число, чем допускает разрядность ЭВМ, как правило, происходит так называемая ошибка переполнения. Но строгая, точная математическая логика должна предоставлять инструмент для описания бесконечного ряда чисел (оставляя пока за скобками то, что логика в Прологе реализуется на реально существующей ЭВМ с присущими ей ограничениями)! Действительно, логический подход позволяет использовать функтор *s(X)*, смысл которого заключается в том, что мы вводим для натурального числа операцию взятия следующего за ним по порядку числа (эквивалентную прибавлению единицы). Например, для нуля это будет единица, для единицы — двойка и т. д. Во введенных обозначениях единица — число, следующее за нулем, — может записываться как *s(0)*, следующее за единицей число два — как *s(s(0))* и т. д.

Фактически, мы воспользовались рекурсией. Натуральные числа — простейшая рекурсивная структура данных. Следующий важный шаг состоит в том, что мы вводим логическое правило вывода о том, что если некоторое число является натуральным, то следующее за ним по порядку (получаемое с помощью функтора *s(X)*) также будет являться натуральным числом. На Прологе (GNU Prolog) это может быть записано как

nat(s(X)): — nat(X).

Данный пример служит наглядной иллюстрацией ранее упоминавшегося факта о широком использовании рекурсии при программировании на Прологе. Остановом рекурсии служит предложение, в котором в качестве аргумента фигурирует 0.

Итак, программа на Прологе, определяющая натуральные числа (пока без операций над ними), выглядит следующим образом:

nat(0). nat(s(X)):- nat (X).

Кому-либо это может показаться удивительным, но может быть строго математически доказано, что порождаемые этой Пролог-программой сущности *0, s(0), s(s(0))* и т. д. эквивалентны натуральным числам, по крайней мере до момента, пока программа не прервется вследствие исчерпания наличествующей памяти компьютера.

Продолжим развитие нашей логической программы, описывающей натуральные числа. На множестве натуральных чисел существует естественный порядок. Программа может задать отношение ≤ (меньше или равно) следующим образом:

```
mir(0,X):–nat(X).
mir(s(X),s(Y)):–mir(X,Y).
```

Это тоже рекурсивная программа, основывающаяся на том, что если X ≤ Y, то X + 1 ≤ Y + 1. Останов рекурсии, или граничное условие, здесь базируется на том, что ноль меньше любого натурального числа или равен ему.

Перейдем к арифметическим действиям. Фундаментальным является сложение. На Прологе логически оно может быть введено как

```
suma(0,X,X):–nat(X).
suma(s(X),Y,s(Z)):-suma(X,Y,Z).
```

Попробуем разобраться, на чем строится данный фрагмент программы. Поскольку операция сложения двухместная, ей может быть сопоставлен трехместный предикат *suma*. Будем подразумевать, что предикат истинен, если первый аргумент плюс второй аргумент дают третий аргумент. Первая строка означает, что добавление нуля не изменяет любое натуральное число. Вторая - что если у нас есть тройка чисел, таких, что *X + Y = Z, то (X + 1) + Y = (Z + 1)*. Возможно, кому-то покажется удивительным, но этого достаточно, чтобы логически определить сложение. Более того, эта же программа умеет выполнять вычитание. Убедиться в этом несложно — достаточно задать Пролог-системе несколько вопросов (здесь для краткости ответ системы записывается в той же строке после стрелки), например:

```
?suma(0,s(0),s(0)). → ДА (yes)
?suma(0,s(0),X). → s(0)
?suma(s(s(0)),s(s(0)),X). → s(s(s(s(0))))
?suma(s(0),X,s(s(s(0)))). → s(s(0))
?suma(X,Y,s(s(s(0)))). %ищет все пары X,Y такие что X+Y=3
```

Здесь сначала проверяется 0 + 1 = 1?, во второй строке вычисляется сумма нуля и единицы. Помимо прочего — на всякий случай! — мы проверяем, чему *у* нашей программы равно *2 + 2*. Две последующие строки иллюстрируют мощь Пролог-системы — мы используем определение сложения фактически для вычитания, или решаем уравнение *1 + X = 3*, а в следующей строке вообще автоматически получаем все пары натуральных чисел, дающих в сумме 3.

Естественно, в приводимых здесь программах для записи чисел - аргументов и интерпретации аргументов используется принятая нотация

0 - 0, 1 — s(0), 2 — s(s(0)) и т. д.

Получив такие обнадеживающие результаты для сложения, перейдем к умножению. Аналогично предикату suma определим трехместный предикат um:

um(0,X,0). % любое число при умножении на ноль дает ноль
*um(s(X),Y,Z):-um(X,Y,W),suma(W,Y,Z).%(X+1)*Y=Z,если X*Y=W, Z=X+W*

Предикат определяется рекурсивно, как и suma, но при определении умножения мы дополнительно опираемся на ранее определенное логически сложение. Проверим, чему равно 2 · 2 у нашей программы:

?um(s(s(0)),s(s(0)),X). → *s(s(s(s(0))))*

На волне успеха перейдем к определению операции возведения в степень:

```
step(step(X),0,0).
step(0,step(X),s(0)).
step(step(N),X,Y):-step(N,X,Z),um(Z,X,Y).
```

Далее приводятся чуть более сложные программы (тем не менее опирающиеся на логическое введение натуральных чисел), сопровождаемые примерами вопросов для их проверки:

```
%факториал
fakt(0,s(0)).
fakt(s(X),Z):-fakt(X,Q),um(s(X),Q,Z).

?fakt(s(s(s(0))),X).

%минимум
mini(X,Y,X):-mir(X,Y).
mini(X,Y,Y):-mir(Y,X).

?mini(s(s(0)),s(0),X).

%строго меньше
men(0,s(X)):-nat(X).
men(s(X),s(Y)):-men(X,Y).

? men(X,s(s(0))).

%остаток от деления
ost(X,Y,X):-men(X,Y).
ost(X,Y,Z):-suma(X1,Y,X),ost(X1,Y,Z).

?ost(s(s(s(0))),s(s(0)),X).


%наибольший общий делитель числа
nod(X,0,X). nod(X,Y,G):-men(0,X),men(0,Y),ost(X,Y,Z),nod(Y,Z,G).

?nod(s(s(s(s(0)))),s(s(0)),X).
```

2.5.5. Обработка списков на языке Пролог

В языке программирования Пролог, как и в Лиспе, активно используется рекурсия. Есть в нем и удобные средства для обработки такой базовой рекурсивной структуры данных, как список. Для задания списка в Прологе достаточно заключить в квадратные скобки некоторую последовательность термов, перечисленных через запятую. В предельном случае список может быть пустым и обозначается просто [] (или nil).

Список может включать в свой состав вложенные списки, как и в языке программирования Лисп. Примеры списков на Прологе:

[a,b,c]
[1,2]
[]
[[a,b],2,[[a]]]

Весьма удобным средством Пролога, превосходящим по краткости средства Лиспа CAR и CDR, является способ взятия начального элемента и хвоста списка. Предположим, что в списке надо выделить голову и обозначить ее X и хвост, обозначив его Y. В Прологе достаточно написать:

[X|Y]

Просто поставив вертикальную черту, можно сразу выделить и первый элемент, и все остальные. Более того, Пролог допускает записи вида

[X,Y|Z]

В приведенном примере выделяются первый и второй элементы списка X и Y и хвост Z. Заметим, что нам для этого не потребовались ни цикл, ни переменная-итератор, которые, скорее всего, нужны были бы в аналогичной программе на императивном языке.

