**Виртуальная машина**

*Об архитектуре машины фон Неймана и машинных языках*

Компьютер — это набор микросхем; если не оснастить его программами, он будет абсолютно бесполезен. ≪Сердцем≫ компьютера является центральный процессор, который умеет выполнять лишь элементарные операции, такие как арифметические

(например, сложение двух чисел), логические (например, исключающее ИЛИ), копирование содержимого из одной области памяти в другую и т. п. Последовательность этих элементарных действий и образует программу. Таким образом, решение любой задачи, например обработка изображения в PhotoShop или решение уравнений в MATLAB, сводится к выполнению длинной последовательности арифметических, логических и других элементарных операций. Эти элементарные операции мы будем в дальнейшем называть *машинными командами.* Множество команд, понимаемых данным процессором, называется *машинным языком.*

Процессор умеет работать только с данными, хранящимися в оперативной памяти (RAM — Random Access Memory) или на регистрах. Напомним, что процессор имеет небольшой набор ячеек собственной памяти, называемых регистрами; регистры нужны процессору для непосредственного выполнения арифметических и логических операций, адресации памяти и т. д.

Процессор выполняет программу (собственно, это он только и умеет делать и все время делает). И программы, и данные, с которыми эти программы работают, хранятся в оперативной памяти, ибо, как было отмечено выше, процессор умеет работать только с тем, что загружено в память. Хранятся они там в двоичном виде (в виде пресловутых нулей и единиц). Программа, загруженная в память, состоит из большого числа машинных

команд. Как же процессор понимает, какую команду ему сейчас выполнять? Особое место среди регистров процессора занимает регистр IP (Instruction Pointer). Именно в нем

хранится адрес команды, которую нужно выполнять в данный момент.

Примитивизм языка машинных команд приводит к тому, что решение даже самых, казалось бы, простых задач, например вывод содержимого файла на экран, становится очень трудоемким для программиста — требует многих десятков и сотен команд.

Язык — это средство и одновременно способ общения. Язык машинных команд — средство общения программиста с машиной.

В 1946 году Д. фон Нейман, Г. Голдстайн и А. Беркс в своей совместной статье изложили новые принципы построения и функционирования ЭВМ. В последствие на основе этих принципов производились первые два поколения компьютеров. В более поздних поколениях происходили некоторые изменения, хотя принципы Неймана актуальны и сегодня.

По сути, Нейману удалось обобщить научные разработки и открытия многих других ученых и сформулировать на их основе принципиально новое.

**Принципы фон Неймана**

1. **Использование двоичной системы счисления в вычислительных машинах**. Преимущество перед десятичной системой счисления заключается в том, что устройства можно делать достаточно простыми, арифметические и логические операции в двоичной системе счисления также выполняются достаточно просто.
2. **Программное управление ЭВМ**. Работа ЭВМ контролируется программой, состоящей из набора команд. Команды выполняются последовательно друг за другом. Созданием машины с хранимой в памяти программой было положено начало тому, что мы сегодня называем программированием.
3. **Память компьютера используется не только для хранения данных, но и программ**. При этом и команды программы и данные кодируются в двоичной системе счисления, т.е. их способ записи одинаков. Поэтому в определенных ситуациях над командами можно выполнять те же действия, что и над данными.
4. **Ячейки памяти ЭВМ имеют адреса, которые последовательно пронумерованы**. В любой момент можно обратиться к любой ячейке памяти по ее адресу. Этот принцип открыл возможность использовать переменные в программировании.
5. **Возможность условного перехода в процессе выполнения программы**. Не смотря на то, что команды выполняются последовательно, в программах можно реализовать возможность перехода к любому участку кода.

Самым главным следствием этих принципов можно назвать то, что теперь программа уже не была постоянной частью машины (как например, у калькулятора). Программу стало возможно легко изменить. А вот аппаратура, конечно же, остается неизменной, и очень простой.

*О трансляторах, компиляторах, интерпретаторах и виртуальных машинах*

Как бы хороша ни была программа на C++, процессору она не понятна, так как процессор понимает свой и только свой язык. Нужна программа, которая переводит текст, написанный на языке программирования высокого уровня, в язык машинных команд. Как же осуществляется этот перевод? Существует несколько подходов к этой весьма непростой проблеме.

Трансляция программы — преобразование программы, представленной на одном из языков программирования, в программу на другом языке, в определенном смысле равносильную первой.

