

## Глава II. Программирование линейных алгоритмов

Приступая к лечению, врач должен делать это изящно и весело; хмурый врач никогда не преуспеет в своём ремесле.

—М.Монтень

Прежде чем переходить к непосредственному программированию, мы хотим дать два совета:

- овладеть основами программирования можно лишь при условии повседневной практики на вычислительной машине;
- не рекомендуем читать описание языка программирования «просто так», в расчёте на то, что «может быть, потом пригодится».

Выберите задачу и начинайте составлять программу её решения параллельно с лекционным курсом. Тогда Вы поймёте главное: зачем нужны и как используются те или иные конструкции языка. Если по постановке Ваша задача отличается от задач, разобранных нами в качестве примеров, ищите аналогию в действиях.

В процессе составления алгоритма и программирования учтите, что:

1. В любой сколь угодно малой программе есть, по крайней мере, одна ошибка.
2. Всякую программу можно сократить, по меньшей мере, на одну команду.
3. Начиная с некоторого уровня сложности программы, появление непредвиденной ошибки становится неизбежным.
4. Всякое непроверенное вычисление неверно.

—Из старинной книги «Заповеди программиста»

5. Человеку свойственно ошибаться, глупцу упорствовать (в своих ошибках).

—Марк Туллий Цицерон

6. В программировании как в гольфе: любой может научиться за месяц, но в конце месяца программируют так же, как и играют: *очень плохо*.

—J.R.Rice

7. ЭВМ многократно увеличивает некомпетентность вычислителя.

—Принцип некомпетентности Питера

8. Экспериментальное тестирование программ может служить для доказательства наличия ошибок, но никогда не докажет их отсутствие.

—Н.Вирт

9. Умствуй — и придет!

—Л.Ф.Магницкий

### II.1. Режимы работы

После одновременного нажатия на клавиатуре компьютера клавиш **CTRL** и **STOP** (что обозначается **CTRL+STOP**) на экран дисплея выводится «системная подсказка» «Ok» («o'ka» — «o'кей, все в порядке»).

С этого момента Вы можете работать в любом из двух режимов:

1. Режим прямого, или непосредственного, выполнения команд; при работе в режиме прямого выполнения команд на клавиатуре ЭВМ набирается последовательность команд (указаний, предписаний), которая воспроизводится на экране дисплея. Затем она вводится в оперативную память компьютера нажатием клавиши **Ввод**  которая имеет следующие названия:

- клавиша «RETURN»,
- клавиша «возврата каретки—перевода строки»,
- клавиша «ввода», «ENTER» («to enter» — «вводить»),
- клавиша «CR» («Carriage Return» — «возврат каретки»),
- клавиша «кляшка» (шутливое название).

Немедленно после этого вычислительное устройство ЭВМ выполняет данную последовательность команд. Сами команды после исполнения не запоминаются (теряются).



Важно помнить, что ввести информацию в компьютер — это значит не только набрать на клавиатуре соответствующий текст, но и нажать после этого клавишу Ввод!

Простейшие вычисления в режиме прямого выполнения команд производятся при помощи команд: LET и PRINT (см. разделы II.4.1. и II.4.4.).

Например, вычисление значения арифметического выражения  $SQR(3^2+4^2)$  выполняется по команде:

```
PRINT SQR(3^2+4^2)
```

После набора этой команды на клавиатуре и введения её в оперативную память компьютера (нажатием клавиши Ввод), производится вычисление значения арифметического выражения, стоящего справа от служебного слова PRINT.

В результате на экране дисплея остаётся следующая информация:

```
Ok
PRINT SQR(3^2+4^2)
5
Ok
```

Напомним, что признаком готовности ЭВМ к работе в режиме прямого выполнения команд является наличие «системной подсказки» «Ok» на экране дисплея. Пока «Ok» не выведено на экран дисплея, Вы не можете обратиться к ЭВМ: она занята своей работой и только по окончании её выводит на экран дисплея «системную подсказку» «Ok»!

Приведём ещё пример:

```
Ok
print "Стремиться к простоте программ."
```

Заметим, что служебное слово print при вводе можно заменять символом «?», например:

```
Ok
? "Стремиться к простоте программ."
```

После набора этой команды на клавиатуре и нажатия клавиши Ввод, экран дисплея примет следующий вид:

```
Ok
? "Стремиться к простоте программ."
Стремиться к простоте программ.
Ok
```

Таким образом, по указанной команде компьютер вывел на экран дисплея текст, который был заключён в кавычки. Появившееся после этого сообщение «Ok» свидетельствует о готовности интерпретатора (см. II.2.) «принять» новую команду.

Несколько следующих друг за другом команд прямого режима можно помещать на одной строке, которая называется *командной* строкой. При этом команды одной командной строки отделяются друг от друга двоеточием.

Максимально допустимое число символов в одной командной строке равно 255, поэтому длинная строка может занимать до семи 40-символьных или четырёх 80-символьных дисплейных (физических) строк. При вводе командной строки клавишу Ввод нажимают всего лишь один раз после набора последней команды данной строки. Если при вводе командной строки дисплейная строка уже заполнена, то дальнейший текст *автоматически* переносится на следующую дисплейную строку (без какого-либо символа переноса).

*Не опасайтесь этого!*

Прямой режим полезен при *отладке* программ и при использовании **MSX BASIC** в качестве калькулятора для проведения

коротких вычислений.

Примеры:

○ 1)

```
Ok
print (5^2-7)
18
Ok
```

○ 2)

```
Ok
? SIN(ATN(1)*4/2)
1
Ok
```

○ 3)

```
Ok
? LOG(EXP(1))
.999999999999986
Ok
```

○ 4)

```
Ok
? ATN(1)*4
3.1415926535898
Ok
```

○ 5)

```
Ok
? 25.68MOD6.99
1
Ok
```

○ 6)

```
Ok
? 15AND14
14
```

○ 7)

```
Ok
print 25.68\6.99
4
Ok
```

○ 8)

```
Ok
? (-1)OR(-2)
-1
Ok
```

○ 9)

```
Ok
print 3>1
-1
Ok
```

○ 10)

```
Ok
? "BONY"+"M"
BONY'M
Ok
```

○ 11)

```
Ok
? "1AAA"<"2aaa"
-1
Ok
```

○ 12)

```
Ok
? (4<=>4)AND(9<1)
0
Ok
```

o 13)

```
Ok
? 1/SIN(0)
Division by zero
Ok
```

o 14)

```
Ok
? 1/COS(2*ATN(1))
-15915494309189
Ok
```

o 15)

```
Ok
? TAN(2*ATN(1))
-15915494309189
Ok
```

o 16)

```
Ok
? (-32768%)/(-1%)
Overflow
Ok

однако...
Ok
? (-32768)/(-1)
32768
Ok
```

Заметим, что арифметические, логические и строковые выражения, стоящие после служебного слова PRINT, могут содержать имена переменных. Разумеется, этим переменным до вычисления значения выражения должны быть присвоены начальные значения. Это осуществляется при помощи команды присваивания LET.

o 17)

```
Ok
LET A#=-1/3: PRINT ABS(A#*2)+2
2.6666666666667
Ok
```

o 18)

```
Ok
DIM A(78):A(45)=SIN(4*ATN(1)):?A(45)
-1.2566370614359E-13
Ok
```

Отметим, что служебное слово LET при записи команды присваивания можно опускать.

o 19) Длину окружности и площадь круга радиуса R=2 можно вычислить при помощи следующей последовательности команд:

```
Ok
PI=4*ATN(1):R=2:PRINT 2*PI*R:PRINT PI*R^2
12.566370614359
12.566370614359
Ok
```

Более того, команда PRINT в прямом режиме позволяет мгновенно переводить числа из двоичной, восьмеричной и шестнадцатеричной систем счисления в десятичную, например:

o 20)

```
Ok
? &b11100111
231
Ok
```

o 21)

```
Ok
print &hffff
-1
Ok
```

○ 22)

```
Ok
print &04567
2423
Ok
```

○ 23)

```
Ok
? &hff-&o7+&b11
251
Ok
```

○ 24)

```
Ok
? &hffMOD&b10
1
Ok
```

○ 25)

```
Ok
? &03456/&HF2
7.595041322314
Ok
```

2. **Режим выполнения программы** (или **программный режим**); вначале ответим на вопрос: «Что же такое *программа* ?» В «Математической энциклопедии» (т.4, стр.646) читаем: «*Программа* — план действий, подлежащих выполнению некоторым исполнителем, обычно автоматическим устройством, чаще всего ЭВМ; предписание, алгоритм. Программа представляется в виде конечной совокупности *операторов* (команд, инструкций), каждый из которых побуждает исполнителя выполнить некоторую элементарную операцию над данными, хранящимися в памяти исполнителя, и имена которых являются параметрами оператора. Автоматизм исполнения достигается тем, что любой текущий оператор, кроме завершающего, указывает однозначно на оператор программы, который должен выполняться после текущего...».

Итак, вместо того, чтобы вводить команды в режиме непосредственного (немедленного) исполнения, можно составить из них программу, которая будет выполняться позже в нужный момент. Такой способ работы с ЭВМ называется *режимом выполнения программы* или *программным режимом*. В этом режиме команды обычно называются *операторами*, хотя часто оба термина употребляются как синонимы. Это связано с тем, что большинство команд режима прямого выполнения можно использовать в качестве операторов режима выполнения программы и наоборот.

Однако есть такие команды (например, AUTO, RENUM), которые целесообразно использовать только в режиме прямого выполнения, и такие операторы (например, DATA, DEFFN), которые выполняются только в программном режиме.

## II.2. Интерпретаторы и компиляторы

BASIC — язык программирования высокого уровня. Это значит, что он универсален, не связан с конкретной вычислительной машиной. Чтобы программа, написанная на таком языке, могла быть исполнена на реальной ЭВМ, необходима особая программа — *транслятор*, выполняющая роль переводчика на машинный язык. Благодаря наличию транслятора программист приобретает существенное удобство: ему нет нужды знать специфику конкретной машины. Написанная на языке высокого уровня, его программа становится, как говорят, «переносимой» с одной ЭВМ на другую.