Логическими средствами Пролога список может быть определен как

```
список([]).
список([X|S]):-список(S).
```

Разберем приведенную программу. Итак, сначала утверждается, что пустой список есть список. И во второй строке добавление в начало списка S произвольного элемента X оставляет структуру списком. Приведенный пример позволяет в первом приближении почувствовать стиль написания программ обработки списков на языке Пролог.

Пойдем дальше. Вхождение элемента в список проверяет следующая программа:

```
членсп(X,[X|_]).
членсп(X,[Y|S]):-членсп(X,S).
```

Перейдем от проверки вхождения одного элемента в список (хотя этим элементом может быть и подсписок) к проверке того, является ли некоторая последовательность элементов подсписком некоторого списка. Для упрощения решения задачи построения предиката подсписок удобно решать ее поэтапно и сначала реализовать два других предиката: префикс и суффикс. Под префиксом здесь понимается подсписок в начале другого списка, под суффиксом — подсписок, завершающий другой список.

В программировании на Прологе использование самостоятельных содержательных отношений в качестве вспомогательных предикатов — типовая техника. Произвольный подсписок может быть потом определен как суффикс некого префикса или префикс некого суффикса:

- [А,Б,В,Г,Д,Е] — исходный список;
- [А,Б,В] — префикс;
- [Б,В,Г,Д,Е] — суффикс;
- [Б,В,Г] — подсписок.

Приведем определение подсписка через суффикс и префикс на Прологе:

подсписок(X,Y):-префикс(P,Y),суффикс (X,P).%префикс суффикса

подсписок (X,Y):-префикс(X,S),суффикс(S,Y). %суффикс префикса Сами предикаты префикс и суффикс могут быть определены следующим образом:

префикс([],_).

префикс([X|S],[X|Y]):-префикс(S,Y).

В приведенной программе утверждается, что, во-первых, пустой список может считаться префиксом любого списка, а во-вторых, если некоторый подсписок S является префиксом списка Y, добавление одинаковых элементов в начало S и Y не изменит ситуации:

суффикс(X,X).

суффикс(X,[_|S]):-суффикс(X,S).

Здесь сначала определяется, что список может считаться собственным суффиксом, а затем рекурсивно определяется суффикс при добавлении в начало рассматриваемого списка произвольного элемента. Однако определение подсписка через префикс суффикса или, наоборот, суффикс префикса не является единственно возможным. Вообще при программировании на Прологе часто встречается ситуация, когда одно и то же понятие можно определить разными способами. Как и при программировании на других языках, мы имеем дело с ситуацией семантически одинаковых, но синтаксически различающихся программ. Но разница обычно не только в записи — разные реализации могут иметь существенно различающиеся показатели эффективности, например количество операций, необходимых для достижения поставленной цели.

В следующей программе предикат подсписок определяется через префикс и рекурсию:

подсписок(X,Y):-префикс(X,Y).

подсписок(X,[Y|S]):-подсписок(X,S).

Сначала говорится, что префикс - частный случай подсписка, а затем - что подсписок некоторого списка остается им при добавлении в начало этого списка элемента X. Следующей интересной операцией над списками является слияние списков (приписывание, или конкатенация). Определим предикат слияние, имеющий три аргумента и становящийся истинным, если при приписывании второго аргумента-списка к первому получится третий список:

слияние([],Y,Y).

слияние([D|X],Y,[D|Z]):-слияние(X,Y,Z).

Данная программа рекурсивная, в ней говорится сначала о том, что приписывание пустого списка не меняет любой список (граничное условие рекурсии), а затем — что если два списка давали при слиянии некоторый список, добавление в начало результата и одного из аргументов одинаковых элементов не изменит ситуации. Любопытно, что, имея предикат слияние, можно еще одним способом определить предикат подсписок, весьма кратко:

подсписок(X,S):-слияние(A,W,S),слияние(X,B,W).

Сравним эту строку с воображаемой программой на Паскале или Си, делающей то же самое.

Следующая программа выделяет в списке концевой элемент:

последний(X,S):-слияние(_,[X],S).

Интересным является написание — по аналогии с ранее созданной программой на Лиспе — и сравнение с ней программы обращения списков на Прологе:

обрат([],[]). обрат([X|S],Z):-обрат(S,Y),слияние(Y,[X],Z).

Рассмотрение программ обработки списков было бы неполным, если бы мы не рассмотрели программу подсчета суммы нечетных членов числовой последовательности на языке Пролог:

```
СУМНЕЧ([],0). %останов рекурсии и сумма для пустого списка =0
СУМНЕЧ([X|Y],S):-ЧЕТ(X), СУМНЕЧ (Y,S). %четные — пропускаем
СУМНЕЧ ([X|Y],S):-НЕЧЕТ(X),СУМНЕЧ(Y,L),СЛОЖЕНИЕ(L,X,S).
 % нечетные складываем
```

Правда, в случае отсутствия в стандартной библиотеке используемой версии Пролога предикатов ЧЕТ и НЕЧЕТ их придется определить отдельно.

Рассмотрим программы сортировки списков. В программировании на языке Пролог плодотворно применяется метод «образовать и проверить». Дело в том, что, как мы видели ранее, Пролог-система в процессе вычислений способна перебирать возможные значения аргумента. В методе «образовать и проверить» один из предикатов служит для генерации возможных вариантов ответа, а другой — для проверки того, является ли полученное решение искомым — удовлетворяющим заданным требованиям. Данный подход применим для решения переборных задач, встречающихся в задачах создания интеллектуальных систем, например логических игр и пр.

Если подходить к задаче сортировки с позиций метода «образовать и проверить», можно создать предикат *упорядочен*, который будет проверять список, полученный в качестве аргумента, на отсортированность. Другой же предикат может предназначаться для генерации всех возможных перестановок списка. Несомненно, среди них будет и искомая — когда элементы выстроены по порядку. Начнем с предиката *упорядочен*:

```
упорядочен([X]).
упорядочен([X,Y|S]):-МЕНЬШЕ(X,Y),упорядочен([Y|S]).
```

Первая строка программы говорит о том, что мы можем считать список, состоящий из одного элемента, упорядоченным. Вторая строка показывает, что если мы имеем некоторый упорядоченный список, начинающийся с элемента Y и имеющий хвост S, и добавляем в его начало меньший Y элемент X, список останется упорядоченным. Для предиката

перестановка нам потребуется вспомогательный предикат *вырез*, который будет извлекать из списка некоторый элемент, точнее, его первое вхождение (операция вырезания). Поскольку Пролог не реализует непосредственно функций, предикат будет иметь следующие аргументы: первый — элемент, подлежащий извлечению из списка, второй — список, подлежащий обработке, и наконец — результирующий список:

```
вырез(X,[X|S],S).
вырез(X,[Y|S],[Y|Z]):-вырез(X,S,Z).
```

В первой строке говорится, что если извлечь первый элемент из списка с головой X и хвостом S, получается список S. Вторая строка рекурсивная и показывает, что если в результате вырезания элемента X из списка S получается список Z, то в случае рассмотрения списков, полученных путем добавления головы Y к S и Z, ситуация не изменится.