Язык, на котором представлена входная программа, называется *исходным языком*, а сама программа — *исходным кодом*. Выходной язык называется *целевым языком.*

Транслятор — 1) программа или техническое средство, выполняющее трансляцию программы; 2) машинная программа, которая транслирует с одного языка на другой, в частности с одного языка программирования на другой.

Итак, цель трансляции — преобразовать текст с одного языка на другой, который понятен адресату. В случае программ-трансляторов адресатом является либо техническое устройство (процессор), либо программа-интерпретатор.

Транслятор, который преобразует программы в машинный язык, принимаемый и исполняемый непосредственно процессором, называется компилятором. Словарь по программированию и информатике А. Б. Борковского дает такое определение

компилятора: это программа, переводящая текст программы, написанной на языке высокого уровня, в эквивалентную программу на машинном языке.

Программа, подвергаемая компиляции, как правило, зависит от сервисов, предоставляемых операционной системой и сторонними библиотеками (например, файловый ввод-вывод или графический интерфейс). Машинный код, получаемый

при компиляции программы, также связан с этими сервисами, следовательно, ориентирован на исполнение в среде конкретной операционной системы. Архитектура (набор программно-аппаратных средств), для которой производится компиляция,

называется *целевой машиной.* Для каждой целевой машины (IBM, Apple и т. д.) и каждой операционной системы или семейства операционных систем, работающих на целевой матине, требуется свой компилятор. (Кроме того, компиляторы могут быть оптимизированы под разные типы процессоров из одного семейства путем использования специфичных для этих процессоров инструкций. Например, код, скомпилированный под процессоры семейства Pentium, может использовать специфичные для этих процессоров наборы инструкций — MMX, SSE, SSE2.) Платформозависимость получаемого кода является недостатком этого подхода.

Компилятор полностью (от начала и до конца) просматривает исходный файл и создает исполняемый файл. Компиляция и исполнение — два разных процесса, разделенных во времени. Процессор использует исполняемый файл, ни к исходному файлу, ни к компилятору не обращаясь. Отсюда одно из достоинств компилятора — не требуется наличие компилятора на целевой машине, для которой программа была скомпилирована.

Другой метод состоит в том, что программа транслируется и параллельно с этим исполняется с помощью интерпретатора.

Чистый интерпретатор — 1) вид транслятора, осуществляющего пооператорную (покомандную) обработку и выполнение исходной программы или запроса; 2) программа (иногда аппаратное средство), анализирующая команды или операторы программы и тут же выполняющая их; 3) языковый процессор, который построчно анализирует исходную программу и одновременно выполняет предписанные действия, а не формирует на машинном языке скомпилированную программу, которая выполняется впоследствии.

Чистая интерпретация применяется, как правило, для языков с простой структурой (например, АПЛ или Лисп). Интерпретаторы командной строки обрабатывают команды в скриптах в UNIX или в пакетных файлах (.bat) в MS-DOS также, как правило, в режиме чистой интерпретации.

Алгоритм работы интерпретатора следующий:

1) прочитать инструкцию;

2) проанализировать инструкцию и определить соответствующие действия;

3) выполнить соответствующие действия;

4) если не достигнуто условие завершения программы, прочитать

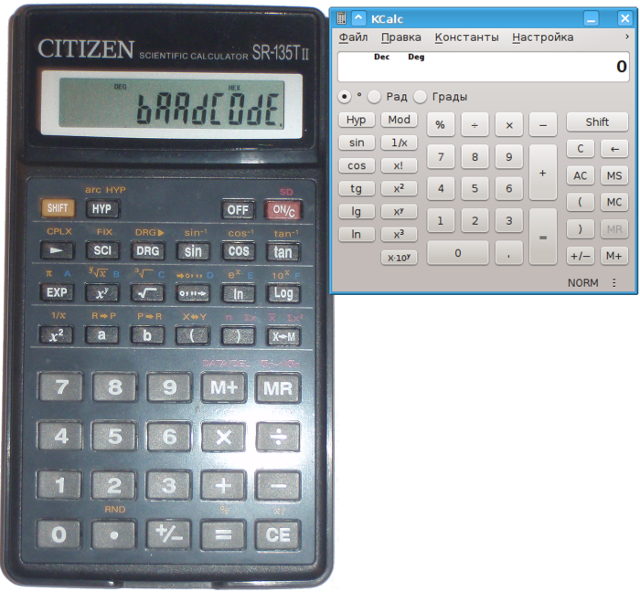
следующую инструкцию и перейти к пункту 2 .

