

Additional Calculation Topics

10

Before discussing additional TUTOR calculational capabilities, let's review briefly those aspects which have been covered so far:

- 1) Expressions follow the rules of high school algebra. Multiplication takes precedence over division, which takes precedence over addition and subtraction. Superscripts may be used to raise numbers to powers. The symbol π may be used to mean 3.14159. . . . The degree sign ($^\circ$) may be used to convert between degrees and radians.
- 2) There are 150 student variables, v1 through v150, which may be named with the `-define-` command. These variables can be set or altered by assignment (\Leftarrow) and by `-store-`, `-storen-`, or `-storea-` commands. If a "define student" set of definitions is provided, the student may use variable names in his or her responses.
- 3) Logical expressions are composed using the operators `=`, `≠`, `>`, `<`, `≥`, `≤`, `and`, `or`, and the "not" function. Logical expressions have the value true (-1) or false (0).
- 4) There are several available system variables such as "where", "wherey", "anscnt", "jcount", "spell", etc. Available system functions include `sin(x)`, `sqrt(x)`, etc. A full list of system variables and functions is given in Appendix C.
- 5) The `-show-` command (and its relatives `-showt-`, `-showz-`, `-showe-`, and `-showo-`) will display the numerical value of an

expression. The `-showa-` command will display stored alphanumeric information. These commands may be embedded within `-write-` and `-writec-` statements.

- 6) The `-calcc-` and `-calcs-` commands make it easy to perform (conditionally) one of a list of calculations or assignments.
- 7) The `-randu-` command with one argument picks a fraction between 0 and 1. With two arguments, it picks an integer between 1 and the limit specified. There is a set of commands associated with permutations: `-setperm-`, `-randp-`, `-remove-`, and `-modperm-`.
- 8) The iterative form of the `-do-` command facilitates repetitive operations.

Now let's look at additional TUTOR calculational capabilities.

Defining Your Own Functions

While many important functions such as $\ln(x)$ and $\log(x)$ are built-in to the TUTOR language, it is frequently convenient to define your own functions. To take a simple example, suppose you define a cotangent function:

```
define cotan(a)=cos(a)/sin(a)
```

Then, later in your lesson you can write:

```
calc r←cotan(3x+y-5)
```

and TUTOR will treat this as though you had written:

```
calc r←[cos(3x+y-5)/sin(3x+y-5)]
```

Such use of functions not only saves typing but improves readability.

CAUTION: In defining a function, the *arguments* must not be already defined. For example, the following definition will be rejected by TUTOR (with a suitable error message):

```
define x=v1
      cube(x)=x3
```

This must be rewritten as:

```
define x=v1
      cube(dummy)=dummy3
```

or anything similar. A function definition may involve previously defined quantities on the *right* side of the “=” sign, however. You might have:

```
define x=v1
      new(c)=c4+2x
```

In this case you might have a -calc- that looks like:

```
calc x←15.7
     y←3new(8)
```

and this would be equivalent to:

```
calc x←15.7
     y←3[(8)4+2x]
```

Sometimes it is convenient to define “functions” that have *no* arguments:

```
define r=v1
      quad=r2-100
      r3=r1/3
      root=sqrt(r)
      prod=r3×root
      trans=(r←prod)
```

Note that “prod” depends on two previous definitions, each of which (in turn) depends on the definition of “r”. There is no limit on how deep you can go in definition levels. The unusual definition of “trans” permits you to write an unusual -calc- (where the assignment is implicit in the definition of “trans”):

```
calc trans
```

Essentially anything is a legal definition. The only rule is that the definition make sense when enclosed in parentheses (since a defined name when encountered in an expression is replaced by its meaning and surrounded by parentheses). This means that you cannot define “minus=-” because (-), a minus sign enclosed in parentheses, is not permitted in an expression. On the other hand, “minus=-1” is all right because (-1) is meaningful.

A function may have up to six arguments. Here is a function of two arguments:

```
define modulo(N,base)=N-[base×int(N/base)]
```

This means that modulo (17,5) in an expression will have the value 2; the “int” or “integral part” function throws away the fractional part of 17/5, leaving 3, so that we have $(17-5 \times 3) = (17-15) = 2$. This modulo function, therefore, gives you what is left over in division of “N” by “base”.