Напишем теперь предикат *перестановка*, основанный на операции *вырез*:

```
перестановка([],[]).
перестановка(X,[Z|S]):-вырез(Z,X,Y),перестановка(Y,S).
```

Первая строка показывает, что как ни переставляй содержимое пустого списка, получишь лишь пустой список. Вторая строка говорит, что, например, получить из списка X его перестановку можно, взяв некоторый элемент Z из списка и переставив его на первое место. Наконец, сам предикат сортировка, который и будет собственно процедурой сортировки, выглядит весьма просто:

```
сортировка(X,Y):-перестановка(X,Y),упорядочен(Y).
```

Можно сказать, что здесь написание программы напоминает формальную постановку задачи. При этом программист избавлен от мелких деталей, связанных с реализацией решения на ЭВМ, таких как размер массива ячеек памяти, отведенных под данные, или индекс в этом массиве. Иными словами, Пролог поднимает программирование на более высокий уровень абстракции.

2.5.6. Программы обработки информации, записанной символами

Как говорилось ранее, Пролог, как и Лисп, изначально ориентирован на обработку символьной информации, а не на вычисления, как, например, Фортран. В языке программирования Пролог переменная может принимать как числовое, так и символьное значение. Мощь обработки информации, записанной символами, может быть проиллюстрирована приведенными далее примерами, в первом из которых Пролог «обучается» правильно распознавать в некой последовательности символов корректно записанный в соответствии с определенными правилами многочлен, а затем выполнять дифференцирование функций (имеется в виду получение по записи исходной функции записи формулы вычисления производной). Чем не искусственный интеллект?!

В силу некоторых ограничений версии Пролог-Д примеры в данном подразделе приводятся на GNU Prolog.

Программа, распознающая многочлены:

```
%Программа распознавания многочленов
%Вспомогательные предикаты
constant(X):-atom(X).
constant(X):-integer(X).
natural_number(X):-integer(X),X>0.
%Символьное распознавание полиномов
poly(X,X).
poly(Term,X):-constant(Term).
 poly(Term1+Term2,X):-poly(Term1,X),poly(Term2,X).
poly(Term1-Term2,X):-poly(Term1,X),poly(Term2,X).
 poly(Term1*Term2,X):-poly(Term1,X),poly(Term2,X).
 poly(Term1/Term2,X):-poly(Term1,X),constant(Term2).
poly(Term^N,X):-natural_number(N),poly(Term,X).
```

Приведенной программе можно задать вопрос

*?poly(x^2-3*x,x)*

и получить ответ «да» на английском языке.

Следующей программой будет программа символьного дифференцирования (подобные преобразования иногда называют символьными вычислениями, на них специализируются системы компьютерной алгебры, например Maxima).

Предикат *derivative* истинен, если последний аргумент - запись производной по второму аргументу функции, запись которой подается в качестве первого аргумента. Программа символьного дифференцирования на языке GNU Prolog:

```
%Программа символьного дифференцирования выражений
 %Справочник базовых производных для популярных функций
 diff(X,X,1).
diff(X^N,X,N*X^(N-1)).
diff(sin(X),X,cox(X)).
diff(cos(X),X,-sin(X)).
diff(e^X,X,e^X).
diff(log(X),X,1/X).

%Правила дифференцирования сумм, разностей, частных
 diff(F+G,X,DF+DG):-diff(F,X,DF),diff(G,X,DG).
diff(F-G,X,DF-DG):-diff(F,X,DF),diff(G,X,DG).
 diff(F*G,X,F*DG+DF*G):-diff(F,X,DF),diff(G,X,DG).
diff(1/F,X,-DF/(F*F)):-diff(F,X,DF).
 diff(F/G,X,(G*DF-F*DG)/(G*G)):-diff(F,X,DF),diff(G,X,DG).
```

Нетрудно убедиться в способностях программы, задавая ей вопросы вида, например,

?derivative(e^x,x,e^X) или diff(e^x+sin(x),x,X).

2.5.7. Отрицание и отсечения в Прологе

Некоторую сложность в логическом программировании с помощью Пролога имеет рассмотрение логического отрицания. Дело в том, что используемый в Прологе механизм логического вывода на основе метода резолюции не позволяет точно устанавливать ложность некоторого предложения. В силу этого «настоящего» строгого логического отрицания большинство версий Пролога не включают. Частичной, но не строгой заменой логического понятия ЛОЖЬ может служить расценивание ложности как *невыводимости*. Иначе говоря, если, исходя из заданных программой фактов и правил вывода, заданную цель вывести не удается, она считается ложной. Однако при реализации описанного подхода нас тоже подстерегают подводные камни: в связи с ограниченностью ресурсов ЭВМ логический вывод может прерваться, не дойдя до доказательства нужного предложения.

Отрицание в смысле невыводимости может быть реализовано в Прологе с помощью так называемого отсечения. Под отсечением понимается заведомое отбрасывание некоторых ветвей в дереве перебора вариантов ответа. Во многих программах это имеет смысл, например, из эвристических соображений. Кроме того, у Пролога имеется проблема получения нескольких одинаковых ответов, если в программе можно логически прийти к ним разными путями. Отсечение позволяет прервать процесс логического вывода, если уже получен ответ. Записывается отсечение с помощью восклицательного знака *!*, который рассматривается как специальный предикат без аргументов, всегда возвращающий ИСТИНУ и не дающий откатиться назад, чтобы выбрать альтернативное решение для уже установленных подцелей, то есть тех, которые в строке программы расположены левее отсечения. На подцели, расположенные правее, отсечение не влияет. Кроме этого, отсечение отбрасывает все предложения программы, находящиеся ниже места, где сработало отсечение.

Пример программы поиска максимума с использованием отсечения:

```
maxi(X,Y,X):-X>Y,!.
maxi(_,Y,Y).
```

Нахождение максимума из трех чисел с использованием отсечения:

```
max3(X,Y,Z,X):-X>Y,X>Z,!.
max3(_,Y,Z,Y):-Y>=Z,!.
max3(_,_,Z,Z).
```

В программах на Прологе отсечение позволяет имитировать конструкцию *if…then…else…* императивных языков следующим образом:

```
%if <Y> then P
% else P2
S:-<Y>,!,P.
S:-P2.
```

После унификации порождающей цели с заголовком предложения, содержащего отсечение, цель выполняется и фиксируется для всех возможных выборов предложений. Конъюнкция целей до отсечения

приводит не более чем к одному решению, но не влияет на те, что выводятся после него. В случае возврата они могут порождать более одного решения.

Пример программы определения многочлена при использовании отсечения:

```
polinom (X,X):-!.
polinom (Term,X):-constant(Term),!.
polinom (Term1+Term2,X):-!,
polinom (Term1,X), polinom (Term2,X).
polinom (Term1-Term2,X):- !, polinom (Term1,X), polinom (Term2,X).
 polinom (Term1*Term2,X):-!, polinom (Term1,X), polinom (Term2,X).
 polinom (Term1/Term2,X):-!, polinom (Term1,X), polinom (Term2,X).
 polinom (Term1^N,X):-!, natural_number(N),polinom (Term1,X).
```

Как видите, отсечение позволяет достичь двух целей:

- получать более компактные программы;
- оптимизировать процесс вычислений за счет отбрасывания заведомо неперспективных ветвей дерева поиска.

Вернемся к определению отрицания в Прологе. Можно использовать следующую программу, использующую также встроенные системные предикаты fail и call (пример на GNU Prolog):

```
not(X):-call(X),!,fail.
not(X).
```

Выполняется эта программа следующим образом. Применяется первое правило. Если предложение X доказуемо, происходит отсечение, соответственно, цель не выполняется. Если X не выводится, то происходит переход ко второй строке.