Недостаток такого подхода состоит в том, что интерпретируемая программа не может выполняться отдельно без программы-интерпретатора — интерпретатор должен быть в

наличии на целевой машине, где должна исполняться программа (ср. с компиляторами). С другой стороны, достигается платформонезависимость на уровне исходных кодов: программа будет работать на любой платформе, где установлен соответствующий интерпретатор. Также к недостаткам интерпретатора относят и то, что интерпретируемая программа выполняется медленнее, поскольку промежуточный анализ исходного кода и планирование его выполнения требуют дополнительного времени в сравнении с непосредственным исполнением машинного кода, в который мог бы быть скомпилирован исходный код. Кроме того, практически отсутствует оптимизация кода, что приводит к дополнительным потерям в скорости работы интерпретируемых программ.

Существуют промежуточные между компиляцией и чистой интерпретацией решения — система из компилятора, переводящего исходный код программы в промежуточное представление, например в байт-код, и собственно интерпретатора, который выполняет полученный промежуточный код. Байт-код — машинно-независимый код низкого уровня, генерируемый транслятором и исполняемый интерпретатором. Программа на байт-коде обычно выполняется интерпретатором байткода и, как следствие, является сортируемой, т. е. может исполняться на разных платформах и архитектурах (это родовое преимущество интерпретируемых языков над компилируемыми). Преимущество таких систем над чистой интерпретацией состоит в том, что за счет выноса анализа исходного кода в отдельный, разовый проход и минимизации этого анализа в интерпретаторе достигается более высокая скорость выполнения программ. По этой причине многие современные интерпретируемые языки транслируют исходные тексты программ в байт-код и запускают интерпретатор байт-кода. К таким языкам относятся Perl, РНР и Python.

Словарь «Random House College Dictionary» определяет «virtual» как «проявляющий свойства и эффекты чего-либо, но не являющийся таковым на самом деле».

Если говорить широко, то виртуальная машина — это *программа*, задача которой состоит в реализации спецификаций определённого вычислительного устройства или класса устройств. В этом её главное отличие от «просто» физической машины, реализующей то же самое, но в аппаратуре. Всякая спецификация (архитектура компьютера) чаще всего включает в себя определение интерфейсов устройств и описание переходов между состояниями машины. Однако определение интерфейса, как известно, не должно налагать ограничений на способы его реализации.  
  
Многие из нас, наверное, не замечают, как часто ежедневно они сталкиваются с обеими типами машин — виртуальными и реальными. Например, самый простой калькулятор имеет две реализации — как специализированное устройство и как программа:  
  
  
  
И аппаратная машина слева, и виртуальная машина справа предоставляют один и тот же интерфейс — кнопки и экран, — и реализуют одни и те же функции — арифметические, логические, тригонометрические операции над числами, отображаемыми на экране.  
  
Виртуальный калькулятор — это пример виртуальной машины — программной копии того, что изначально существовало только в виде аппаратуры, физической машины, вполне конкретной и осязаемой *системы*.

## Языковые ВМ. Другой случай — это когда программа создаётся для чего-то «нереального» с самого начала, например, для языка программирования или среды исполнения (runtime environment). В этом случае такая *языковая* виртуальная машина будет способом реализации спецификации языка или среды.

## Языковые ВМ обычно проектируются для исполнения одного гостевого приложения (иногда многопоточного) в одной копии виртуальной среды. Другими словами, они не берут на себя типичные для многопользовательской/многозадачной операционной системы функции разграничения доступа к ресурсам. Задача языковой ВМ — предоставить программе окружение, напрямую не зависящее от деталей (и в какой-то мере ограничений) нижележащей физической системы, таких как используемые в последней процессоры, объёмы ОЗУ и дисков, наличие и особенности периферийных устройств и т.д.

## Языковая ВМ стоит на середине пути: от языков высокого уровня до машинных кодов того компьютера, на котором она исполняется. Поэтому при создании новой архитектуры языковой ВМ следует учитывать два фактора: удобство преобразования выбранных входных языков и скорость исполнения на конкретных аппаратных системах. От первого будет зависеть универсальность и расширяемость создаваемой среды, а от второго — верхняя граница скорости работы программ для этой ВМ.