Here are a couple of other examples of multi-argument function definitions:

```
define big(a,b)=-[a×(a≥b)+b×(b>a)]
      small(a,b)=-[a×(a≤b)+b×(b<a)]
```

The minus sign appears because logical true is represented by -1. If you have “big(x+y,z)” in an expression, with $(x+y)=7$ and $z=3$, this expands to:

$$-[7 \times (7 \geq 3) + 3 \times (3 > 7)]$$

which reduces to $-[7 \times (-1) + 3 \times (0)]$ which is 7. So our “big” function picks out the larger of two arguments.

Arrays

It is often important to be able to deal with arrays of data such as a list of exam scores, the number of Americans in each 5-year age group together with their corresponding mortality and fertility rates, a list of which pieces are where on a chess board, or the present positions of each of several molecules in the simulation of the motion of a gas.

Suppose we have somehow entered the exam scores for twenty students into variables v31, v32, v33 . . . up to v50. Here is a unit which will let you see the score of the 5th or 13th or Nth student:

```
unit      see
back     index
at       1215
write    Which student number?
         (Press BACK when done.)
arrow    1518
store    N
wrongv   10.5,9.5 $$ range 1 to 20
write    The score of the <s,N>th student is <s,v(30+N)>.
```



(The -wrongv- rather than -ansv- makes it easy to ask another question.)
The new element here is the “indexed variable”:

$$v(30+N)$$

which means “evaluate $30+N$, round to the nearest integer, and choose the corresponding variable”. For example, if N is 9, $v(30+N)$ is $v(39)$ or $v39$. If N is 13.7, $v(30+N)$ means $v44$.

We might list and total all the scores:

```

.
.
calc  total←0          $$ initialization step
do    showem,N←1,20
at    3035
write The average score is <s,total/20>.
*
unit  showem
at    835+100N
show  v(30+N)
calc  total←total+v(30+N)

```

As usual, it is preferable to define a name for this data, such as:

```
define scores(i)=v(30+i)
```

in which case we would write our last unit as:

```

unit  showem
at    835+100N
show  scores(N)
calc  total←total+scores(N)

```

Due to the special meaning attached to “ $v(\text{expression})$ ” you must exercise some care in using a variable named “ v ”, in that you must write “ $v \times (a+3b)$ ” and not “ $v(a+3b)$ ” if you mean multiplication. We will see later that the same restriction applies to the names “ n ”, “ vc ”, and “ nc ”. This restriction does not apply to students entering algebraic responses, where “ $v(a+3b)$ ” is taken to mean “ $v \times (a+3b)$ ”. Students can use indexed variables only if they are named (as in “scores” in the above example). Such definitions must, of course, be in the “define student” set.

Suppose you have three sets of exam scores for the twenty students. This might conveniently be thought of as a 3×20 (“two-dimensional”)

array. Suppose we put the first twenty scores in v31 through v50, the second set in v51 through v70, and the third set in v71 through v90. It might be convenient to redefine your array in the following manner:

```
define scores(a,b)=v(10+20a+b)
```

Then, if you want the 2nd test score for the 13th student, you just refer to scores (2,13) which is equivalent to $v(10+40+13)$ or $v(63)$. If you wanted to display all the scores you might use “nested” -do- statements:

```
.
.
do    column,i<=1,3
*
unit  column
do    rows,j<=1,20
*
unit  rows
at    820+10i+100j
show  scores(i,j)
```

Unit “column” is done three times and for each of these iterations, unit “rows” is performed twenty times.

There is an alternative way to define our array:

```
define i=v1,j=v2
       scores=v(10+20i+j)
```

Then our unit “rows” would look like:

```
unit  rows
at    820+10i+100j
show  scores
```

The indices specifying which test is for which student are implicit. This form is particularly useful when you have large subroutines where “i” and “j” are fixed and it would be tiresome to type over and over again “scores(i,j)”. Just set “i” and “j”, then -do- the subroutine.