2.6. Объектно-ориентированное программирование

2.6.1. Основные положения ООП

Концепция объектно-ориентированного программирования (ООП) появилась более сорока лет назад, как развитие идей процедурного программирования. Основная программа в процедурном программировании также является процедурой (функцией), в теле которой могут быть вызовы других процедур и функций – подпрограмм. Суть процедурного программирования проста: данные отдельно, поведение отдельно. Разделение кода на подпрограммы, во-первых, позволяет выделить повторно используемые фрагменты кода, а во-вторых, делает код программы структурированным.

Идеология объектно-ориентированного программирования, как следует из самого названия, строится вокруг понятия объект. Объект объединяет в себе и данные и поведение. Объект – это любая сущность, с которой имеет дело программа, а именно: объекты предметной области, моделируемые программой; ресурсы операционной системы; сетевые протоколы и многое другое. По сути, объект – это та же структура (составной тип), но

дополненная процедурами и функциями, управляющими элементами этой структуры. К примеру, для работы с файлом в процедурном языке программирования отдельно была бы создана переменная для хранения имени файла и отдельно – для хранения его дескриптора (уникальный идентификатор ресурса в операционной системе), а также ряд процедур работы с файлом: открыть файл, прочитать данные из файла и закрыть файл. Все бы эти процедуры, помимо обычных параметров и переменных для хранения результата, обязаны были бы принимать тот самый дескриптор, чтобы понять, о каком именно файле идет речь.

В объектно-ориентированном языке для этих же целей был бы описан объект-файл, который также бы хранил внутри себя имя и дескриптор и предоставлял бы пользователю процедуры для открытия, чтения и закрытия себя самого (файла, ассоциированного с конкретным объектом). Разница была бы в том, что дескриптор был бы скрыт от остальной части программы, создавался бы в коде процедуры открытия файла и использовался бы неявно только самим объектом. Таким образом, пользователю объекта (программному коду внешней по отношению к объекту программы) не нужно было бы передавать дескриптор каждый раз в параметрах процедур.

Объект – это комплект данных и методов работы с этими данными, часть из которых может быть скрыта от окружающего его мира, к которой и относятся детали реализации.

Объектом в объектно-ориентированном языке программирования является практически все, за исключением операторов: и элементарные типы являются объектами, и описание ошибки является объектом и, наконец, основная программа также является объектом.

Вторым основополагающим понятием ООП является класс. Класс – это тот самый новый в сравнении с процедурным программированием тип данных, экземпляры которого и называются объектами. Класс, как уже было сказано, похож на составной тип данных или структуру, но дополненный процедурами и функциями (методами) для работы со своими данными.

Перед тем, как перейти к описанию преимуществ, которые дает ООП разработчикам программного обеспечения в процессе проектирования, кодирования и тестирования программных продуктов необходимо познакомиться с наиболее часто встречающимися терминами в этом области.

Класс – тип данных, описывающий структуру и поведение объектов.

Объект – экземпляр класса.

Поле – элемент данных класса: переменная элементарного типа, структура или другой класс, являющийся частью класса.

Состояние объекта – набор текущих значений полей объекта.

Метод – процедура или функция, выполняющаяся в контексте объекта, для которого она вызывается. Методы могут изменять состояние текущего объекта или состояния объектов, передаваемых им в качества параметров.

Свойство – специальный вид методов, предназначенный для модификации отдельных полей объекта. Имена свойств обычно совпадают с именами соответствующих полей. Внешне работа со свойствами выглядит

точно так же, как работа с полями структуры или класса, но на самом деле перед тем, как вернуть или присвоить новое значение полю может быть выполнен программный код, осуществляющий разного рода проверки, к примеру, проверку на допустимость нового значения.

Член класса – поля, методы и свойства класса.

Модификатор доступа – дополнительная характеристика членов класса, определяющая, имеется ли к ним доступ из внешней программы, или же они используются исключительно в границах класса и скрыты от окружающего мира. Модификаторы доступа разделяют все элементы класса на детали реализации и открытый или частично открытый интерфейс.

Конструктор – специальный метод, выполняемый сразу же после создания экземпляра класса. Конструктор инициализирует поля объекта – приводит объект в начальное состояние. Конструкторы могут быть как с параметрами, так и без. Конструктор без параметров называют конструктором по умолчанию, который может быть только один. Имя метода конструктора, чаще всего, совпадает с именем самого класса.

Деструктор – специальный метод, вызываемый средой исполнения программы в момент, когда объект удаляется из оперативной памяти. Деструктор используется в тех случаях, когда в состав класса входят ресурсы, требующие явного освобождения (файлы, соединения с базами данных, сетевые соединения и т.п.)

Интерфейс – набор методов и свойств объекта, находящихся в открытом доступе и призванных решать определенный круг задач, к примеру, интерфейс формирования графического представления объекта на экране или интерфейс сохранения состояния объекта в файле или базе данных.

Статический член – любой элемент класса, который может быть использован без создания соответствующего объекта. К примеру, если метод класса не использует ни одного поля, а работает исключительно с переданными ему параметрами, то ничто не мешает его использовать в контексте всего класса, не создавая отдельных его экземпляров. Константы в контексте класса обычно всегда являются статическими его членами.

Рассмотрим свойства, которые приобретает программа при использовании объектно-ориентированного подхода к ее проектированию и кодированию.

Инкапсуляция обозначает сокрытие деталей реализации классов средствами награждения отдельных его членов соответствующими модификаторами доступа. Таким образом, вся функциональность объекта, нацеленная на взаимодействие с другими объектами программы группируется в открытый интерфейс, а детали тщательно скрываются внутри, что избавляет основной код бизнес-логики информационной системы от ненужных ему вещей. Инкапсуляция повышает надежность работы программного кода, поскольку гарантирует, что определенные данные не могут быть изменены за пределами содержащего их класса.

Наследование. Краеугольный камень ООП. В объектно-ориентированном программировании есть возможность наследовать структуру и поведение класса от другого класса. Класс, от которого наследуют, называется базовым или суперклассом, а класс, который получается вследствие наследования – производным или просто потомком. Любой класс может выступать как в роли суперкласса, так и в роли потомка. Связи наследования классов образуют иерархию классов. Множественным наследованием называют определение производного класса сразу от нескольких суперклассов. Не все объектно-ориентированные языки программирования поддерживают множественное наследование. Наследование – это эффективный способ выделения многократно используемых фрагментов кода, но у него есть и минусы, о которых будет рассказано далее.

Абстрагирование. Возможность объединять классы в отдельные группы, выделяя общие, значимые для них всех характеристики (общие поля и общее поведение). Собственно, абстрагирование и есть следствие наследования: базовые классы не всегда имеют свою проекцию на объекты реального мира, а создаются исключительно с целью выделить общие черты целой группы объектов. К примеру, объект мебель – это базовое понятие для стола, стула и дивана, всех их объединяет то, что это движимое имущество, часть интерьера помещений, и они могут быть выполнены для дома или офиса, а также относиться к «эконом» или «премиум» классу. В ООП есть для этого отдельное понятие абстрактный класс – класс, объекты которого создавать запрещено, но можно использовать в качестве базового класса. Наследование и абстрагирование позволяют описывать структуры данных программы и связи между ними точно так же, как выглядят соответствующие им объекты в рассматриваемой модели предметной области.