Есть два основных типа трансляторов: *интерпретаторы* и *компиляторы*. Для BASIC типичны первые, для многих других языков программирования (например, Паскаля) — вторые.

*Интерпретатор* работает как синхронный переводчик: он пытается понять каждое очередное предложение (оператор) программы, то есть выразить его «смысл» в виде цепочки команд ЭВМ и тут же выполнить их.

По-другому действует *компилятор*. Если продолжить аналогию, он работает как создатель добротного литературного перевода; он переводит программу, как некий связный рассказ, учитывая взаимоотношения её частей и выявляя не только ошибки в отдельных элементах, но и несогласованности между ними. Требования к «правильности» программы у компилятора намного выше. Компиляция часто производится за несколько просмотров программы. Программа при этом не исполняется: перевод готов, но «не звучит». Одна из причин этого состоит в том,

что компилируется программа обычно по частям (*модулям*), особенно если она большая и сложная. Особая программа — *компоновщик* собирает отдельные модули, полученные после трансляции, в единое целое: окончательный, готовый к исполнению *машинный код*.

Уже из этих кратких характеристик понятно, что интерпретатор намного проще в обращении. Вследствие этого BASIC позволяет непосредственно ощутить процесс программирования. Программа может быть составлена (даже не полностью) и тут же исполнена «на пробу». Программист может вмешиваться в процесс исполнения и сразу вносить необходимые коррективы в работу программы, пока она ещё не свободна от ошибок, а затем быстро исправлять её текст с учётом пробного запуска и тут же повторять все эти действия снова. Для начинающего программиста, который часто ошибается, эти свойства языка, конечно, чрезвычайно привлекательны, поскольку ошибка во многих случаях обнаруживается легко и быстро.

В некоторых других языках программирования (Pascal) обнаружение ошибки в программе и её исправление сопряжено со значительно большими затратами труда и времени.

Во-первых, нужно ещё дождаться, пока можно будет попробовать программу в деле. Компиляция каждого модуля может длиться несколько минут, а ведь затем надо ещё собрать их воедино.

Во-вторых, даже при безошибочной компиляции отдельных частей отнюдь не гарантирована правильная работа программы в целом: части могут быть плохо пригнаны друг к другу. Предположим теперь, что ошибка обнаружена. Мы не можем исправить одно лишь ошибочное место программы так, как мы делаем это на BASIC. Придётся перекомпилировать весь модуль, иногда из-за одной строки — несколько сотен.

Однако, потом оттранслированная и скомпонованная программа будет работать намного быстрее, чем подобная программа на BASIC. Ведь интерпретатор BASIC переводит каждый оператор перед каждым его выполнением, а значит — каждый раз тратит время на перевод. Как синхронный переводчик, он, сказав, тут же забывает сказанное, и в следующий раз должен будет повторить всю проделанную работу по переводу. Создатель литературного перевода держит в уме весь текст целиком; он знает, что его перевод, сделанный однажды, будут потом читать тысячи читателей в течение многих лет, и работает над ним тщательнейшим образом. Компилятор выполняет свою работу, так же тщательно выверяя окончательный результат. После компиляции и сборки отлаженную программу можно запускать многократно, и затраты труда на её разработку так же, как и затраты времени на трансляцию, окупятся с лихвой.

Существует ещё один вид транслятора. В советском компьютере [БК-0010.01](#) используется компилятор с элементами интерпретатора, что позволяет в какой-то мере получить достоинства и того и другого. В частности, такой необычный «гибридный» транслятор по столь важному параметру, как быстродействие, ставит [БК-0010.01](#) в ряд популярных зарубежных *бытовых* компьютеров (т.е. ЭВМ для дома, для игр, развлечений, создания личного архива, картотеки, справочника, решения сравнительно простых задач).

## II.3. Оформление и редактирование программ

При записи программы на [MSX BASIC](#) необходимо соблюдать ряд требований, а именно:

1. Текст программы записывается *построчно*, чёткими *печатными* символами; при записи служебных слов разрешается использовать как прописные, так и строчные латинские буквы в любом сочетании, причём все строчные латинские буквы после ввода преобразуются интерпретатором в прописные (однако, символы, заключённые в кавычки, всегда выводятся на экран дисплея без изменения).
2. Каждая строка программы (программная строка) начинается *номером* этой строки и заканчивается нажатием клавиши «key»'Ввод '↵' до нажатия этой клавиши ЭВМ «не интересуется» тем, что пользователь написал на экране дисплея, а после нажатия «пристывает» к анализу содержимого программной строки. Интерпретатор игнорирует все пробелы между номером строки и началом оператора в этой строке (аналогично не учитывается и большинство остальных пробелов (хотя и не всегда!)).
3. Программные строки нумеруются целыми числами от 0 до 65529 в возрастающем порядке.
4. При составлении начального варианта программы рекомендуется нумеровать строки с интервалом 10: 10, 20, 30, .... Нумерация программных строк через 10 позволяет дополнять программу, вставляя новые программные строки между введёнными в память ранее (новые строки при этом имеют промежуточные номера: 15, 17, 21, 23 и т.д. и, даже будучи введёнными с клавиатуры в разное время, размещаются в памяти ЭВМ в строгом соответствии со своими номерами).
5. В каждой программной строке старайтесь размещать один *законченный* оператор (в случае размещения в строке нескольких законченных операторов операторы внутри строки отделяются друг от друга символом « : » (двоеточие), причём после последнего оператора в программной строке двоеточие не ставится); строка программы может занимать до *семи* дисплейных строк на 40-символьном экране дисплея, так как

максимальное число знаков в строке программы, включая номер строки, равно 255. Избегайте «синдрома одной строки»! Данная «болезнь» начинающего программиста заключается в старательном размещении в каждой программной строке «огромного» количества операторов, в результате чего программа становится совершенно «нечитабельной».

6. Последним оператором программы может быть оператор окончания программы END с другой стороны, иногда можно обойтись и без этого оператора: выполнив последнюю строку программы, ЭВМ останавливается сама — ей больше нечего делать, операторы (команды) кончились!
7. Для записи программы в память ЭВМ и для её исполнения необходимо сопроводить программу *командами* (разумеется, они не нумеруются!) NEW и RUN («new» — «новый», «to run» — «бежать»), первая из которых записывается на первой перед программой дисплейной строке, вторая — на свободной дисплейной строке, расположенной за последней строкой программы. Обе команды заканчиваются нажатием клавиши `Ввод ↵`.

По команде NEW из оперативной памяти удаляются все предыдущие программы пользователя и значения переменных. Обязательно используйте эту команду для уничтожения старых программ, прежде чем начать вводить новую программу, иначе Вы только «измените» Вашу старую программу.

Отметим, что команда NEW является и оператором!

Команда RUN в её простейшей форме (в виде одного служебного слова) является сигналом к выполнению введённой программы, начиная с программной строки с наименьшим номером.

Команда RUN является также и *оператором*. Если эта команда задана как строка Вашей программы, то она перезапускает Вашу программу таким образом, как будто она была сначала остановлена, а затем запущена снова.

Также существует модификация команды RUN, позволяющая запускать программу с *любой другой строки*: для этого после служебного слова RUN указывается номер существующей программной строки, с которой программист пожелает начать выполнение программы, например:

```
RUN 23
```

Отличительной особенностью команды (оператора) RUN *n* является «стирание» значений всех переменных. Таким образом достигается идентичное состояние памяти при каждом новом запуске программы.

Запомните, что если Вы хотите начать выполнение программы со строки с заданным номером *n*, но при этом сохранить значения всех переменных, не используйте команду

```
RUN n
```

а примените в режиме прямого выполнения команду

```
GOTO n
```

или

```
GOSUB n
```

(см. разделы III.1 и IV.4).

Разумеется, номер строки, указанной в операторе RUN не должен находиться внутри цикла или подпрограммы!

Таким образом, программы на BASIC имеют такой общий вид:

```
NEW  
Ok  
10 оператор 1  
20 оператор 2  
30 оператор 3:оператор 4  
...  
NN [END]  
run
```

Заметим, что здесь и всюду далее *квадратные скобки* — обозначение, указывающее, что информация внутри них не

является обязательной.

Напомним ещё раз, что при вводе программы в память в конце каждой программной строки надо нажимать клавишу Ввод!

Нумерация строк программы имеет целью:

1. отметить каждую программную строку индивидуальной меткой — *номером*, что позволяет обратиться к любой строке из любой другой программной строки;
2. упорядочить расположение операторов в программе, которая обрабатывается интерпретатором построчно;
3. существенно облегчить процедуру *редактирования* (исправления) программы, так как интерпретатор выдаёт сообщение об ошибке в конкретной программной строке, указывая её номер, и пользователь получает возможность либо исключить строку из программы, либо записать в память ЭВМ откорректированный (исправленный) текст строки.

Рассмотрим несколько команд редактирования программ.

### II.3.1. Команда AUTO. Команда RENUM

Автоматизировать процедуру присвоения номеров программным строкам можно командой

```
AUTO [n1] [, n2] ,
```

где:

- AUTO («AUTOmatic» — «автоматический») — служебное слово;
- n1, n2 — целые числовые константы;  $0 \leq n1, n2 \leq 65529$ .

По этой команде происходит задание режима автоматической нумерации программных строк от строки с номером n1 и далее с шагом n2.

По умолчанию n1=10 и n2=10. Таким образом, команда AUTO без параметров позволяет автоматически нумеровать программные строки через 10, начиная с программной строки с номером 10.

Нажатие клавиш CTRL+STOP отменяет действие команды AUTO.