It is frequently necessary to initialize an entire array to zero. One way to do this is with -do- statements:

```
unit  clear
do    clear2,i<=1,3
*
```

```

unit clear2
do clear3,j<=1,20
*
unit clear3
calc scores(i,j)<=0

```

A simpler way to accomplish the same task is to say:

```
zero scores(1,1),60
```

You simply give the starting location (the first of the 60 variables) and the number of variables to be cleared to zero. As another example, you can clear all of your variables by saying:

```
zero v1,150
```

Not only is the -zero- command simpler to use, but TUTOR can carry out the operation several hundred times faster! TUTOR keeps a block of its own variables, each of which always contains zero. When you ask for 150 variables to be cleared, TUTOR does a rapid block transfer of 150 of its zeroed variables into your specified area. This ultra-high-speed block transfer capability can be used in other ways. For example:

```
transfr v10;v85;25
```

performs a block transfer of the 25 variables starting with v10 to the 25 variables starting with v85. In this way you can move an entire array from one place to another with one -transfr- command, and at speeds hundreds of times faster than are possible by other means.

Segmented Variables

Storing three scores for each of your twenty students required the use of 60 variables, out of an available 150. We're running out of room! You can save space by defining "segmented" variables which make it easy to keep several numbers in each student variable. For example, you can write a definition of the form:

```
define segment,score=v31,7
```

This identifies "score" as an array which starts at v31 and consists of segments holding positive integers (whole numbers) smaller than 2⁷ (which is 128). It turns out that each student variable will hold 8 such

segments, so “score(8)” is the last segment in v31, while “score(9)” is the first segment in v32. Since “score(60)” is the fourth segment in v38, we need only eight variables to hold all sixty scores. You can use “score(expr)” in calculations. The expression “expr” will be rounded to the nearest integer and the appropriate segment referenced. As a simple example:

```
calc score(23)←score(3)+5
```

will get the third segment, add 5 to it, and store the result in the twenty-third segment.

If we define a segmented one-dimensional array “score”, we can define a two-dimensional array as before:

```
define segment,score=v31,7
      scores(a,b)=score(20a-20+b)
```

With these definitions, “scores(1,1)” means “score (20-20+1)” or “score(1)”, which is the first segment in v31. As before, “scores” could use implicit indices:

```
define i=v1,j=v2
      scores=score(20i-20+j)
```

In this case you use “scores” rather than “scores(expr1,expr2) in calculations. NOTE: At the present writing, the commands -zero- and -transfr- cannot be used with segmented variables because these commands refer to entire variables. You could, however, zero all of the scores by saying “zero v31,8” which sets v31 through v38 to zero, which has the effect of zeroing all the segments contained in those eight variables. You can make such manipulations more readable by defining your segmented array this way:

```
define start=v31
      segment,score=start,7
```

Then you can write “zero start,8” rather than “zero v31,8”. Similar remarks apply to the -transfr- command.

It is possible to store integers (whole numbers) that can be negative as well as positive:

```
define segment,temp=v5,7,signed
```


The addition of the word “signed” (or the abbreviation “s”) permits you to hold in “temp(i)” any integer from -63 to +63. The range 2⁷ (128) has been cut essentially in half to accommodate negative as well as positive values. The following table summarizes the unsigned and signed ranges of integers permissible for various segment size specifications up to 30 (sizes up to 59 are allowed, though beyond 30 there is only one segment per variable).