Полиморфизм. Еще одно свойство, которое является следствием наследования. Дело в том, что объектно-ориентированные языки программирования позволяют работать с набором объектов из одной иерархии точно так же, как если бы все они были объектами их базового класса. Если вернуться к примеру про мебель, то можно предположить, что в контексте создания информационной системы для мебельного магазина в базовый класс для всех видов мебели разумно добавить общий для всех метод «показать характеристики». При распечатке характеристик всех видов товара программа бы без разбору для всех объектов вызывала бы этот метод, а каждый конкретный объект уже сам бы решал, какую информацию ему предоставлять. Как это реализуется:

Во-первых, в базовом классе определяют общий для всех метод с общим для всех поведением. В случае с нашим примером это будет метод, печатающий общие для любых типов мебели параметры.

Во-вторых, в каждом производном классе, где это необходимо, переопределяют базовый метод (добавляют метод с тем же именем), где расширяют базовое поведение своим, например, выводят характеристики, свойственные только конкретному виду мебельной продукции. Метод в

базовом классе иногда вообще не обязан содержать какой-либо код, а необходим только для того, чтобы определить имя и набор параметров – сигнатуру метода. Такие методы называют абстрактными методами, а классы, их содержащие, автоматически становятся абстрактными классами.

Итак, полиморфизм – это возможность единообразного общения с объектами разных классов через определенный интерфейс. Идеология полиморфизма гласит, что для общения с объектом вам не нужно знать его тип, а нужно знать, какой интерфейс он поддерживает.

Интерфейс. В некоторых языках программирования (C#, Java) понятие интерфейса выделено явно - это не только открытые методы и свойства самого класса. Такие языки, как правило, не поддерживают множественного наследования и компенсируют это тем, что любой объект может иметь один базовый объект и реализовывать любое количество интерфейсов. Интерфейс в их интерпретации – это подобие абстрактного класса, содержащего только описание (сигнатуру) открытых методов и свойств.

Реализация интерфейса ложится на плечи каждого класса, который собирается его поддерживать. Один и тот же интерфейс могут реализовывать классы абсолютно разных иерархий, что расширяет возможности полиморфизма. К примеру, интерфейс «сохранение/восстановление информации в базе данных» могли бы реализовывать как классы иерархии «мебель», так и классы, связанные с оформлением заказов на изготовление мебели, а при нажатии на кнопку «сохранить» программа бы прошлась по всем объектами, запросила бы у них этот интерфейс и вызвала бы соответствующий метод.

Объектно-ориентированное программирование постоянно развивается, порождая новые парадигмы, такие как аспектно-ориентированное, субъектно-ориентированное и даже агентно-ориентированное программирование. Нужно отметит, что лавры ООП не дают покоя остальным теоретикам, и они спешат предложить свои варианты его совершенствования и расширения.

Некоторые элементы современного объектно-ориентированного программирования

Время не стоит на месте, да и времени с момента появления ООП уже прошло довольно много, поэтому не стоит удивляться, что сегодня словарь по объектно-ориентированному программированию серьезно разросся. Итак, вот некоторые новые термины и понятия, связанные с ООП.

События. Специальный вид объектов, создаваемый для оповещения одних объектов о событиях, происходящих с другими объектами. В разных языках программирования механизм событий реализуется по-разному: где-то с помощью специальных синтаксических конструкции, а где-то силами базовых средств ООП.

Универсальный тип. Концепция универсальных типов не связана непосредственно с концепцией ООП, но она является причиной появление таких элементов, как универсальный класс, универсальный метод, универсальное событие и т.д. Универсальный тип – это тип,

параметризованный другим типом (набором типов). Кем является этот тип-параметр в контексте проектирования универсального типа неизвестно, хотя есть возможность ограничить значения типов-параметров, заставив их быть производными от конкретного класса или реализовывать определенные интерфейсы. В качестве примера можно привести универсальный класс сортировки последовательности элементов, где тип элемента в последовательности заранее неизвестен. При проектировании такого класса важно указать, что тип-параметр должен поддерживать операцию сравнения.

Исключения. Еще один специальный вид объектов, поддерживаемый встроенным в конкретный язык программирования механизмом обработки ошибок и исключительных ситуаций. Исключения, помимо кода ошибки, содержат ее описание, возможные причины возникновения и стек вызовов методов, имевший место до момента возникновения исключения в программе.

2.6.2. Практика использования ООП

Рассмотрим примеры использования основных положений ООП - инкапсуляции, наследования и полиморфизма. Объектно-ориентированное программирование это программирование от объектов. Программа представляет собой набор связанных объектов. Каждый объект представляет собой набор каких-то данных и набор действий, которые он умеет делать. Естественно с объектом связывать именно те действия, которые необходимы при выполнении привязанных к нему действий. Эти действия называют **методами объекта**.

Типы объектов называются классами объектов. Объектно-ориентированный-программист описывает именно классы, а не объекты. В объектно-ориентированной программе также присутствует процедура, запускаемая при инициализации, которая создает объект базового класса, а затем этот объект уже сам всё что нужно делает - занимается порождением и уничтожением других объектов.

Инкапсуляция. Механизм работы инкапсуляции в ООП можно описать примерно так: при вызове того или иного метода класса сначала ищется метод в самом классе. Если метод найден, то он выполняется, и поиск этого метода завершается. Если же метод не найден, то происходит обращение к родительскому классу и ищется вызванный метод в нем. Если он найден, то происходит, как при нахождении метода в самом классе. А если нет, то продолжается дальнейший поиск вверх по иерархическому дереву, вплоть до корня (верхнего класса) иерархии. Этот пример отражает так называемый механизм раннего связывания.

Рассмотрим примет объекта в Турбо Паскале (Delphi). Для описания объектов в нем зарезервировано слово *object*. Тип *object* — это структура данных, которая содержит поля и методы. Описание объектного типа выглядит следующим образом:

```
<Идентификатор типа объекта>=Object
  <поле>;
    . . .
  <поле>;
  <метод>;
    . . .
  <метод>;
End;
```

Поле содержит имя и тип данных. Методы - это процедуры или функции, объявленные внутри декларации объектного типа, в том числе и особые процедуры, создающие и уничтожающие объекты (конструкторы и деструкторы). Объявление метода внутри описания объектного типа состоит только из заголовка (как в разделе *Interface* в модуле языка Pascal).

Для примера опишем объект «обыкновенная дробь» с методами «НОД числителя и знаменателя», «сокращение», «натуральная степень».

```
Type
  Natur=l..32767;
    Frac=Record
      P: Integer;
      Q: Natur
    End;
    Drob=Object A: Frac;
      Procedure NOD (Var C: Natur);
      Procedure Sokr;
      Procedure Stepen(N: Natur; Var C: Frac);
 End;
```

Описание объектного типа, собственно, и выражает такое свойство, как инкапсуляция.