Если очередной номер строки, сгенерированной режимом автоматической нумерации, совпадает с номером уже находящейся в памяти строки (введённой ранее), то на экране дисплея индицируется номер этой программной строки со звёздочкой «\*», например: 55\*. В этом случае, если мы хотим сохранить «старое» значение этой программной строки, мы должны нажать клавишу Ввод — компьютер сохранит эту программную строку и перейдёт к генерации следующего номера.

Общий вид команды изменения нумерации программных строк:

```
RENUM [n1] [, n2] [, n3] ,
```

где:

- RENUM («to RENUMber» — «перенумеровать») — служебное слово;
- n1, n2, n3 — целые константы,  $0 \leq n1, n2, n3 \leq 65529$ .

По команде RENUM производится перенумерация (изменение нумерации) программных строк, начиная со строки с номером n2 и дальше с шагом n3. Причём, n1 — это новый номер прежней строки n2. По умолчанию: n1=10, n3=10, n2 — номер начальной строки программы. Параметр n2 может быть опущен, но запятая, относящаяся к нему, должна быть сохранена. В качестве параметров n1 и n2 может быть использован символ «.», который является ссылкой на «текущий» номер программной строки (имеется в виду последний, упомянутый номер программной строки).

Команда RENUM, кроме того, проверяет, все ли программные строки, на которые имелись ссылки в операторах

программы, действительно существуют (независимо от этого осуществляется перенумерация всех строк); если программная строка с номером  $n$  не существует, а на неё имеется ссылка в строке  $m$ , то выдаётся сообщение:

«Undefined line  $n$  in  $m$ »  
(«Отсутствует строка  $n$  в  $m$ »).

Перенумеровать операторы можно:

1. перед последующим слиянием программ;
2. для проверки существования строк программы, на которые имеются ссылки. Для этого используйте команду RENUM с параметром  $n2$ , превышающим все существующие номера строк, например,

```
RENUM 65000,65000
```

Перенумерация при этом не осуществляется, но Вы получите информацию обо всех пропущенных номерах строк.

*Примеры:*

```
RENUM  
RENUM 200,150,5  
RENUM 200,100
```

Обратите внимание: новый номер строки должен быть больше или равен старому номеру!

### II.3.2. Команда DELETE. Команда [L]LIST

Куда девал сокровища убиенной тобою тёщи?

—Отец Фёдор

Команда DELETE может быть записана в одной из следующих форм:

```
DELETE n1-n2  
DELETE -n2  
DELETE n1  
DELETE .  
DELETE .-n2  
DELETE n1- .
```

Здесь:

- DELETE («to delete» — «стереть») — служебное слово;
- $n1$  и  $n2$  — целые константы,  $0 \leq n1$ ,  $n2 \leq 65529$ .

По команде DELETE из программы удаляются программные строки с номерами от  $n1$  до  $n2$  включительно. Если параметр  $n1$  отсутствует, то в качестве  $n1$  берётся номер первой строки программы. Символ «.», используемый вместо номера строки обозначает «текущий» (последний упомянутый) номер программной строки.

*Примеры:*

```
DELETE 70-100  
DELETE -50  
DELETE 23  
DELETE .  
DELETE .-34  
DELETE 40- .
```

Если строка с номером, указанным в команде DELETE, в программе отсутствует, то выдаётся сообщение об ошибке

«Illegal function call»  
(«Неправильный вызов функции»).

Формат команды [L]LIST:

```
[L]LIST [n1] [-n2]
```

где:

- [L]LIST(«Line printer LIST out», «list» — «список») — служебное слово;
- n1, n2 — целые константы ( $0 \leq n1, n2 \leq 65529$ ) или десятичная точка («.»).

Эта команда, которая может быть оператором программы, выводит текст программы, находящейся в данный момент в памяти. Если указано служебное слово LIST, то вывод происходит на экран дисплея; если используется служебное слово LLIST, то текст отсылается на печатающее устройство (*принтер*) и печатается на бумаге.

Если параметры не указаны, то выводится на экран (или принтер) вся программа. Если после служебного слова [L]LIST задан номер программной строки, то выводится только строка с указанным номером. Тире до или после этого номера означает, что будет выведен весь текст программы соответственно до или после строки с этим номером. Два номера программных строк, разделённые тире, указывают, что должны быть выведены только строки программы с номерами, находящимися внутри данного диапазона.

Чтобы указать на «текущий» номер программной строки, Вы можете использовать символ «.» вместо параметров n1 и n2.

Клавиша **STOP** используется для организации паузы в процессе вывода текста программы на экран дисплея или принтер. Для продолжения вывода программы следует нажать клавишу **STOP** вторично.

*Пример.*

```
NEW
Ok
200 DEFSTR A-C
150 DEFDBL C-K
100 DEFSNG K-M
LIST .
100 DEFSNG K-M
Ok

LIST -175
100 DEFSNG K-M
150 DEFDBL C-K
Ok
```

## II.4 Линейные программы

Напомним, что в школьном алгоритмическом языке *линейными* назывались алгоритмы, состоящие из серии команд присваивания.

Для программирования линейных алгоритмов практически необходимы знания правил употребления только трёх операторов: оператора ввода данных, оператора присваивания, оператора вывода данных. Эти операторы, естественно, применяются и в более сложных программах.

*Линейной* будем называть программу, в которой все операторы, начиная с первого, выполняются последовательно один за другим, причём выполняется каждый оператор (притом, только *один* раз).

## II.4.1. Оператор присваивания LET

Чтобы выпить из чаши, нужно её наполнить!

—Восточная мудрость

Структура оператора присваивания:

```
[LET] переменная = выражение
```

где:

- LET («пусть») — служебное слово;
- *переменная* — имя новой или существующей переменной или имя элемента массива, или имя псевдопеременной (см. [раздел I.7.5.](#));
- «=» — знак присваивания;
- *выражение* — либо строковое, либо арифметическое выражение того же типа, что и *переменная*, либо логическое выражение.

В результате выполнения оператора присваивания *переменная* (имя которой стоит слева от знака присваивания) приобретает новое значение (являющееся значением *выражения*, стоящего справа от знака присваивания). **MSX BASIC** — язык арифметики, а не алгебры: оператор LET присваивает переменной слева именно *число*, а никак не алгебраическое выражение, стоящее справа!

Примеры:

- 1)

```
NEW
Ok
10 LET A1=10.04
20 END
```

Данная программа позволяет присвоить переменной A1 начальное значение 10.04 (инициализировать её). Проверим это. Сначала выполним программу:

```
run
Ok
```

А теперь...

```
print A1
10.04
Ok
```

- 2)

```
NEW
Ok
10 P1=3.14159
20 P2=INT(P1/0.001+.5)*0.001
run
Ok
print P1
3.14159
Ok
print P2
3.142
Ok
```

- 3)

```
NEW
Ok
```

```
10 LET X=5.4
20 LET Y=FIX(X+.5)
30 END
run
Ok
print Y
5
Ok
```

- 4)

```
NEW
Ok
10 LET Z=5>=-1
20 END
run
Ok
print Z
-1
Ok
```

Обратите внимание на то, что оператор, расположенный в программной строке 20 примера 2, позволил округлить значение переменной X.

Два следующих оператора присваивания позволят вычислить старший и младший байты целого числа A%:

```
H%=A%\256:L%=A%MOD256
```

Отметим, что служебное слово LET в записи оператора присваивания можно опускать, что не меняет сущности оператора присваивания; так, например, идентичны следующие фрагменты программ:

```
20 LET M%=A%-INT(A%/B%)*B%
```

и

```
20 M%=A%-INT(A%/B%)*B%
```

Хотелось бы отметить, что начинающих программистов часто ставит в тупик, например, оператор вида:

```
X = X + 1
```

В математике такая запись бессмысленна. Но по определению оператора присваивания такая запись вполне законна: вначале вычисляется значение выражения, стоящего в правой части (значение переменной X, существовавшее до момента выполнения этого оператора (старое значение), увеличивается на единицу), и полученное значение присваивается той же переменной X.

Ещё один пример применения оператора присваивания и операции вычисления остатка для организации циклических изменений значений целых числовых переменных:

```
X%=(X%+1)MOD256
```

Этот оператор позволяет увеличивать X% с 0 до 255. Когда значение X% будет равно 255, оператор присваивания тут же установит значение X% равным 0.

Ещё раз обратите внимание, что символ «= $\equiv$ » здесь отнюдь не совпадает по смыслу с символом равенства «= $=$ » в математике, несмотря на одинаковую форму и обманчивое сходное использование!

Оператор присваивания обладает следующими особенностями.

Если переменной некоторого типа присваивается значение выражения другого типа, то тип выражения преобразуется к типу переменной. Попытка присвоить значение строкового выражения числовой переменной или наоборот, вызовет сообщение об ошибке

«Type mismatch»

(«Несоответствие типов»).

Примеры:

- 1)

```
Ok
A%=23.42:print A%
23
Ok
```

- 2)

```
Ok
A="нельзя!":print A
Type mismatch
Ok
```

- 3)

```
Ok
a%=123456:? a%
Overflow
Ok
```

- 4)

```
Ok
b!=123456.78:? b!
123457
Ok
```

- 5)

```
Ok
d!=123456789:?d!
123457000
Ok
```

Если переменной целого типа присваивается значение выражения, вычисленное с одинарной или двойной точностью, то у значения выражения отбрасывается дробная часть.

- 6)

```
Ok
C%=55.88:print C%
55
Ok
```

- 7)

```
Ok
C%=-1.97:? C%
-1
Ok
```

Если переменной двойной точности присваивается значение выражения, вычисленное с одинарной точностью, то только первые 6 цифр значения выражения являются точными (напомним, что одинарная точность обеспечивает 6 значащих цифр результата), т.е. 8 разрядов, добавляемых к значению, всегда равны нулю. Некоторые компьютеры при этом добавляют цепочку непредсказуемых разрядов, из-за чего следует быть очень осторожным, чтобы в вычислениях, требующих высокой точности, не возникали абсурдные результаты!