| Segment size | n | 2 ⁿ | unsigned range | signed range | No. of segments per variable |
|--------------|----|----------------|--------------------|------------------------------|------------------------------|
| 1 | 1 | 2 | 0 to 1 | — | 60 |
| 2 | 2 | 4 | 0 to 3 | -1 to +1 | 30 |
| 3 | 3 | 8 | 0 to 7 | -3 to +3 | 20 |
| 4 | 4 | 16 | 0 to 15 | -7 to +7 | 15 |
| 5 | 5 | 32 | 0 to 31 | -15 to +15 | 12 |
| 6 | 6 | 64 | 0 to 63 | -31 to +31 | 10 |
| 7 | 7 | 128 | 0 to 127 | -63 to +63 | 8 |
| 8 | 8 | 256 | 0 to 255 | -127 to +127 | 7 |
| 9 | 9 | 512 | 0 to 511 | -255 to +255 | 6 |
| 10 | 10 | 1 024 | 0 to 1 023 | -511 to +511 | 6 |
| 11 | 11 | 2 048 | 0 to 2 047 | -1 023 to +1 023 | 5 |
| 12 | 12 | 4 096 | 0 to 4 095 | -2 047 to +2 047 | 5 |
| 13 | 13 | 8 192 | 0 to 8 191 | -4 095 to +4 095 | 4 |
| 14 | 14 | 16 384 | 0 to 16 383 | -8 191 to +8 191 | 4 |
| 15 | 15 | 32 768 | 0 to 32 767 | -16 383 to +16 383 | 4 |
| 16 | 16 | 65 536 | 0 to 65 535 | -32 767 to +32 767 | 3 |
| 17 | 17 | 131 072 | 0 to 131 071 | -65 535 to +65 535 | 3 |
| 18 | 18 | 262 144 | 0 to 262 143 | -131 071 to +131 071 | 3 |
| 19 | 19 | 524 288 | 0 to 524 287 | -262 143 to +262 143 | 3 |
| 20 | 20 | 1 048 576 | 0 to 1 048 575 | -524 287 to +524 287 | 3 |
| 21 | 21 | 2 097 152 | 0 to 2 097 151 | -1 048 575 to +1 048 575 | 2 |
| 22 | 22 | 4 194 304 | 0 to 4 194 303 | -2 097 151 to +2 097 151 | 2 |
| 23 | 23 | 8 388 608 | 0 to 8 388 607 | -4 194 303 to +4 194 303 | 2 |
| 24 | 24 | 16 777 216 | 0 to 16 777 215 | -8 388 607 to +8 388 607 | 2 |
| 25 | 25 | 33 554 432 | 0 to 33 554 431 | -16 777 215 to +16 777 215 | 2 |
| 26 | 26 | 67 108 864 | 0 to 67 108 863 | -33 554 431 to +33 554 431 | 2 |
| 27 | 27 | 134 217 728 | 0 to 134 217 727 | -67 108 863 to +67 108 863 | 2 |
| 28 | 28 | 268 435 456 | 0 to 268 435 455 | -134 217 727 to +134 217 727 | 2 |
| 29 | 29 | 536 870 912 | 0 to 536 870 911 | -268 435 455 to +268 435 455 | 2 |
| 30 | 30 | 1 073 741 824 | 0 to 1 073 741 823 | -536 870 911 to +536 870 911 | 2 |

Table 10-1.

As an example of the use of this table, suppose you are dealing with integers in the range from -1200 to $+1800$. You would need a segment size of 12 (signed), which gives a range from -2047 to $+2047$. There would be 5 segments in each variable. Your `-define-` might look like:

```
define segment,dates=v140,12,signed
```

It is not necessary to understand the rationale behind this table in order to be able to use segments effectively. Explanations of the underlying “binary” or “base 2” number system and the associated concept of a “bit” are discussed later in an optional section of this chapter.

Segments are frequently used to set “flags” or markers in a lesson. For example, you might like to keep track of the topics the student has completed or which questions in a drill have been attempted. A segment size of just one is sufficient for such things, with the segment first initialized to zero, then set to one when the topic or question has been covered. The definition might look like this:

```
define flags=v2
      segment,flag=flags,1
```

In the first unit, (not the “initial entry unit”) use the statement “zero flags” to clear all sixty segments in `v2`. If you use up to 120 markers you would use “zero flags,2” to clear two variables, each containing 60 segments. When the student completes the fourth topic you use “calc flag(4) \leftarrow 1” to set the fourth flag. You can retrieve this information at any time to display to the student which topics he or she has completed. Note that the `-restart-` command can be used to restart the student somewhere after the first unit (where the flags would otherwise be cleared), so that you can remind the student of which sections he or she completed during previous sessions.

Although only whole numbers can be kept in segments, it is possible to use the space-saving features of segments even when dealing with fractional numbers. Suppose you have prices of items which (in dollars and cents) involve fractions such as \$37.65 (37 dollars plus 65 hundredths of a dollar). Assume that \$50 is the highest price for an item. Simply express the prices in *cents*, with the top price then being 5000 cents. Using the table, we see that a segment size of 13 will hold positive integers up to 8191, so we say:

```
define price=v1  $$ in dollars and cents
      segment,cents=v2,13
      put(i)=[cents(i) $\leftarrow$ 100price]
      get(i)=[price $\leftarrow$ cents(i)/100]
```

A sequence using these definitions might look like:

```

calc  price←28.37
.
.
.
calc  put(16)      $$ equivalent to "cents(16)←100price"
.
.
.
show  get(16)      $$ equivalent to "price←cents(16)/100"

```

The final -show- will put "28.37" on the screen, even though between the "put" and "get", the number was the integer "2837". Notice the unusual "calc put(16)" which has an assignment (\leftarrow) implicit in the definition of "put". Also notice that the variable "price" is changed as a side-effect of "get". If this is not desired, we could define "get(i)=cents(i)/100".