Проиллюстрируем далее работу с описанным объектом, реализацию его методов и обращение к указанным методам. При этом понадобятся некоторые вспомогательные методы.

```
Type
  Natur=l..Maxint;
    Frac=Record
        P: Integer;
       Q: Natur
    End;
    {Описание объектного типа}
    Drob=Object A: Frac;
      Procedure Vvod; {ввод дроби}
      Procedure NOD(Var C: Natur); {НОД}
      Procedure Sokr;
      Procedure Stepen(N: Natur; Var C: Frac);
      Procedure Print; {вывод дроби}
    End;
    {Описания методов объекта}
    Procedure Drob.NOD;
      Var M,N: Natur;
    Begin
```

```
  M:=Abs(A.P); N:=A.Q;
  While M<>N Do
    If M>N
       Then If M Mod N<>0 Then M:=M Mod N Else M:»=N
        Else If N Mod M<>0 Then N:=N Mod M Else N:=M;
  C:=M
End;
Procedure Drob.Sokr;
   Var N: Natur;
Begin
   If A.P<>O
     Then Begin Drob.NOD(N); A.P:=A.P Div N; A.Q:=A.Q Div N End
     Else A.Q:=1
End;
Procedure Drob.Stepen;
   Var I: Natur;
Begin
   C.P:=1; C.Q:=1;
   For I:=1 To N Do   Begin C.P:=C.P*A.P; C.Q:=C.Q*A.Q End;
End;
Procedure Drob.Vvod;
Begin
   Write('Введите числитель дроби:'); ReadLn(A.P) ;
   Write('Введите знаменатель дроби:');ReadLn(A.Q) ;
End;
Procedure Drob.Print;
Begin
   WriteLn(A.P,'/',A.Q)
End;
{Основная программа}
Var Z: Drob; F: Frac;
Begin
Z.Vvod; {ввод дроби}
Z.Print; {печать введенной дроби}
Z.Sokr; {сокращение введенной дроби)
Z.Print; (печать дроби после сокращения}
Z.Stepen(4,F); (возведение введенной дроби в 4-ю степень}
WriteLn(F.P,'/'/F.Q)
End.
```

Прокомментируем отдельные моменты в рассмотренном примере.

Во-первых, реализация методов осуществляется в разделе описаний, после объявления объекта, причем при реализации метода достаточно указать его заголовок без списка параметров, но с указанием объектного типа, методом которого он является. Все это напоминает создание модуля в программе на Паскале, где те ресурсы, которые доступны при его подключении, прежде всего объявляются в разделе *Interface*, а затем реализуются в разделе *Implementation*. В действительности объекты и их методы реализуют чаще всего именно в виде модулей.

Во-вторых, все действия над объектом выполняются только с помощью его методов.

В-третьих, для работы с отдельным экземпляром объектного типа в разделе описания переменных должна быть объявлена переменная (или переменные) соответствующего типа. Легко видеть, что объявление статических объектов не отличается от объявления других переменных, а их использование в программе напоминает использование записей.

Наследование. Объектные типы можно выстроить в иерархию. Один объектный тип может наследовать компоненты из другого объектного типа. Наследующий объект называется потомком. Объект, которому наследуют, - предком. Если предок сам является чьим-либо наследником, то потомок наследует и эти поля и методы. Следует подчеркнуть, что наследование относится только *к типам, но не экземплярам объекта*.

Описание типа-потомка имеет отличительную особенность:

<имя типа-потомка>=Object(<имя типа-предка>),

дальнейшая запись описания обычная.

Следует помнить, что поля наследуются без какого-либо исключения. Поэтому, объявляя новые поля, необходимо следить за уникальностью их имен, иначе совпадение имени нового поля с именем наследуемого поля вызовет ошибку. На методы это правило не распространяется.

Опишем объектный тип «Вычислитель» с методами «сложение», «вычитание», «умножение», «деление» (некоторый исполнитель) и производный от него тип «Продвинутый вычислитель» с новыми методами «степень», «корень n-й степени».

```
Type
  BaseType=Double;
    Vichislitel=0bject A,B,C: BaseType;
    Procedure Init; {ввод или инициализация полей}
    Procedure Slozh;
    Procedure Vich;
    Procedure Umn;
    Procedure Delen
    End;
    NovijVichislitel=Object(Vichislitel)
    N: Integers;
    Procedure Stepen;
    Procedure Koren
End;
```

Обобщая вышесказанное, перечислим правила наследования:

- информационные поля и методы родительского типа наследуются всеми его типами-потомками независимо от числа промежуточных уровней иерархии;
- доступ к полям и методам родительских типов в рамках описания любых типов-потомков выполняется так, как будто бы они описаны в самом типе-потомке;
- ни в одном из типов-потомков не могут использоваться идентификаторы полей, совпадающие с идентификаторами полей

какого-либо из родительских типов. Это правило относится и к идентификаторам формальных параметров, указанных в заголовках методов;

- тип-потомок может доопределить произвольное число собственных методов и информационных полей;
- любое изменение текста в родительском методе автоматически оказывает влияние на все методы порожденных типов-потомков, которые его вызывают;
- в противоположность информационным полям идентификаторы методов в типах-потомках могут совпадать с именами методов в родительских типах. При этом одноименный метод в типе-потомке подавляет одноименный ему родительский, и в рамках типа-потомка при указании имени такого метода будет вызываться именно метод типа-потомка, а не родительский.

Вызов наследуемых методов осуществляется согласно следующим принципам:

- при вызове метода компилятор сначала ищет метод, имя которого определено внутри типа объекта;
- если в типе объекта не определен метод с указанным в операторе вызова именем, то компилятор в поисках метода с таким именем поднимается выше к непосредственному родительскому типу;
- если наследуемый метод найден и его адрес подставлен, то следует помнить, что вызываемый метод будет работать так, как он определен и компилирован для родительского типа, а не для типа-потомка. Если этот наследуемый родительский тип вызывает еще и другие методы, то вызываться будут только родительские или вышележащие методы, так как вызовы методов из нижележащих по иерархии типов не допускаются.

Полиморфизм. Как отмечалось выше, полиморфизм (многообразие) предполагает определение класса или нескольких классов методов для родственных объектных типов так, что каждому классу отводится своя функциональная роль. Методы одного класса обычно наделяются общим именем.

Пусть имеется родительский объектный тип «выпуклый четырехугольник» (поля типа «координаты вершин, заданные в порядке их обхода») и типы, им порожденные: параллелограмм, ромб, квадрат