- 8)

```
Ok
A!=SQR(2):B=A!:PRINT A!,B
1.41421      1.41421
```

```
Ok
```

• 9)

```
Ok
A=TAN(2*ATN(1)):B!=TAN(2*ATN(1)):C=B!?: A;B!;C
-15915494309189 -159155000000000
-159155000000000
Ok
```

Перед отбрасыванием «лишних» значащих цифр производится *округление*: анализируется старшая из отбрасываемых цифр и если она больше или равна 5, то младшая из оставшихся цифр увеличивается на 1.

При вычислении значения выражения все операнды во встречающихся операциях и результат вычислений приводятся к одной и той же степени точности — к наиболее высокой.

Учтите, что наивысшая степень точности — у операндов двойной точности, далее следуют в порядке убывания степеней точности: операнды с одинарной точностью и целые операнды.

*Пример:*

```
Ok
D=6/7!:print D
.85714285714286
Ok
```

Десятичная константа 6 — двойной точности, константа 7 имеет тип одинарная точность, а переменная D по умолчанию имеет тип двойная точность, поэтому арифметическое действие было выполнено с двойной точностью, и полученный результат имеет также тип двойная точность.

*Пример:*

```
Ok
D!=6/7:print D!
.857143
Ok
```

Арифметическое действие было выполнено с двойной точностью (оба операнда по умолчанию имеют двойную точность!), результат присваивается переменной одинарной точности, поэтому перед присваиванием выполнено округление и выведено значение переменной D одинарной точности.

Оператор присваивания — наиболее употребительный во всех языках программирования. В BASIC он часто используется также в качестве оператора ввода данных.

Наконец, заметим, что в языках программирования легко сказать «запомни», но сложно говорить «забуди»!

## II.4.2. Оператор SWAP

Оператор SWAP используется для обмена значениями двух переменных и записывается в виде:

```
SWAP α, β
```

или

```
SWAP γ, σ
```

где:

- SWAP («swap» — «обмен») — служебное слово;
- α, β — имена числовых переменных или имена элементов числового массива (числовых массивов);
- γ, σ — имена строковых переменных или имена элементов строкового массива (строковых массивов).

Этот оператор позволяет осуществлять обмен значениями двух переменных одинаковых типов. Обычно, не зная оператора SWAP, программисты используют для обмена значениями следующие приёмы:

```
TEMP=A : A=B : B=TEMP
```

 или

```
A=A+B : B=A - B : A=A - B
```

Оператор SWAP позволяет выполнить обмен за одну операцию, т.е. со значительным увеличением скорости и экономией места.

Заметим, что к двум переменным можно применить оператор SWAP только в том случае, если предварительно им присвоены определённые значения!

*Примеры:*

- 1)

```
A=3 : B=2 : SWAP A, B : PRINT A ; B
2 3
Ok
```

- 2)

```
C$="мама" : D$="папа" : SWAP C$, D$ : PRINT C$ ; D$
папамама
Ok
```

Ещё раз отметим, что переменные, заданные в качестве аргументов, должны быть одного и того же типа, причём в отличие от других операторов, это требование более жёсткое, так, например, нельзя обменивать значения переменных одинарной и двойной точности. Нарушение этого требования приводит к сообщению об ошибке:

«Type mismatch»  
(«Несоответствие типов»).

Наиболее эффективно применение оператора SWAP при сортировке массивов (см. пример в [разделе III.4.](#))!

### II.4.3. Оператор комментария REM

Оператор REM имеет следующий синтаксис:

```
REM Текст комментария
```

После служебного слова REM и до конца данной программной строки может следовать любой текст. Оператор REM не оказывает никакого влияния на ход выполнения программы; единственное его назначение — сделать программу более ясной и понятной. Весь текст строки программы, следующий за служебным словом REM, воспринимается интерпретатором как некоторое поясняющее замечание. Поэтому, если комментарий является частью программной строки, содержащей несколько операторов, то соответствующий оператор REM должен быть *последним* оператором данной программной строки.

Например,

```
10 Z=10.1 : REM Начальное значение присвоено!
```

Другой способ включения в текст программы комментариев — это использование *апострофа* («'»). Все символы, стоящие в программной строке за апострофом, рассматриваются как поясняющее сообщение. Если комментарий стоит в конце строки, содержащей несколько операторов, а вместо слова REM используется апостроф, то двоеточие перед апострофом не ставится (за исключением оператора DATA).

Например,

```
95 D=P*Q 'Уточнение коэффициента D!
```

После завершения написания и *отладки* программы Вы можете создать вариант Вашей программы только для счёта, исключив все комментарии.

Это приведёт к уменьшению времени выполнения Вашей программы и уменьшению объёма используемой оперативной памяти.

Забегая вперёд, отметим, что под *отладкой программ* понимается обычно один из этапов решения, во время которого с помощью компьютера происходит обнаружение и исправление ошибок, имеющихся в программе; в ходе отладки программист хочет добиться определённой степени уверенности в том, что его программа соответствует своему назначению и не делает того, для чего она не предназначена.

Многие начинающие программисты обычно пренебрегают комментариями или приписывают сообщения, не содержащие полезной информации, к готовой программе. Обычно они утверждают, что и без комментариев знают все, что в ней происходит. Это, конечно так, но через несколько недель обязательно что-нибудь да забудется! Начинающие программисты считают, что им никогда не придётся модифицировать уже отлаженную программу. Это грубая ошибка. Необходимость в неожиданных добавлениях и изменениях готовой программы может привести к хаосу и недоразумениям, когда, к примеру, забыто назначение некоторого оператора, либо неизвестно, зачем в данном месте программы обнуляется некоторая переменная.

Как же избежать подобной ситуации? Ответ прост. Вам следует всегда очень подробно документировать текст Вашей программы ещё во время её создания. В этом заключается один из способов застраховать себя от непредсказуемых заранее неприятностей.

Существует *три уровня* документирования, которых следует придерживаться.

*Первый уровень* — это общее описание программы в начальном блоке комментариев. Он включает в себя даты создания и последующих модификаций программы, номер её версии, имя автора. Здесь же указываются аргументы и результаты работы алгоритма, по которому составлена программа. Кроме того, следует описать используемый алгоритм шаг за шагом. Эта часть может быть использована в качестве введения для неискущённого пользователя, разъясняющая ему, что делает Ваша программа и как ею пользоваться.

Более того многие программисты считают, что перед непосредственным программированием необходимо написать инструкцию по использованию будущей программы!

*Второй уровень* состоит в документировании каждого блока программы (либо каждой подпрограммы) (см. [раздел IV.4.](#)). В каждом блоке программы следует отмечать комментарием его начало, где упомянуть, что он соответствует определённым шагам, описанным ранее в алгоритме. Начало любой подпрограммы следует снабжать комментариями, объясняющими, что же все-таки она делает.

Наконец, *третий уровень* документирования — построчный. Необходимо пояснять любую необычную комбинацию операторов или любой «хитроумный» приём программирования.

Комментируйте так, как будто бы Вы отвечаете на вопросы читателя. Очень эффективный метод состоит в том, чтобы сначала написать текст программы без комментариев, а затем посмотреть на него глазами читателя. Если окажется, что у читателя в некоторой точке программы может возникнуть вопрос, следует вставить содержательный комментарий с ответом на него.

Физически «выдвигайте» все комментарии из текста собственно программы. При печати комментарии следует смещать вправо от текста программы так, чтобы читатель мог просматривать программу, не прерываемую комментариями.

Прокомментируйте все переменные. Понимание данных — ключ к пониманию программы. Каждое предложение, объявляющее некоторую переменную, должно сопровождаться комментарием, поясняющим смысл этой переменной.

Если Вы будете придерживаться этих рекомендаций, то время, которое Вы потратите написание подробных комментариев с лихвой окупится тем, что через полгода(!) Вы будете понимать свою собственную программу!

Кроме того, для программиста, который первый раз видит чужую программу и хочет в ней разобраться, комментарии и примечания на обычном человеческом языке, поясняющие логику, структуру, конкретные операторы программы, играют неоценимую роль. Хорошие, понятные комментарии — это элемент не только культуры программирования, но и культуры общения, показатель отношения автора программы к коллегам и потенциальным потребителям программы в целом.

## II.4.4. Оператор вывода данных PRINT в простейшем случае

...мало толку в программе, которая только и делает, что секретничает сама с собой.

—Л.Хэнкок, М.Кригер

Оператор вывода данных на экран дисплея в простейшем случае состоит из служебного слова PRINT («to print» — «печатать»), за которым следует список выражений любого типа, значения которых должны быть выведены на экран дисплея. Служебное слово PRINT для сокращения записи может быть заменено символом «?».

Выражения в составе списка разделяются в простейшем случае символами *разделителями*: либо запятой, либо точкой с запятой, что определяет взаимное расположение выводимых значений на экране дисплея (впрочем, символы-разделители можно иногда опускать; см. ниже [пример 11](#)).

Оператор PRINT в простейшем случае выполняет *автоматическое форматирование*, при котором распределение элементов данных на экране осуществляется соответственно символам-разделителям, расположенным между ними.

Если выражения в списке оператора PRINT разделены *запятой*, то указанный символ служит интерпретатору сигналом разделить дисплейную строку на зоны, по 14 позиций каждая (ясно, что количество полных зон может быть не больше двух в случае 40-символьного экрана!) и выводить значения выражений последовательно с первой позиции каждой зоны, т.е. обеспечить вывод значений в *зонном* формате (отметим, что позиции дисплейной строки нумеруются, начиная с нуля!).

Если на экран выводятся значения более двух выражений, то значение третьего и всех последующих выражений отображаются в начале тех же зон, но с новой дисплейной строки. Если данные занимают не более 13 позиций, то следующая зона располагается прямо за концом данных без вывода пробелов.