As another example of the use of segments with fractional numbers, suppose you have automobile trip mileages up to 1000 miles which you want to store to the nearest tenth-mile (such as 243.8 miles). In this case you must multiply by 10 when storing into a segment and divide by 10 when retrieving the information. You would use a segment size of 14, since your biggest number is 10000. It should be pointed out that rounding to the nearest integer occurs when storing a non-integer value into a segment:

```

calc  miles←539.47
      seg(2)←10 miles  $$ 5394.7 becomes 5395
      miles←seg(2)/10  $$ 5395/10 or 539.5


```

So, by going into and out of the segment, the "539.47" has turned into "539.5".

Aside from the restriction to integers, calculations with segmented variables have one further disadvantage: they are much slower than calculations with whole variables. This is due to the extra manipulations the computer must perform in computing which variable contains the Nth segment, and extracting or inserting the appropriate segment. Segments save space at the expense of time. In many cases this does not matter, but you should avoid doing a lot of segment calculations in a heavily-computational repetitive loop, such as an iterative -do- which is done ten thousand times. (There are other kinds of segments, "vertical" segments, which are handled much faster but these have quite different space requirements than regular segmented variables.)

Branching Within a Unit: -branch- and -doto-

All of the branching or sequencing commands discussed so far referred to -unit-s (or -entry-s). It is often convenient to be able to branch *within* a unit, which is possible with the -branch- command:

| | | |
|---|--------|-----------------------|
| | unit | somethin |
|  | branch | count-4,5,x,8after |
| | at | 1215 |
| | write | "count" is equal to 4 |
| | 5 | |
| | do | countit |
| | 8after | count<=15 |

The tag of the -branch- command is like the tag of a -goto-, except that unit names are replaced by "statement labels." These labels appear at the beginning of statements and must start with a *number* (0 through 9) to distinguish them from commands, which start with letters. A statement beginning with a label need not have any tag (as in the line above labeled "5"), but it can have a tag like that of a -calc-, as in the last statement above ("8after count<=15"). In fact, a labeled statement *is* essentially a -calc- statement. As with -goto-, "x" in a -branch- means "fall through" to the next statement.

It is not permissible in a unit to label two statements with the same label (nor can you have two units with the same name in a lesson). On the other hand, since -branch- operates only *within* a unit and cannot refer to labels in other units, it is all right to use the same label in different units. (Similarly, you can use the same *unit* name in different *lessons*.) Note that -entry- is similar to -unit-, so -branch- cannot be used to branch to a label if an -entry- command intervenes.

It is often convenient to use -branch- rather than -goto-. In addition, -branch- requires less computer processing than -goto-, so that heavily computational iterations are better done with -branch- where possible. Generally speaking, about the only time you must consider the computational efficiency of one TUTOR technique compared with another is when you do a large number of iterations of some process. Unless you are making many passes through the same statements, merely write your TUTOR statements in what seems to be the simplest and most readable manner. It is a mistake to spend time worrying about questions of efficiency if the student will make only one pass through the statements.

Just as -branch- is a fast -goto- within a unit, there is a fast -doto- (analogous to the iterative -do-) for use within a unit:

```

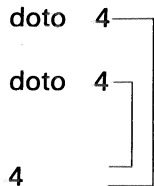
      doto 8end,i<=first,last,incr
      calc a<=b×sin(5i°)
      at 100,200+2a-i
      write T
      8end
      circle 100
    
```

The tag of the `-doto-` is similar to an iterative `-do-`, but instead of naming a unit to be done repeatedly you name a statement label. For each iteration TUTOR executes statements from the `-doto-` down to the named statement label. After the last iteration is performed, TUTOR proceeds to the statement which follows the `-doto-` label (`-circle-` in the above example).