## Базовой единицей исполнения для ВМ является *машинная инструкция*. Каждая такая инструкция должна определять операцию, выполняемую над данными, а также местоположение самих данных. Естественно, что набор операций сильно зависит от конкретной ВМ и может варьироваться в широких пределах. «Железные» наборы инструкций в этом куда более. С другой стороны, в подходах к организации обрабатываемых данных среди ВМ не так много разнообразия. Фактически есть две устоявшиеся концепции — хранить данные на стеке (стеках) и использовать выделенный набор регистров.

## Суммарно классификация ВМ выглядит следующим образом: https://habrastorage.org/files/b55/7a7/fed/b557a7fed2ef41a2a0dd878bb905fe6f.png

**Почему язык имеет значение**

В начале, при зарождении компьютерных дисциплин, не было языков

программирования. Программы выглядели так:

00110001 00000000 00000000

00110001 00000001 00000001

00110011 00000001 00000010

01010001 00001011 00000010

00100010 00000010 00001000

01000011 00000001 00000000

01000001 00000001 00000001

00010000 00000010 00000000

01100010 00000000 00000000

Это программа, складывающая числа от 1 до 10 и выводящая результат (1 +

2 +… + 10 = 55). Она может выполняться на очень простой гипотетической

машине. Для программирования первых компьютеров было необходимо

устанавливать большие массивы переключателей в нужные позиции, или

пробивать дырки в перфокартах и скармливать их компьютеру. Можете

представить, какая это была утомительная, подверженная ошибкам

процедура. Написание даже простых программ требовало большого ума и

дисциплины. Сложные программы были практически немыслимы.

Конечно, ручной ввод этих мистических диаграмм бит (нулей и единиц) давал

программисту возможность ощутить себя волшебником. И это чего-то стоило

в смысле удовлетворения работой.

Каждая строка указанной программы содержит одну инструкцию. На

обычном языке их можно описать так:

1 записать 0 в ячейку памяти 0

2 записать 1 в ячейку памяти 1

3 записать значение ячейки 1 в ячейку 2

4 вычесть 11 из значения ячейки 2

5 если у ячейки 2 значение 0, тогда продолжить с пункта 9

6 добавить значение ячейки 1 к ячейке 0

7 добавить 1 к ячейке 1

8 продолжить с пункта 3

9 вывести значение ячейки 0

Этот вариант легче прочесть, чем кучу бит, но он всё равно не очень удобен.

Использование имён вместо номеров инструкций и ячеек памяти может

улучшить понимание.

установить ‘total’ в 0

установить ‘count’ в 1

[loop]

установить ‘compare’ в ‘count’

вычесть 11 из ‘compare’

если ‘compare’ равно нулю, перейти на [end]

добавить ‘count’ к ‘total’

добавить 1 к ‘count’

перейти на [loop]

[end]

вывести ‘total’

Вот теперь уже не так сложно понять, как работает программа.

Первые две строки назначают двум областям памяти начальные значения.

total будет использоваться для подсчёта результата вычисления, а count

будет следить за числом, с которым мы работаем в данный момент. Строчки,

использующие ‘compare’, наверно, самые странные. Программе нужно

понять, не равно ли count 11, чтобы прекратить подсчёт. Так как наша

воображаемая машина довольно примитивна, она может только выполнить

проверку на равенство переменной нулю, и принять решение о том, надо ли

перепрыгнуть на другую строку. Поэтому она использует область памяти под

названием ‘compare’, чтобы подсчитать значение count – 11 и принять

решение на основании этого значения. Следующие две строки добавляют

значение count в счетчик результата и увеличивают count на 1 каждый раз,

когда программа решает, что ещё не достигла значения 11

Вот та же программа на JavaScript:

var total = 0, count = 1;

while (count <= 10) {

total += count;

count += 1;

}

console.log(total);

// → 55

Еще несколько улучшений. Главное – нет необходимости вручную

обозначать переходы между строками. Конструкция языка while делает это

сама. Она продолжает вычислять блок, заключённый в фигурные скобки,

пока условие выполняется (count <=10), то есть значение count меньше или

равно 10 Уже не нужно создавать временное значение и сравнивать его с

нулём. Это было скучно, и сила языков программирования в том, что они

помогают избавиться от скучных деталей.

В конце программы по завершению while к результату применяется операция

console.log с целью вывода.

И наконец, вот так могла бы выглядеть программа, если б у нас были

удобные операции range и sum, которые, соответственно, создавали бы

набор номеров в заданном промежутке и подсчитывали сумму набора:

console.log(sum(range(1, 10)));

// → 55

Мораль сей басни – одна и та же программа может быть написана и долго, и

коротко, читаемо и нечитаемо. Первая версия программы была совершенно

смутной, а последняя – почти настоящий язык – записать сумму диапазона

номеров от 1 до 10.