Примеры:

- 1)

```
NEW
Ok
10 X=2:Y=X^2
20 PRINT X,Y
30 END
run
·2··········▲4
Ok
      |
      15—я позиция (начало второй зоны)
```

```
или
NEW
Ok
10 X=2
20 PRINT X,X^2
30 END
run
·2··········▲4
Ok
      |
```

В том месте дисплейной строки, где надо обратить Ваше внимание на наличие символа пробел (« »), мы будем использовать в тексте символ «>»!

Если же данные занимают более 13-ти позиций, то следующая зона пропускается.

- 2)

```
PRINT 1234567.34,"TIMER"
·1234567.34···TIMER
Ok
      ▲
      15—я позиция
```

- 3)

```
? 123456.789101124,"TIMER"
1234567890
·123456.78910112
TIMER          ▲
Ok             |
              16—я позиция
```

При наличии *точки с запятой* в качестве символа-разделителя выражений в списке оператора PRINT, компьютер выводит значения в *плотном* формате (значения выражений отделены друг от друга *одним* пробелом).

Оператор PRINT без параметров вызывает переход в начало первой зоны следующей дисплейной строки.

- 4)

```
NEW
Ok
10 X=3
20 PRINT X;X^3
25 PRINT X;-X^4
30 END
run
·3··27
·3·-81
Ok
```

- 5)

```
NEW
Ok
10 X=3
20 PRINT X:PRINT:PRINT X^3
30 END
run
·3
·27
Ok
```

Если в списке встречаются лишние запятые, соответствующие пустым элементам данных, то пропускается соответствующее число зон печати.

Например, для вывода на экран дисплея данных не с первой, а со второй зоны используйте оператор:

```
PRINT ,A,B
```

Поговорим теперь о формате вывода числовых данных. Прежде всего выводится знак числа. Если число отрицательное, выводится знак «-», в случае положительного числа выводится пробел. Более того, пробел также помещается за каждым выведенным числовым значением.

Если возможно, интерпретатор «избегает» индикации на экране всех десятичных позиций выводимого значения и выводит числа в форме с плавающей точкой со стандартной мантиссой (это происходит, когда целая часть выводимого числа содержит не менее *пятнадцати* значащих цифр или когда число по модулю  $\leq 0.001$ ), например:

- 6)

```
Ok
PRINT 12345678910112.1434
·12345678910112
Ok
```

- 7)

```
Ok
PRINT 123456789101121.434
·1.2345678910112E+14
Ok
```

Заметим, что каждый оператор PRINT выводит информацию на экран дисплея, начиная с первой позиции новой строки. Однако, имеется возможность продолжить вывод с помощью оператора PRINT на той же строке, где он закончился при выполнении предшествующего оператора PRINT. Для этого достаточно поставить в конце списка выражений, содержащегося в предшествующем операторе PRINT, символ-разделитель « , » или « ; ».

• 8)

```
NEW
Ok
10 X=3
20 PRINT X;X^3;
25 PRINT X;X^4
30 END
run
·3··27··3··81
Ok
```

• 9)

```
NEW
Ok
10 X=3
20 PRINT X;X^3,
25 PRINT X;X^4
30 END
run
·3··27······▲3··81
Ok
      |
      15—я позиция
```

• 10)

```
NEW
Ok
10 ?"YAMAHA";
20 PRINT:PRINT!"
run
YAMAHA
!
Ok
```

С помощью оператора PRINT можно (и должно), используя строковую константу, (последовательность символов, заключённую в кавычки), организовать более информативный, *красивый* вывод результатов работы программы.

Пример 11.

[0244-11.bas](#)

 [0244-11.bas](#)

```
NEW
Ok
10 X=5.08:Y=15.5
20 PRINT "РЕЗУЛЬТАТЫ"
30 PRINT "X = ";X,"Y = ";Y
31 PRINT "X = "X"Y = "Y
40 END
run
РЕЗУЛЬТАТЫ
X·=··5.08······Y·=··15.5
      ▲
      |
      15—я позиция (начало второй зоны)

X·=··5.08·Y·=··15.5
Ok
```

Опишем интересные *дополнительные* возможности оператора PRINT:

1. существует возможность начать вывод значения некоторого выражения с точно указанной позиции дисплейной строки (с позиции N). Для этого достаточно поместить в операторе PRINT перед этим выражением функцию

TAB( $\alpha$ )

где:

- TAB («to TABulate» — «табулировать») — служебное слово;
- $\alpha$  — арифметическое выражение, целая часть значения которого лежит на отрезке [0,255]; обозначим  $N = \text{INT}(\alpha)$ .

Функция TAB( $\alpha$ ) сообщает оператору PRINT, что от текущей позиции курсора на дисплейной строке и до позиции с указанным номером N включительно надо вывести пробелы (*курсор*— специальный знак «█», отмечающий одну из позиций ввода на экране дисплея). Если значение N *меньше* номера позиции текущего положения курсора, то перемещения не происходит.

Напомним Вам, что номера позиций курсора на дисплейной 40-символьной строке следующие: 0, 1, 2, ..., 39.

Пример 12.

[0244-12.bas](#)

 0244-12.bas

```
NEW
Ok
10 V=1:S=3.5
20 PRINT TAB(3);"V=";V;TAB(11);"S=";S
30 END
run
..▲V=.1...▲S=.3.5
 |         |
 | 11—я позиция
 | 3—я позиция
Ok
```

Пример 13.

[0244-13.bas](#)

 0244-13.bas

```
NEW
Ok
10 V=1:INPUT M,N 'Ширина экрана - 39 позиций!
20 PRINT TAB(M);"V=";V;TAB(N);"V=";V
run
? 1,5
.V=.1.V=.1
Ok

run
? 2,6
..V=.1.V=.1
Ok

run
? 1,7
.V=.1.V=.1
Ok

run
? 4,43
...V=.1...
...V=.1
Ok
      ▲
      |
39—я позиция
```

Забегая вперёд, отметим, что номер позиции текущего положения курсора на дисплейной строке можно получить, используя встроенную функцию POS( $\theta$ ) (см. [раздел V.3.](#)).

Кроме того, функция TAB( $n$ ) позволяет программисту получить удобное размещение колонок выводимых результатов, если вывод по зонам, задаваемым запятой (через 14 позиций), слишком широк.

Пример 14.

0244-14.bas  
 0244-14.bas

```
NEW
Ok
10 INPUT X,A,B,C
20 PRINT A;TAB(X);B;TAB(2*X);C'Символы";"можно опускать!
run
? 5,2,2,2
·2··▲·2··▲·2
  |      |
  |      |
5—я позиция |
  10—я позиция (2·5)
Ok

run
? 3,2,2,2
·2··2··2
Ok
```

1. В операторе (команде) PRINT часто используется функция

SPC( $\alpha$ )

где:

- SPC(«SPaCe» — «пространство») — служебное слово;
- $\alpha$  — арифметическое выражение, целая часть значения которого лежит на отрезке  $[0,255]$ .

Обозначим  $N$  — целую часть значения арифметического выражения  $\alpha$  ( $N \in [0,255]$ ). Тогда функция SPC( $\alpha$ ) выведет  $N$  пробелов, начиная с текущей позиции курсора.

*Пример 15.*

0244-15.bas  
 0244-15.bas

```
NEW
Ok
10 INPUT Z,U,V,W
20 PRINT U;SPC(Z);V;SPC(Z);W
run
? 3,2,2,2

▲2▲····▲2▲····▲2
| | | | |
пробелы для знака
| |
обязательные пробелы после числа
```

Запомните, что встроенные функции TAB() и SPC() допустимы *только* в операторе (команде) PRINT!

## II.4.5. Операторы ввода данных DATA и READ. Оператор RESTORE

Блоком данных будем называть последовательность констант. С блоком данных связана некоторая величина, называемая *указателем считывания*. Она всегда «настроена» на конкретную константу блока данных. При запуске программы на счёт указатель «метит» позицию первой константы блока.

Оператор DATA предназначен для хранения данных в блоке данных и для ввода его в программу.

Структура оператора DATA:

```
DATA β1[,β2][,β3] ...
```

где:

- DATA («data» — «данные») — служебное слово;
- β1, β2, β3, ... — числовые константы или строковые константы в кавычках или без кавычек (но в этом случае не содержащие запятых, кавычек и двоеточий (, " : )).

Оператор DATA позволяет хранить «начальные» значения в теле программы и может находиться в любом её месте. В режиме прямого выполнения команд оператор DATA игнорируется.

*Основной характеристикой* оператора DATA является его полное игнорирование и пропуск во время выполнения программы — он используется только тогда, когда в программе встречается оператор READ. Каждый оператор DATA считается частью общего «банка данных» программы. Поэтому 10 разбросанных по всей программе операторов DATA, каждый из которых содержит одну константу, аналогичны одному оператору DATA, содержащему все константы из этих операторов.

При чтении данные берутся из самого первого встретившегося в программе оператора DATA (в смысле номеров строк); все последующие операторы DATA выбираются по очереди. Для повторного считывания данных из блока данных следует использовать оператор RESTORE (см. данный [раздел ниже](#)).

Апостроф «'» не считается признаком конца оператора DATA и началом комментария: в операторе DATA такую роль играет двоеточие «:» и следующий за ним апостроф «'».

Например:

```
DATA 5, "кабан",ZE-4 :!Начальные значения!
```

В операторе DATA допускается использование числовых констант любых типов (но не выражений или имён переменных): в частности числовые константы могут содержать префиксы &H, &O, &B или могут быть записаны в экспоненциальной форме, а также могут иметь указатель типа (!, %, #).

Оператор DATA — это самый короткий способ инициализации переменных и массивов в программе. Он очень удобен при документировании программ.

Оператор READ предназначен для организации выборки (чтения) данных из блока данных. Формат оператора READ:

```
READ α1[,α2][,α3] ...
```

где:

- READ («to read» — «читать») — служебное слово;
- α1, α2, α3, ... — имена переменных или имена элементов массива(ов).