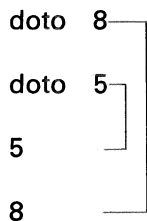
Just as it is possible to have nested `-do-` iterations, it is also possible to have nested `-doto-`s. Here is a comparison of `-do-` and `-doto-` for displaying a two-dimensional array:

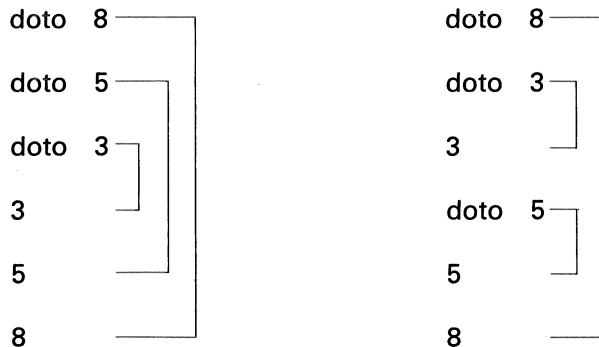
| -do- | -doto- |
|------------------|------------------|
| do column,i<=1,3 | doto 4,i<=1,3 |
| unit column | doto 4,j<=1,20 |
| do rows,j<=1,20 | at 820+10i+100j |
| unit rows | show scores(i,j) |
| at 820+10i+100j | 4 |
| show scores(i,j) | |

This nested `-doto-` example has the structure:

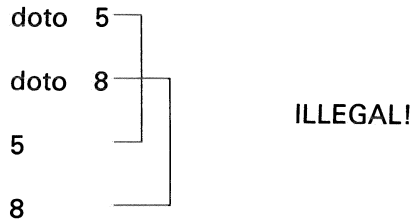


Other possible structures include the following:





Note that in each case the “inner” -doto-s are nested within the “outer” -doto-s. Here is a counter-example of a structure which is *not* permissible:



When do you use -doto- instead of an iterative -do-? Use -doto- whenever the contents of the loop are very short, because the “overhead” associated with each -doto- iteration is much less than the “overhead” associated with each -do- iteration. This is due to the extra manipulation involved in getting to the “done” unit. If the contents of the loop are long, the overhead becomes insignificant, and either -do- or -doto- can be used, whichever you prefer or whichever is more readable.

Array Operations

You have seen how to operate on individual elements of an array by using indexed variables. It is also possible to define an array in such a way as to permit operating on the array *as a whole*. Here are two sets of statements, one using true “arrays” and the other using indexed variables, with both routines calculating the sum of sixty scores (three scores for each of twenty students):

```

TRUE ARRAY
define total=v1
      array,scores(3,20)=v31
    
```

```

INDEXED VARIABLE
define total=v1,j=v2,j=v3
      scores(a,b)=v(10+20a+b)
    
```

```

calc   total←Sum(scores)           calc   total←0
                                           doto  4,i←1,3
                                           doto  4,j←1,20
                                           calc   total←total+scores(i,j)
                                           4

```

The calculation using indexed variables involves initializing “total” to zero, then using nested *-doto-s* to add in each element of “scores”. The true array calculation is much simpler, involving a single *-calc-* statement!

The statement “define array,scores(3,20)=v31” tells TUTOR to put scores (1,1) in v31, scores (1,2) in v32, scores (1,3) in v33, etc., with scores (2,1) in v51, scores (2,2) in v52, etc. Moreover, this “array” definition permits you to work with the whole array, and there are various array functions such as “Sum” to help you. The expression “Sum(scores)” means “add up all the numbers in all the elements of the array”. Similarly, the statement “scores←scores+1” will cause *all* sixty array elements to be increased by one.

Such whole-array operations are not possible with indexed variables, because (with indexed variables) TUTOR does not know how many elements make up the whole array. On the other hand, the complexities of handling true arrays limits their size to 255 elements at present and to only two “dimensions” (that is, you can’t say “define array,points(2,5,4)=v1”, which would define a three-dimensional array). So, ordinary indexed variables do have their uses, particularly when manipulating large databases (as discussed in the next chapter). While the most useful feature of true arrays is the ability to deal with all elements at once, you can also refer to individual elements, such as scores(2,15), just as you would with indexed variables.