Хороший язык программирования помогает программисту сообщать

компьютеру о необходимых операциях на высоком уровне. Позволяет

опускать скучные детали, даёт удобные строительные блоки (while и

console.log), позволяет создавать свои собственные блоки (sum и range), и

делает простым комбинирование блоков.

**Постановка задачи**

Требуется:

1) придумать собственный низкоуровневый язык, выразительные способности которого будут достаточны для решения простых вычислительных задач;

2) написать на этом языке а) программу «определить количество ненулевых цифр в случайном целом числе (<2000000).» и б) программу нахождения наименьшего общего кратного двух натуральных чисел (числа определяются случайным образом);

3) написать на JScript виртуальную машину, исполняющую программы, написанные на разработанном языке; иными словами, написать на JScript интерпретатор, переводящий команды разработанного языка в команды JScript.

Указания к решению

Обсудим, что значит ≪низкоуровневый язык≫, о котором идет речь в постановке задачи. Низкоуровневый язык программирования (язык программирования низкого уровня) —язык программирования, близкий к языку машинных команд реального или виртуального процессора. Для обозначения команд в низкоуровневом языке программирования обычно применяется мнемоническое обозначение. Это позволяет запоминать команды не в виде последовательности двоичных нулей и единиц, а в виде осмысленных сокращений слов человеческого языка (обычно английских). Иногда одно мнемоническое обозначение соответствует целой группе машинных команд.

Среди синтаксических конструкций разрабатываемого низкоуровнего языка не будет, к примеру, ≪цикла for≫. Вместо этого будут команды, позволяющие непосредственно работать с памятью и регистрами (не настоящими, конечно, а эмулированными).

Примерами таких команд могут быть ≪разместить в памяти по адресу такому-то значение такое-то≫ или ≪присвоить регистру IP значение такое-то≫.

Язык программирования характеризуется прежде всего *синтаксисом* и *семантикой.* Синтаксис описывает структуру программы как набора символов (безотносительно к содержанию), семантика определяет смысловое значение предложений алгоритмического языка.

Какие же средства должен предоставлять наш будущий язык? Поскольку речь идет о вычислении факториала натурального числа, то в языке должна быть команда ввода этого самого числа с клавиатуры; иначе как объяснить компьютеру, факториал чего считать. В языке C++ для ввода используется команда сіn ≫ а , в Pascal — команда read (а). Как видно из этих примеров, синтаксис команды ввода разный, а смысл одинаковый. Договоримся, что наша команда ввода будет называться input. Обратим теперь внимание на то, что команды ввода данных с клавиатуры имели аргумент — куда считать, т. е. куда сохранить считанные данные. В высокоуровневых языках аргументом служит имя переменной, в низкоуровневом языке само понятие переменной может отсутствовать. Договоримся, что аргументом команды input будет адрес ячейки памяти, куда нужно сохранить введенное значение. Например, input 13 означает, что введенное с клавиатуры число нужно сохранить в 13-й ячейке эмулированной памяти.

Ясно, что после того, как факториал будет посчитан, значение (хранящееся в какой-то ячейке эмулированной памяти) надо вывести на экран. Язык должен иметь необходимые для этого средства. Пусть соответствующая команда называется output, единственным аргументом которой будет адрес ячейкипамяти, откуда надо вывести содержимое на экран. Например, output 7 - вывести на экран содержимое 7-й ячейки эмулированной памяти.

Потребуется также команда сложения двух чисел, определим ее. Команда add argl arg2 arg3 складывает числа, xpaнящиеся в памяти по адресам argl и arg2, и записывает сумму в память по адресу arg3. По аналогии могут быть введены команды ≪переход по условию≫, ≪присвоение значения≫ и т. д.

Приведем пример программы сложения двух чисел, написанной на разрабатываемом языке.

input 11

input 12

add 11 12 12

output 12

Сразу возникают вопросы: куда этот текст писать? Где и в каком формате его сохранять? Исходный текст можно сохранить в простом текстовом формате, например, в файле first\_prog.txt.

Необходимо отдавать себе отчет в том, что придумать язык, описать его синтаксис и семантику — это только половина дела. Программа first\_ prog.txt не может быть исполнена, так как еще не создан интерпретатор, понимающий команды языка, на котором написана эта программа. К его созданию переходим, при этом будем пользоваться термином ≪виртуальная машина≫.