Для указанных фигур необходимо создать методы «вычисление углов» (в градусах), «вычисление диагоналей», «вычисление длин сторон», «вычисление периметра», «вычисление площади».

```
Type
  BaseType=Double;
    FourAngle=Object x1,y1,x2,y2,x3,y3,x4,y4,A,B,C,D,D1,D2,Alpha,Beta,Gamma,Delta;
    P,S: BaseType;
    Procedure Init;
    Procedure Storony;
```

```
Procedure Diagonali;
Procedure Angles;
Procedure Perimetr;
Procedure Ploshad;
Procedure PrintElements;
End;
Parall=Object(FourAngie)
Procedure Storony;
Procedure Perimetr;
Procedure Ploshad;
End;
Romb=0bject(Parall)
Procedure Storony;
Procedure Perimetr;
End;
Kvadrat=0bject(Romb)
Procedure Angles;
Procedure Ploshad;
End;
Procedure FourAngie.Init;
  Begin
    Write ('Введите координаты вершин заданного четырехугольника:');
    ReadLn(x1, y1, x2, y2, x3, y3, x4, y4);
End;
Procedure FourAngie.Storony;
   Begin A:=Sqrt(Sqr(x2-xl)+Sqr(y2-yl));
      B:=Sqrt(Sqr(x3-x2)+Sqr(y3-y2));
      C:=Sqrt(Sqr(x4-x3)+Sqr(y4-y3));
      D:=Sqrt(Sqr(x4-xl)+Sqr(y4-yl));
   End;
Procedure FourAngle.Diagonali;
   Begin
      Dl:=Sqrt(Sqr(xl-x3)+Sqr(yl-y3));
      D2:=Sqrt(Sqr(x2-x4)+Sqr(y2-y4));
   End;
Procedure FourAngle.Angles;
Function Ugol(Aa,Bb,Cc: BaseType): BaseType;
   Var
      VspCos, VspSin: BaseType;
  Begin
     VspCos:=(Sqr(Aa)+Sqr(Bb)-Sqr(Cc))/(2*Aa*Bb);
     VspSin:=Sqrt(1-Sqr(VspCos));
     If Abs(VspCos)>le-7
       Then Ugol:=(ArcTan(VspSin/VspCos) +Pi*Ord(VspCos<0))/Pi*180
       Else Ugol:=90
    End;
Begin
   Alpha:=Ugol(D,A,D2);
   Beta:=Ugol(A,B,Dl);
   Gamma:=Ugol(B,C,D2);
   Delta: =Ugol (C,D, Dl);
End;
```

```
Procedure FourAngle.Perimetr;
  Begin P:=A+B+C+D End;
Procedure FourAngle.Ploshad;
  Var Peri, Per2: BaseType;
  Begin
    Perl:=(A+D+D2)/2;
    Per2:=(B+C+D1)/2;
    S:=Sqrt(Perl*(Perl-A)*(Perl-D)*(Perl-D2))+Sqrt(Per2*(Per2-B)*(Per2-C)*(Per2-
Dl))
 End;
Procedure FourAngle.PrintElements;
Begin
 WriteLn('Стороны:',A:10:6,B:10:6,C:10:6,D:10:6);
 WriteLn('Углы:',Alpha:10:4,Beta:10:4,Gamma:10:4,Delta:10:4);
 WriteLn('Периметр:',P:10:6,'Площадь:',S:10:6);
 WriteLn('Диагонали:', D1:10:6,D2:10:6)
End;
Procedure Parall.Storony;
  Begin
    A:=Sqrt(Sqr(x2-xl)+Sqr(y2-yl));
    B:=Sqrt(Sqr(x3-x2)+Sqr(y3-y2)) ;
    C:=A; D:=B
 End;
Procedure Parall.Perimetr;
  Begin
     P:=2*(A+B)
  End;
Procedure Parall.Ploshad;
  Var Per: BaseType;
  Begin
     Per:=(A+D+D2)/2;
     S:=2*Sqrt(Per*(Per-A)*(Per-D)*(Per-D2))
  End ;
Procedure Romb.Storony;
  Begin
     A:=Sqrt(Sqr(x2-xl)+Sqr(y2-yl));
     B:=A; C:=A; D:=A
  End;
Procedure Romb.Perimetr ;
  Begin
     P:=2*A
  End;
Procedure Kvadrat.Angles;
  Begin
    Alpha:=90; Beta:=90; Gamma:=90; Delta:=90;
  End;
Procedure Kvadrat.Ploshad;
  Begin
     S:=Sqr(A)
   End;
{Основная программа}
  Var obj: Kvadrat;
```

```
Begin
obj.Init;
obj.Storony;
obj.Diagonali;
obj.Angles;
obj.Perimetr;
obj.Ploshad;
obj.PrintElements
End.
```

Таким образом, вычисление соответствующего элемента в фигуре, если это действие по сравнению с другими является уникальным, производится обращением к своему методу.

Объектно-ориентированный подход активно применяется при разработке графических интерфейсов: окошки, кнопки, текстовые надписи, чекбоксы и другие графические элементы интерфейса (graphical user interface controls) - всё это объекты, вложенные друг в друга и посылающие друг другу сообщения. Например, когда главное окно получает сообщение закрыться, оно должно отправить аналогичные сообщения всем своим дочерним окнам (объектам).

Кроме того, объектно-ориентированный подход естественным образом используется при программировании сложных структур данных. Объект, соответствующий структуре данных, инкапсулирует в себе все данные этой структуры а также всю функциональность связанную с извлечением и модификацией данных.

2.6.3. Достоинства ООП

От любого метода программирования мы ждем, что он поможет нам в решении наших проблем. Но одной из самых значительных проблем в программировании является сложность. Чем больше и сложнее программа, тем важнее становится разбить ее на небольшие, четко очерченные части. Чтобы побороть сложность, мы должны абстрагироваться от мелких деталей. В этом смысле ООП представляют собой весьма удобный инструмент.

- Классы позволяют проводить конструирование из полезных компонент, обладающих простыми инструментами, что дает возможность абстрагироваться от деталей реализации.
- Данные и операции вместе образуют определенную сущность и они не «размазываются» по всей программе, как это нередко бывает в случае процедурного программирования.
- Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
- Инкапсуляция информации защищает наиболее критичные данные от несанкционированного доступа.

ООП дает возможность создавать расширяемые системы (extensible systems). Это одно из самых значительных достоинств ООП и именно оно

отличает данный подход от традиционных методов программирования. Расширяемость (extensibility) означает, что существующую систему можно заставить работать с новыми компонентами, причем без внесения в нее каких-либо изменений. Компоненты могут быть добавлены на этапе выполнения.

Расширение типа (type extension) и вытекающий из него полиморфизм переменных оказываются полезными преимущественно в следующих ситуациях.

- Обработка разнородных структур данных. Программы могут работать, не утруждая себя изучением вида объектов. Новые виды могут быть добавлены в любой момент.
- Изменение поведения во время выполнения. На этапе выполнения один объект может быть заменен другим. Это может привести к изменению алгоритма, в котором используется данный объект.
- Реализация родовых компонент. Алгоритмы можно обобщать до такой степени, что они уже смогут работать более, чем с одним видом объектов.
- Доведение полуфабрикатов. Компоненты нет надобности подстраивать под определенное приложение. Их можно сохранять в библиотеке в виде полуфабрикатов (semifinished products) и расширять по мере необходимости до различных законченных продуктов.
- Расширение каркаса. Независимые от приложения части предметной области могут быть реализованы в виде каркаса и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения.

Многоразового использования программного обеспечения на практике добиться не удается из-за того, что существующие компоненты уже не отвечают новым требованиям. ООП помогает этого достичь без нарушения работы уже имеющихся клиентов, что позволяет нам извлечь максимум из многоразового использования компонент.

- Мы сокращаем время на разработку, которое с выгодой может быть отдано другим проектам.
- Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.
- Когда некая компонента используется сразу несколькими клиентами, то улучшения, вносимые в ее код, одновременно оказывают свое положительное влияние и на множество работающих с ней программ.
- Если программа опирается на стандартные компоненты, то ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает ее использование.

2.6.4. Недостатки ООП

Объектно-ориентированное программирование требует знания четырех вещей.

1. Необходимо понимать базовые концепции, такие как классы, наследование и динамическое связывание. Для программистов, уже знакомых с понятием модуля и с абстрактными типами данных, это потребует минимальных усилий. Для тех же, кто никогда не использовал инкапсуляцию данных, это может означать изменения мировоззрения и может отнять на изучение значительное количество времени.
2. Многоразовое использование требует от программиста познакомиться с большими библиотеками классов. А это может оказаться сложнее, чем даже изучение нового языка программирования. Библиотека классов фактически представляет собой виртуальный язык, который может включать в себя сотни типов и тысячи операций. В языке Smalltalk, к примеру, до того, как перейти к практическому программированию, нужно изучить значительную часть его библиотеки классов. А это тоже требует времени.
3. Проектирование классов - задача куда более сложная, чем их использование. Проектирование класса, как и проектирование языка, требует большого опыта. Это итеративный процесс, где приходится учиться на своих же ошибках.
4. Очень трудно изучать классы, не имея возможности их «пощупать». Только с приобретением некоторого опыта можно уверенно себя почувствовать при работе с использованием ООП.