При выполнении оператора READ из блока данных последовательно считываются константы β1, β2, ... и их значения присваиваются соответствующим переменным α1, α2, ... в операторе READ. Считывание начинается с той константы, на которую «настроен» указатель считывания. При этом сам указатель при каждом считывании смещается по блоку данных на одну позицию вправо. Этот процесс повторяется до исчерпания всех переменных в списке оператора READ или до исчерпания данных в списке оператора DATA. В последнем случае выводится сообщение об ошибке:

«Out of DATA»  
(«Данные исчерпаны»).

После считывания последней константы блока значение указателя не определено.

Типы констант в операторе DATA и типы переменных в операторе READ должны совпадать! В противном случае выводится сообщение об ошибке:

«Syntax error»

(«Синтаксическая ошибка»).

Пример 1.

[0245-01.bas](#)

 [0245-01.bas](#)

Написать программу вычисления значения функции Y

$$\sqrt{\left(\frac{A \times B^2}{(K - M)^2}\right)}$$

при следующих значениях аргументов:

a=5.3; b=14.7; k=1.44; m=0.508.

```
NEW
Ok
10 DATA 5.3,14.7,1.44,0.508
20 READ A,B,K,M
30 Y=SQR(A*B^2/(K-M)^2):PRINT"Y=";Y
40 END
run
Y= 36.311095958874
Ok
```

В результате совместного выполнения операторов READ и DATA переменные A, B, K, M получают следующие значения: A=5.3; B=14.7; K=1.44; M=.508 (отметим, что порядок выборки констант из блока данных строго соответствует порядку следования имён переменных в операторе READ).

После действия оператора READ соответствующие значения удаляются из блока данных. Очевидно, что подобная форма ввода данных более экономична, чем с помощью оператора присваивания и, вдобавок, она позволяет уменьшить затраты труда программиста при изменении входных данных.

Действительно, при изменении значений переменных A, B, K, M достаточно переписать только одну программную строку с номером 10.

Пример 2.

[0245-02.bas](#)

 [0245-02.bas](#)

```
NEW
Ok
10 DATA "ДИСПЛЕЙ", "DISKETTA":READ X$,Y$
20 ?X$Y$ ' знак конкатенации можно опускать!
run
ДИСПЛЕЙDISKETTA
Ok
```

Ясно, что в результате выполнения оператора READ строковые переменные X\$,Y\$ получили значения:

```
X$="ДИСПЛЕЙ" и Y$="DISKETTA".
```

Пример 3.

[0245-03.bas](#)

 [0245-03.bas](#)

```
NEW
Ok
100 DATA 34
110 READ A,B,C$,D$,E
120 DATA 234e-7,goldfish
```

```

130 DATA "testing,1..2..3!"
140 PRINT A,B,C$,D$,E
1000 DATA 22
run
 34          2.34E-05
goldfish    testing,1..2..3!
 22
Ok

```

Общий вид оператора RESTORE:

```
RESTORE [n]
```

где:

- RESTORE («to restore» — «восстанавливать») — служебное слово;
- n — номер программной строки,  $n \in [0, 65529]$ .

Этот оператор изменяет состояние указателя считывания. Напомним, что оператор READ перемещает указатель «вниз» по мере ввода данных, пока не будет исчерпано «содержимое» всех операторов DATA.

Оператор RESTORE без указания параметра n возвращает указатель в начало блока данных, после чего все данные могут быть снова прочитаны оператором READ.

Кроме того, в операторе RESTORE может быть задан параметр n, в этом случае указатель считывания указывает на первую константу в операторе DATA в строке с номером n.

Оператор RESTORE может быть использован неоднократно и в любом месте Вашей программы.

*Примеры:*

- 4) [0245-04.bas](#)

 0245-04.bas

```

NEW
Ok
10 DATA 10,11
20 DATA YAMAHA,12,13:DATA "MSX",14
40 READ X,Y,Z$
50 PRINT X;Y;Z$:RESTORE 20
70 READ L$,A,B,M$:PRINT L$;A;B;M$
run
·10··11·YAMAHA
YAMAHA·12··13·MSX
Ok

```

- 5) [0245-05.bas](#)

 0245-05.bas

```

NEW
Ok
10 INPUT "Номер месяца (1-12)";N
20 IF N<1 OR N>12 THEN 10
30 RESTORE 100
40 FOR I=1 TO N
50 READ M$
60 NEXT
70 PRINT M$
80 GOTO 10
100 DATA Январь,Февраль,Март,Апрель,Май,Июнь,Июль,Август,Сентябрь,Октябрь,Ноябрь,Декабрь

```

## II.4.6. Оператор вывода PRINT в общем случае

Формат этого оператора:

```
[L]PRINT[# n,][USING формат;] список вывода
```

(«Line PRINT USING» — «форматная печать строки»), где:

- префикс L задаёт вывод информации на печатающее устройство (принтер); отсутствие данного префикса задаёт вывод данных на экран дисплея;
- *формат* — строковое выражение, содержащее информацию о форме представления выводимых данных;
- *список вывода* — список выражений любого типа, разделённых символами-разделителями: запятой или точкой с запятой; *список* может быть пуст;
- n — номер любого открытого файла в диапазоне от 0 до 15 (см. [раздел V.8.](#)); если в операторе присутствует префикс L, то задание параметра [# n,] приводит к синтаксической ошибке; (применение оператора PRINT с параметром [# n,] см. в [разделе V.8.](#))).

Кроме того, служебное слово PRINT для сокращения записи может быть заменено знаком «?» (для оператора LPRINT это недопустимо!).

MSX BASIC допускает автоматическое форматирование (см. [раздел II.4.4.](#)); кроме того, используя служебное слово USING («using» — «применяемый, используемый»), можно осуществлять форматирование по *усмотрению пользователя*.

Далее, символы-разделители «,» и «;» в *списке вывода* никоим образом не меняют положения курсора. Несколько значений, выводимых по одному *формату*, располагаются друг за другом без дополнительных пробелов. Некоторые символы *формата* являются управляющими и специальным образом влияют на формат вывода. Другие символы *формата* «нейтральны» и просто переносятся в выводимые значения.

Ниже перечислены основные *управляющие символы*, и даны поясняющие примеры.

1. ! — вставка первого символа значения строкового выражения из *списка вывода* вместо символа «!» в значении *формата*.

- 1) [0246-11.bas](#)

 0246-11.bas

```
NEW
Ok
10 X$="банан":Y$="ананас"
20 PRINTUSING "банан+ананас=!нанас";x$,y$
гип
банан+ананас=бананас
Ok
```

- 2) [0246-12.bas](#)

 0246-12.bas

```
NEW
Ok
10 X$="шорох"
20 PRINTUSING "!аба!";X$,X$
гип
шабаш
Ok
```

- 3) [0246-13.bas](#)

 0246-13.bas

```
NEW
Ok
10 X$="шах":y$="кол":z$="!"
20 ?USING"!аба!...? !аба!...!";Y$,Y$;X$;X$,Z$
гип
кабак...? шабаш...!
Ok
```

2. вставка первых (n+2)-х символов значения строкового выражения из *списка вывода* вместо символа «\ \» в значении *формата*.

```
\ ..... \
  ↑
```

п пробелов

- 1) 0246-21.bas

 0246-21.bas

```
NEW
Ok
10 X$="Рокер":Y$="кокон"
20 ?USING"\\ \ ";X$,Y$
run
Рококо
Ok
```

- 2) 0246-22.bas

 0246-22.bas

```
NEW
Ok
10 X$="баран":Y$="коран":Z$="метан"
20 ?USING"\\н\\ \ ";X$,Y$,Z$
run
банкомет
Ok
```

- 3) 0246-23.bas

 0246-23.bas

```
NEW
Ok
10 X$="Банан":Y$="зайка":Z$="!"
20 ?USING" \ \ ! ";X$,Y$,Z$
run
Банзай!
Ok
```

- 4) 0246-24.bas

 0246-24.bas

```
1 A$="комар":B$="герка":C$="\ \пью\ \ ":PRINTUSING C$,A$,B$
компьютер
Ok
```

3. & — вставка значения строкового выражения из списка вывода вместо символа «&» в значении формата.

- 1) 0246-31.bas

 0246-31.bas

```
NEW
Ok
10 Y$="BASIC":A$="MSX-&"
20 PRINTUSING A$,Y$
run
MSX-BASIC
Ok
```

- 2) 0246-32.bas

 0246-32.bas

```
NEW
Ok
10 Y$="BASIC ":Z$="КОМПЬЮТЕР":T$="MSX-&"
20 PRINTUSING T$,Y$,Z$
run
MSX-BASIC MSX-КОМПЬЮТЕР
Ok
```

- 3) 0246-33.bas

 0246-33.bas

```
NEW
Ok
10 X$="MSU":Y$="BASIC"
20 ?USING "\\X-&";X$,Y$
run
MSX-BASIC
Ok
```

- 4) 0246-34.bas

#### 0246-34.bas

```
NEW
Ok
10 X$="MSX":Y$="BASIC":Z$="не структурен"
20 ?USING"&-& &";X$,Y$,Z$
MSX-BASIC не структурен
run
Ok
```

4. ### — примеры значений *формата* для вывода десятичных чисел ###.# с фиксированной точкой. Символ «#» указывает на тот факт, что в данном месте поля вывода должна находиться *цифра*, а символ «.» указывает на то, что в данном месте поля вывода должна находиться десятичная точка. Соответствующий элемент данных из *списка вывода* помещается в поле (в случае отсутствия десятичная точка предполагается находящейся справа); отсутствующие цифры слева от десятичной точки дополняются пробелами; неиспользованные позиции справа от десятичной точки заполняются нулями.

Напомним, что максимальное количество значащих цифр в числе — 14 и что числа при необходимости округляются.