Согласно приведенному выше определению, виртуальная машина — программная система, эмулирующая аппаратное обеспечение некоторой платформы. Как же виртуальная машина эмулирует оперативную память? Самый простой способ сымитировать память — объявить массив, куда можно сохранять и откуда можно считывать данные и команды. Чтобы эмулировать регистр IP, можно объявить переменную, в которой будет храниться адрес (номер ячейки массива) команды, исполняемой в данный момент. Известно, что прежде, чем исполнять программу, хранящуюся на жестком диске, ее необходимо загрузить в оперативную память и в регистре IP установить адрес первой команды этой программы. Для реализации нашей виртуальной машины считаем содержимое файла first\_ prog.txt и разместим команды и данные в ячейках массива, эмулирующего оперативную память. После этого переменной, эмулирующей регистр IP, присвоим соответствующее значение.

Программу, написанную на JScript и реализующую виртуальную машину, назовем, например, vm.js.

mem = new Array()

fso = new ActiveXObject(‘Scripting.FileSystemObject’)

text\_prog= fso.OpenTextFile*(‘*first\_prog.txt’)

var s*=’’*

while(!text\_prog.AtEndOfStream)

s+=text\_prog.ReadLine() *+’ ‘*

s+=’exit’

mem=s.split(‘ ‘*)*

ip=0

Теперь в отладочных целях выполним следующий фрагмент кода:

for(var count=0;count<mem.length;count++)

WScript.echo(’В ячейке ’,count,’ хранится ‘,mem[count])

На экране должен появиться так называемый дамп памяти:

В ячейке 0 хранится input

В ячейке 1 хранится 11

В ячейке 2 хранится input

В ячейке 3 хранится 12

В ячейке 9 хранится 12

В ячейке 10 хранится exit

Программа vm.js должна моделировать цикл выборки-исполнения. Реализуется это следующим образом:

while (mem [ip] ! *=’*exit’)

switch (mem [ip] ) {

case *‘* input*’:*

WScript.Echo*( ‘*Введи значение’)

mem[mem[ip+1]]=parseFloat(WScript.Stdln.ReadLine())

ip+=2

break

case ’output':

WScript. Echo (mem [mem [ip+1] ])

ip+=2

break

case ’add’:

mem [mem [ip+3] ] =mem [mem [ip+1] ] +mem [mem [ip+2] ]

ip+=4

break

case ’exit’:

WScript.Quit()

}

Пример программы вычисления суммы чисел от 1 до N, написанной на разрабатываемом языке.

write Вычисление суммы чисел.

write Введите число:

input 1000

mov 1001 0

mov 1002 1

metka 1003

add 1000 1001 1001

sub 1000 1002 1000

ifgreat 1000

goto 1003

write Сумма равна:

out 1001

Программа, написанная на JScript и реализующая виртуальную машину.

fso = new ActiveXObject ("Scripting.FileSystemObject");

fh = fso.OpenTextFile("d:\\JavaScriptNew1.txt");

var i=0;

var prog = new Array();

var lines ='';

while (!fh.AtEndOfStream){

lines = fh.Readline();

prog[i] = lines.split(' ');

i++;

}

fh.Close();

Nmax=prog.length

var N = 0;

while (N < Nmax)

{

switch (prog[N][0])

{

case "input":

//WScript.echo("Ввод:");

prog[Number(prog[N][1])] = parseInt(WScript.StdIn.ReadLine());

break;

case "mov":

//WScript.echo("Присваивание:");

prog[prog[N][1]] = Number(prog[N][2]);

break;

case "add":

//WScript.echo("Сложение:");

prog[prog[N][3]] = prog[prog[N][1]]+prog[prog[N][2]];

break;

case "sub":

//WScript.echo("Вычитание:");

prog[prog[N][3]] = prog[prog[N][1]]-prog[prog[N][2]];

break;

case "out":

//WScript.echo("Вывод:");

WScript.echo(prog[prog[N][1]]);

break;

case " ifgreat":

result = prog[prog[N][2]]>0;

if (!result)

N++;

break;

case "write":

for (var j=1; j<prog[N].length; j++)

WScript.echo(prog[N][j]);

break;

case "goto":

N = prog[prog[N][1]];

break;

case "metka":

prog[prog[N][1]]=N

break;

default:

WScript.echo('.');

}

N++;

}