Как мы видели, усилия на освоение базовых концепций невелики, но вот в случае библиотек классов и их использования они могут быть очень существенными.

Поскольку детали реализации классов обычно неизвестны, то программисту, если он хочет разобраться в том или ином классе, нужно опираться на документацию и на используемые имена. И время, которое было сэкономлено на том, что удалось обойтись без написания собственного класса, должно быть отчасти потрачено (особенно вначале освоения) на то, чтобы разобраться в существующем классе.

В сложных иерархиях классов поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному классу. Для получения такой информации нужны специальные инструменты вроде навигаторов классов. Если конкретный класс расширяется, то каждый метод обычно сокращают перед передачей сообщения базовому классу. Реализация операции, таким образом, рассредоточивается по нескольким классам, и чтобы понять, как она работает, нам приходится внимательно просматривать весь код.

Методы, как правило, короче процедур, поскольку они осуществляют только одну операцию над данными. Зато количество методов намного выше. Короткие методы обладают тем преимуществом, что в них легче разбираться, неудобство же их связано с тем, что код для обработки сообщения иногда «размазан» по многим маленьким методам.

Абстракция данных ограничивает гибкость клиентов. Клиенты могут лишь выполнять те операции, которые предоставляет им тот или иной класс. Они уже лишены неограниченного доступа к данным. Причины здесь аналогичны тем, что вызвали к жизни использование высокоуровневых языков программирования, а именно, чтобы избежать непонятных программных структур.

Абстракцией данных не следует злоупотреблять. Чем больше данных скрыто в недрах класса, тем сложнее его расширять. Отправной точкой здесь должно быть не то, что клиентам не разрешается знать о тех или иных данных, а то, что клиентам для работы с классом этих данных знать не требуется.

Одно из возможных решений - строить базовый класс с минимальным числом методов, а затем уже реализовывать различные расширения этого класса, которые позволят нарастить функциональность.

Другой подход - дать возможность компоновщику удалять лишние методы. Такие интеллектуальные компоновщики уже доступны для различных языков и операционных систем. Таким образом, нельзя утверждать, что ООП вообще неэффективно.

Если классы используются лишь там, где это действительно необходимо, то потеря эффективности и на этапе выполнения и в смысле памяти сводится практически на нет.

2.6.5. Будущее ООП

Выживет ли объектно-ориентированное программирование, или оно лишь модное поветрие, которое скоро исчезнет?

Классы нашли свое место в большинстве современных языков программирования. Одно лишь это говорит о том, что им суждено остаться. Классы в самом ближайшем будущем войдут в стандартный набор концепций для каждого программиста, точно так же, как многие сегодня применяют динамические структуры данных и рекурсию, которые двадцать лет назад были также в диковинку. В то же время классы — это просто еще одна новая конструкция наряду с остальными. Нам нужно узнать, для каких ситуаций они подходят, и только здесь мы и будем их использовать. Правильно выбрать инструмент для конкретной задачи — обязательно для каждого мастерового и в еще большей степени для каждого инженера.

ООП ввергает многих в состояние эйфории. Пестрящая тут и там реклама сулит нам невероятные вещи, и даже некоторые исследователи, похоже, склонны рассматривать ООП как панацею, способную решить все

проблемы разработки программного обеспечения. Со временем эта эйфория постепенно уляжется. И после периода разочарования люди, быть может, перестанут уже говорить об ООП, точно также как сегодня вряд ли от кого можно услышать о структурном программировании. Но классы будут использовать как нечто само собой разумеющееся, и мы сможем, наконец, понять, что они собой представляют: просто компоненты, которые помогают строить модульное и расширяемое программное обеспечение.

Рекомендуемая литература

1. Голицина О.Л., Попов И.И. Программирование на языках высокого уровня: учебное пособие. – М.:ФОРУМ, 2010.
2. Клоксин У., Меллиш К. Программирование на языке Пролог. М., 1987.
3. Пратт Т., Зелковиц М. Языки программирования: разработка и реализация – СПб: Питер, 2002.
4. Себеста Р.У. Основные концепции языков программирования – М.: Изд. дом «Вильямс», 2001.
5. Филд А., Харрисон П. Функциональное программирование. Мир, 1993.
6. Флойд Р. Парадигмы программирования \\ Лекции лауреатов премии Тьюринга за первые двадцать лет (1966—1985) – М.: Мир, 1993.
7. Goldberg A., Robson D. Smalltalk-80 – The Language and It’s Implementation. Addison-Wesley, 1982.
8. Хьювенен Э., Сеппанен Й. Мир Лиспа., т.1,2, М.: Наука, 1994.
9. Пратт Т., Зелковиц М. Языки программирования. Разработка и реализация. – М. «Питер», 2002, 688 с.
10. Городняя Л.В. Основы функционального программирования. – М.: Интернет-Университет Информационных технологий. 2004. 272 с.
11. McCarthy J. LISP 1.5 Programming Mannual.- The MIT Press., Cambridge, 1963, 106p.
12. Дейкстра Э. Дисциплина программирования. М. «Мир», 1978, 275 с.
13. Фуксман А.П. Технические аспекты создания программных систем. - М.: Статистика, 1979, - 180 с.
14. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог. — М.: Мир, 1990.
15. Петров Ю.И. Программирование на языке высокого уровня Turbo Pascal. Часть 2 Программирование с использованием структурированных типов: Учебное пособие – Иркутск: Изд-во ИрГСХА,2014.-192 с.: ил.
16. Herman H. Goldstine. The Computer from Pascal to von Neumann. — Princeton University Press, 1980. — 365 p. — ISBN 9780691023670. (англ.)

17. Смирнов А. Д. Архитектура вычислительных систем : Учебное пособие для вузов. — М.: Наука, 1990. — С. 104. — 320 с. — ISBN 5-02-013997-1.
18. Лоханин, М. В. Архитектура современного компьютера: учебное пособие [Текст] : Учебное пособие / М. В. Лоханин, М. В. Лоханин, Яросл. гос. ун-т.им. П. Г. Демидова. - Электрон. текстовые дан. - [Б. м.] :ЯрГУ, 2011. - 96 с. - Режим до-ступа: http://rucont.ru/efd/237947
19. Фисун, Александр Павлович. Аппаратные средства вычислительной техники [Текст] : учебник для вузов. В 2-х книгах. Книга 1. / А. П. Фисун, В. А. Минаев [и др.]. - Электрон. текстовые дан. - Орел :ОрелГТУ, 2009. - 311 с. - Режим доступа: http://rucont.ru/efd/206349
20. Фисун, Александр Павлович. Аппаратные средства вычислительной техники [Текст] : учебник для вузов. В 2-х книгах. Книга 2 / А. П. Фисун, В. А. Минаев [и др.]. - Электрон. текстовые дан. - Орел :ОрелГТУ, 2009. - 151 с. - Режим доступа: http://rucont.ru/efd/206350
21. Баженова, Ирина Юрьевна. Языки программирования [Текст] : учеб. для вузов по направлениям "Фундаментальная информатика и информационные технологии" и "Информационная безопасность" / И. Ю. Баженова ; под ред. В. А. Сухомлина. - М. : Академия, 2012. - 357 с.
22. Владова, А. Ю. Разработка масштабируемых программ для многоядерных архитек-тур [Текст] : лаб. практикум / А. Ю. Владова. - Электрон. текстовые дан. - Оренбург : ГОУ ОГУ, 2006. - 53 с. - Режим доступа: http://rucont.ru/efd/190333