- 1) [0246-41.bas](#)

#### 0246-41.bas

```
Ok
1 ?USING"##.# " ;25.1, 7, 69.88, -34.7, -65, 3.7, 456.89, 4.5E-12, 1E-3, 3.64E4, 5E26

25.1.....7.0.....69.9.....%-34.7.....%-65.0.....
.3.7.....%456.9.....0.0.....0.0.....%36400.0.
..% 5E+26
Ok
```

- 2) [0246-42.bas](#)

#### 0246-42.bas

```
NEW
Ok
10 PRINTUSING"###.##," ;1.234;-4578;1234567.89;3.45E-23
run
1.23,%-4578.00,%1234567.89.0.00
Ok
```

Отметим, что если выводимое на экран дисплея или принтер число содержит десятичных позиций больше, чем «размеры» поля вывода, то число выводится, однако перед ним ставится символ процента «%», чтобы предупредить пользователя.

- 3) [0246-43.bas](#)

#### 0246-43.bas

```
NEW
Ok
10 ?USING"#.#####";123.4;12345678910111213
run
%123.400000000000000000 % 1.2345678910111E+16
Ok
```

5. +###.# — примеры значений *формата* для вывода десятичных чисел ##.##+ с фиксированной точкой с указанием знака числа («+» или «-») в начале или конце записи. Посмотрите, пожалуйста, следующий иллюстративный пример:

- 1) [0246-51.bas](#)

#### 0246-51.bas

```
10 A$="+###.# " :B$="#.###+ " 'LEN(A$)=8;LEN(B$)=9
20 ?USING A$; 12.6, 1.26, -35.7, -0.678 'пробелы - для
30 ?USING B$; 12.6, 1.26, -35.7, -0.678 'наглядности!
run
+12.6.....+1.3.....-35.7.....-0.7
%12.600+...1.260+...%35.700-...0.678-
Ok
```

6. ##.- — пример значения *формата* для вывода десятичных чисел с указанием знака «-» в конце выводимых отрицательных значений (для положительных чисел в этом случае печатается пробел в конце числа).

- 1) [0246-61.bas](#)

#### 0246-61.bas

```
NEW
Ok
10 A$="##.##- " 'LEN(A$)=9
```

```
20 ?USING A$;-11.78, 11.78;-3.6, 3.6;-234.567, 234.567
run
11.78····11.78····3.60····3.60····%234.57····%234.57
Ok
```

Отметим, что знак «-» используется только в качестве последнего символа *формата*. Использование знака «-» в качестве первого символа *формата* приводит к выводу знака «-» впереди числа, например:

- 2) 0246-62.bas

 0246-62.bas

```
Ok
1 ? USING"-##.#" ; -3.1, -33.1; 3.1, 33.1
--3.1····%-33.1····.3.1····33.1
Ok
```

7. **\*\*\*.##** — пример значения *формата* для вывода десятичных чисел с фиксированной точкой с заполнением пустого пространства перед ними символами «\*» (разумеется, если количество цифр числа меньше размера поля вывода!)

- 1) 0246-71.bas

 0246-71.bas

```
NEW
Ok
10 ?USING"***##.#" ; 125.3, -125.3; 5., -5.; 345.45, -345.45; 34.67E3, -34.67E3
run
**125.3···*-125.3···***5.0····*-5.0····**345.5···*-345.5···34670.0
···%-34670.0
Ok
```

8. **###^ ^ ^** — пример значения *формата* для вывода десятичных чисел с плавающей точкой.

- 1) 0246-81.bas

 0246-81.bas

```
NEW
Ok
10 ?USING"+##.##^ ^ ^" ; 22.5, -22.5; 234567, -234567; 2.2E-62
run
+22.5E+00···-22.5E+00···+23.5E+04···-23.5E+04···+22.0E-63
Ok
```

9. **###.###** — пример значения *формата* для вывода десятичных чисел в фиксированной форме с разбивкой записи их целой части запятыми на группы по 3 цифры в каждой, причём запятую можно ставить не только после первого, но и после любого другого символа «#», но обязательно до десятичной точки.

- 1) 0246-91.bas

 0246-91.bas

```
NEW
Ok
10 A$="# ,#####.###"
20 ?USING A$;5554322.43
run
5,554,322.43
Ok
```

- 2) 0246-92.bas

 0246-92.bas

```
NEW
Ok
10 A$="# , # , # , # , #####.###"
20 ?USING A$;12345678910.891
run
· · 12,345,678,910.891
Ok
```

10. **\$** — эти два символа, расположенные слева от первого символа «#» в *формате*, вызывают печать одного символа «\$» слева от старшей значащей цифры выводимого числового значения.

11. **\*\*\$** — эти символы вызывают печать одного символа «\$» слева от старшей значащей цифры выводимого числового значения, но в оставшихся позициях *формата* слева от символа «\$» печатаются символы « \* ».

*Пример.*

- 1) [0246-101.bas](#)  
 [0246-101.bas](#)

```
NEW
Ok
10 ?USING "$###.# ";15.3,-15.3;15390
20 ?USING"***###.# ";15.3,-15.3;15390
run
$15.3··-$15.3··%$15390.0
**$15.3··*-$15.3··%$15390.0
Ok
```

Если оператор PRINT USING содержит выражений в *списке вывода* больше, чем управляющих символов в *формате*, то после вывода значения, соответствующего последнему управляющему символу в значении *формата*, формат начинает использоваться повторно с самого начала для следующих выражений из *списка вывода*.

*Пример.*

- 1) [0246-01.bas](#)  
 [0246-01.bas](#)

```
NEW
Ok
10 X$="версия":Y$="MSX - мощная":Z$="!"
20 ?USING "& & ";X$,Y$,X$,Z$ '4 выражения и 2 управляющих символа
run
версия MSX - мощная версия !
Ok
```

Если в значении *формата* определено больше управляющих символов, чем выражений в *списке вывода*, то лишние управляющие символы игнорируются.

*Пример.*

- 2) [0246-02.bas](#)  
 [0246-02.bas](#)

```
NEW
Ok
10 X$="MSU":Y$="BASIC" ' В PRINTUSING:1 выражение
20 ?USING "\\X-&";X$ ' и 2 управляющих символа
run
MSX-
Ok
```

Разумеется, что числовым выражениям в *списке вывода* должны соответствовать числовые *форматы*, а строковым — строковые. В противном случае произойдёт ошибка.

*Пример.*

- 3.1) [0246-031.bas](#)  
 [0246-031.bas](#)

```
NEW
Ok
10 Y$="BASIC"
20 ?USING "##.#";Y$
run
Type mismatch in 20
Ok
```

- 3.2) [0246-032.bas](#)  
 [0246-032.bas](#)

```
NEW
Ok
```

```
10 A$="&&&"
20 ?USING A$;23.1
run
Type mismatch in 20
Ok
```

*Замечание.* Напомним, что при необходимости числа перед выводом округляются и что если число «шире» поля вывода, то оно все равно выводится, но перед ним ставится знак предостережения «%» .

Обычно *формат* служит для вывода сразу нескольких значений.

*Пример.*

- 4) 0246-04.bas

 0246-04.bas

```
NEW
Ok
10 A$="Факультет:\ \.Группа:#!"
20 ?USING A$;"физ","мат","М",5,"А"
run
Факультет:физ-мат.Группа:M5A
Ok
```

Отметим, что оператор вида:

```
LPRINT [USING формат;] список вывода
```

позволяет вывести информацию, получаемую при работе программы, не на экран, а на *печатающее устройство (принтер)*. Управление выводом здесь такое же, как и в случае оператора PRINT.

При выполнении оператора LPRINT на рулоне бумажной ленты(или одиночных листах) печатаются значения выражений элементов списка. При этом предполагается, что по ширине лента условно разбита на зоны по 14 позиций — в каждой (максимальное число зон в строке печати — 9).

## II.4.7. Оператор ввода данных INPUT

Разговор — это здание, которое строят совместными усилиями.

—А.Моруа

Вначале договоримся о терминологии.

*Пользователь* («user») — человек, использующий техническое и программное обеспечение, а также различные виды услуг, предоставляемые ему вычислительным центром, для решения своих задач и самостоятельно осуществляющий подготовку этих задач к решению на ЭВМ.

Режим *пакетной* обработки — один из видов организации вычислительного процесса на ЭВМ, при котором несколько задач пользователей объединяют вместе, образуя входной *пакет*, который затем последовательно обрабатывается компьютером. При пакетной обработке пользователь не имеет прямого доступа к машине, а сдаёт пакет своей задачи оператору, который вводит его в ЭВМ и выдаёт пользователю результат работы.

Режим *диалога* (*интерактивный режим*) — режим общения пользователя с компьютером, при котором пользователю обеспечивается возможность выдавать ЭВМ задание, следить за процессом обработки, получать ответы, фиксировать те или иные результаты, исправлять ошибки, выдавать указания и т.д. Режим диалога пользователь осуществляет с помощью индивидуального *терминала* (пишущей машинки, дисплея) и применяет обычно для таких задач, алгоритмы решения которых ещё не совсем определены.

**MSX BASIC** имеет в своём арсенале простые и эффективные средства организации диалога пользователя с выполняемой программой, в частности, с помощью оператора INPUT.

Вспомним, что слово «диалог» (от греч. «dialogos») означает разговор между двумя или несколькими лицами. В настоящее время участником такого «разговора» становится компьютер.

Однако отметим, что инициатива в диалоге и ответственность за успех его выполнения лежат только на пользователе!

Ясно, что разрабатываемые программы должны быть универсальными для своего класса задач, т.е. программа должна решать задачу при любых (в заданном диапазоне, конечно) значениях исходных данных. Однако, именно необходимость изменять исходные данные и не позволяет создать полностью универсальную программу при ориентации на пакетный режим работы с ЭВМ. Действительно, если обрабатываемая часть программ, работающая лишь с именами переменных, может оставаться в памяти ЭВМ неизменной, то её фрагмент, обеспечивающий ввод данных, должен переписываться при каждом их изменении.

Диалоговые средства **MSX BASIC** (диалоговый режим) позволяют пользователю вводить данные с клавиатуры в нужный момент по запросу самой программы, в определённом месте которой записан оператор вида:

```
INPUT ["β";] α, σ, δ, ...
```

где:

- INPUT («input» — «ввод») — служебное слово;
- β — последовательность произвольных символов алфавита **MSX BASIC** («подсказка»);
- α, σ, δ, ... — список имён переменных или имён элементов массивов, разделённых запятыми.

Напомним, что квадратные скобки означают, что информация внутри них не является обязательной, то есть в данном случае допустима запись оператора INPUT в виде: INPUT α,σ,δ, ...

При выполнении оператора INPUT вычисления приостанавливаются и на экране дисплея выводится «подсказка» β (если она имеется), знак «?», пробел и текстовый курсор, то есть

```
β ? █
```

«Подсказка», то есть текст, появляющийся на экране дисплея перед вводом необходимой для дальнейшей работы информации, — один из элементов культуры программирования. Благодаря такой особенности программа не нуждается в дополнительной инструкции. Она сама подсказывает, как с ней работать.

Далее компьютер *ожидает*, пока с клавиатуры не будут введены значения для всех переменных списка α, σ, δ, ... . Вводимые данные представляют собой числовые или строковые константы. При наборе они отделяются друг от друга запятыми. Тип (числовой, строковый) и количество их должны соответствовать типам и количеству переменных α, σ, δ, ... . После нажатия на клавишу **Ввод** значения вводимых констант последовательно присваиваются этим переменным.

Впрочем, вводить данные можно и частями.

Пример 1.

[0247-01.bas](#)

 [0247-01.bas](#)

```
NEW
Ok
10 INPUT "ГРУППА"; A$, B, C$
20 PRINT A$; B; C$: END
run
ГРУППА? M   Нажмите клавишу "RETURN" !
?? 5,A     Нажмите клавишу "RETURN" !
M 5 A
Ok
```

После ввода компьютер продолжает вычисления.

Если на строке после знака «?» были только пробелы, то считается, что Вы не хотите изменять значения переменных  $\alpha$ ,  $\beta$ ,  $\gamma$ , ... .

Пример 2.

[0247-02.bas](#)

 [0247-02.bas](#)

```
NEW
Ok
10 A=5:INPUT A:PRINT A
20 END
run
? Нажмите клавишу "RETURN" !
5 Препренее значение переменной A сохранилось!
Ok
```

При несоответствии типов вводимых констант типам переменных  $\alpha$ ,  $\sigma$ ,  $\delta$ , ... на следующей экранной строке появляется сообщение об ошибке:

? Redo from start

(«Повторите сначала»),

и в следующей строке вновь выводится «подсказка», символ «?», пробел и текстовый курсор.

Разумеется, у Вас должно быть достаточно информации для обеспечения ввода значения каждой переменной из списка ввода. Если значений переменных недостаточно, то **MSX BASIC** выдаёт символы «??» в следующей строке и переходит в режим ожидания, пока Вы не введёте дополнительных данных. Если значений слишком много, то в следующей строке выдаётся сообщение:

? Extra ignored

(«Лишнее игнорируется»).

Пример 3.

[0247-03.bas](#)

 [0247-03.bas](#)

```
NEW
Ok
10 INPUT "ГРУППА";A$,B,C$
20 PRINT A$;B;C$
30 END
run
ГРУППА? M-
?? L
? Redo from start
ГРУППА? M-,5,-A,-D
? Extra ignored
M- 5 -A
Ok
```

Вводимые значения строковых переменных можно заключать в кавычки; если кавычка является первым символом вводимого значения, то значением переменной становится все содержимое кавычек; если кавычка — не первый встречающийся символ, то она является одним из символов вводимого значения.

Пример 4.

[0247-04.bas](#)

 [0247-04.bas](#)

```
NEW
Ok
```

```
10 INPUT "Гм...";A,B$:PRINT A,B$
```

```
run
Гм...? 123.4
?? ab"g
·123.4········ab"g
Ok
```

```
run
Гм...? e5,"ab"g
? Redo from start
Гм...? e5,"ab'g"
·0··········ab'g
Ok
```

Оператор INPUT переводит программу в режим SCREEN 0 или SCREEN 1, поэтому использование его невозможно, если Вы работаете в графических экранах SCREEN 2 и SCREEN 3 (см. [раздел V.1.](#)).

## II.4.8. Оператор LINEINPUT

Общий вид данного оператора:

```
LINEINPUT ["β";] строковая переменная ,
```

где:

- LINEINPUT («ввод строк») — служебное слово;
- β — последовательность произвольных символов алфавита MSX BASIC («подсказка»);
- *строковая переменная* — имя строковой переменной или имя элемента строкового массива.

Этот оператор обеспечивает чтение символов с клавиатуры. Вначале на экран выводится «подсказка» β, и текстовый курсор, расположенный на следующей за ней позиции дисплейной строки.

Теперь Вы можете вводить любую информацию, даже можете нажимать командные клавиши, комбинацию символов (используя способы редактирования с клавиатуры).

После нажатия клавиши **Ввод** первые 255 символов, начиная с начальной позиции курсора и до конца строки (строка может занимать несколько физических строк), становятся новым значением *строковой переменной*.

Указанный оператор, как правило, используется для ввода в компьютер значений строковых переменных, содержащих кавычки или запятые.

Дело в том, что строку символов, содержащую кавычки, например, строку

```
Курс "Программирование"
```

нельзя ввести как значение строковой переменной в память компьютера оператором INPUT.

*Примеры:*

- 1) [0248-01.bas](#)  
 [0248-01.bas](#)

```
NEW
Ok
10 LINEINPUT"!";A$:PRINT A$
run
!синус
синус
Ok
```

Вы можете при помощи оператора LINEINPUT вводить числа, но после чтения результат нужно преобразовать при помощи функции VAL() (см. [раздел IV.1.3.](#)).

Примеры:

- 2) [0248-02.bas](#)

 [0248-02.bas](#)

```
NEW
OK
10 LINEINPUT "Введите число: ";A$
20 Z=VAL(A$):PRINT Z^2
гип
Введите число:12
144
Ok

гип
Введите число:4E-4
1.6E-07
Ok

гип
Введите число:&B100
16
Ok
```

- 3) [0248-03.bas](#)

 [0248-03.bas](#)

```
NEW
OK
10 LINEINPUT"Введите текст: ";T1$
15 INPUT"Введите этот же текст?";T2$
20 PRINT"Первый раз текст воспринят как: "; :PRINT T1$
30 PRINT"Во второй раз текст воспринят как: "; :PRINT T2$
```

1. гип  
Введите текст:кука  
Введите этот же текст? кука  
Первый раз текст воспринят как:кука  
Во второй раз текст воспринят как:кука  
Ok

2. гип  
Введите текст:ку,ка  
Введите этот же текст? ку,ка  
?Ext ga ignoged  
Первый раз текст воспринят как:ку,ка  
Во второй раз текст воспринят как:ку  
Ok

3. гип  
Введите текст:"ку"ка"  
Введите этот же текст? "ку"ка"  
?Redo from start  
Введите этот же текст? (нажмите клавишу "RETURN")  
Первый раз текст воспринят как:"ку"ка"  
Во второй раз текст воспринят как:ку  
Ok

Из примера 3.2 хорошо видно, что знак «,» в операторе INPUT является разделителем информации, в то время как в операторе LINEINPUT запятая воспринята как символ строки.

Пример 3.3 наглядно показывает, что в операторе INPUT символ «кавычка» не воспринимается как символ — он является указателем границ строковой константы.

Заметим, что нажатие **CTRL+C** или **CTRL+STOP** вызовет прекращение действия оператора LINEINPUT, перевод интерпретатора в режим прямого выполнения команд и выдачу на экране дисплея сообщения:

«Break in ...»  
(«Прерывание в строке ...»).

Оператор LINEINPUT, также как и оператор INPUT, переводит программу в режим SCREEN 0 или режим SCREEN 1.

Отметим, что в школьном алгоритмическом языке ввод информации исполнителю осуществляется с помощью команды **ввод**, действие которой идентично действию оператора INPUT без строки — «подсказки» β. «Подсказкой» в алгоритмическом языке может служить команда **вывод**, где выводимой информацией является текст — «подсказка».

Ясно, что оператор INPUT вскоре отойдёт в прошлое, а пользователь, возможно, будет обращаться к компьютеру с помощью жеста или мимики. Однако пока компьютер не способен воспринимать такие обращения, но **MSX BASIC** уже сейчас представляет пользователю достаточно развитые устройства ввода, например, световое перо или джойстик.

## II.4.9. Операторы END и STOP. Команда CONT

Операторов END в одной программе может быть несколько, причём располагаться они могут в любых её местах. При выполнении END вычисления по программе прекращаются.

Оператор END не является обязательным для программы. При его отсутствии вычисления прекращаются после выполнения последнего оператора программной строки с наибольшим номером (конечно, если этот оператор не предусматривает передачу управления!). Однако, END удобен для документации, так как он обозначает физический конец программы.

Отметим, что имеется второй оператор останова программы

STOP

(«to stop» — «остановиться»).

Разница между END и STOP только в том, что STOP приостанавливает работу по программе, после действия этого оператора можно при желании продолжить работу командой

CONT

(«to CONTinue» — «продолжать») в непосредственном режиме; END же прекращает работу окончательно.

## Диск с примерами

[Загрузить образ диска](#)

 [Открыть диск в WebMSX](#)

[http://sysadminmosaic.ru/msx/basic\\_programming\\_guide/02?rev=1583570469](http://sysadminmosaic.ru/msx/basic_programming_guide/02?rev=1583570469)

2020-03-07 11:41