Suppose we define two arrays, A and B, both ten variables long:

```

define array,A(10)=v141
        array,B(10)=v131

```

The following calculations involving these arrays will have the specified results:

CALCULATION

RESULT

A←2B

Each element of A is assigned the value of two times the corresponding element of B:
A(1)←2B(1), A(2)←2B(2), etc.

(Continued on next page.)

| CALCULATION (continued) | RESULT (continued) |
|----------------------------|---|
| A←25 | Each element of A is set to 25. |
| A←1/A | Each element of A is replaced by its reciprocal. |
| A←A+B | Corresponding elements of A and B are added together, and the sum replaces the element of A: A(1)←A(1)+B(1), A(2)←A(2)+B(2), etc. |
| A←3.4cos(B) | A(1)←3.4cos(B(1)), etc. |
| A←B ² | Presently not allowed: use A←BxB instead. |
| A←A \$and\$ B | Each element of A is replaced by -1 or 0, depending on a logical "and" of the corresponding elements of A and B (which should of course contain logical values, -1 and 0, to begin with). |

There are a couple of special operators unique to array manipulations: A ° B gives the standard "matrix multiplication", with row-by-column multiplication and summation, and A × B gives the standard "vector product" or "cross product". If A and B are one-dimensional arrays, the matrix multiplication A ° B yields a single number, known in mathematics as the "dot product". The symbol ° is typed by means of MICRO-x, and "x" is typed by MICRO-shift-x.

There are some useful functions:

| | |
|-----------|---|
| Sum(A) | Adds up all the elements of A |
| Prod(A) | The product of all the elements: A(1) x A(2) x A(10) |
| Min(A) | Picks out the smallest value |
| Max(A) | Picks out the largest value |
| And(A) | A(1)\$and\$A(2)\$and\$A(3) \$and\$A(10) |
| Or(A) | A(1)\$or\$A(2)\$or\$A(3) \$or\$A(10) |
| Rev(A) | Reverses the order of the elements |
| Transp(A) | Produces the transposed array: A(i,j)←A(j,i) |

Combinations of the various operations and functions can be used to your advantage. For example, a common statistical calculation involves the square root of the sum of the squares of all array elements. This can be easily obtained from sqrt(Sum(A×A)), or from sqrt(A ° A) if A is a one-dimensional array.

Arrays can be filled with a `-set-` command and displayed with a `-showt-` command:

```

define  array,C(2,3)=v16
set     C←100,200,300
        400,500,600
at      1215
showt   2C,5  $$ 5 figures

```

} will display

```

                200 400 600
                800 1000 1200

```

The `-set-` command fills elements in order. For example: `C(1,1)`, `C(1,2)`, `C(1,3)`, `C(2,1)`, `C(2,2)`, `C(2,3)`. The `-showt-` (“show tabular”) command shows the numbers appropriately on the screen. You can also use `-showe-`, `-showo-`, and `-showa-` (but not `-show-` or `-showz-` at present).

It is often convenient for the array elements to be offset, so that the first element is not numbered “one”. For example, you might want an array of the world population from 1900 to 1970. In this case, simply say “`define array,popul(1900;1970)=v1`”, which assigns `popul(1900)` to `v1` and `popul(1970)` to `v71`. Note the semicolon in the `-define-`. A two-dimensional array with offsets is written “`define array,D(-3,0;5,8)=v1`”, where `D(-3,0)` is in `v1`, `D(-3,1)` is in `v2`, etc. The last element of this array is `D(5,8)`.

Integer Variables and Bit Manipulation

This section goes much more deeply into the way a computer represents numbers and character strings. You might start off by skimming this section to see whether you will need to study it in detail. You will need this material only if you pack several pieces of data in one variable or if you want to use `-calc-` operations on character strings.

A variable such as `v150` can hold a number as big as 10^{322} (the number 1 followed by 322 zeros) or a non-zero number as small as 10^{-293} (a 1 in the 293rd position after the decimal point). These huge or tiny numbers may be positive or negative, from $\pm 10^{-293}$ up to $\pm 10^{322}$. Any number held in `v150` is recorded as sixty tiny “bits” of information. For example, whether the number is positive or negative is one bit of information, and whether the magnitude is 10^{+200} or 10^{-200} is another bit of information. The remaining 58 bits of information are used to specify precisely the number held in `v150`.

What is a bit? A bit is the smallest possible piece of information and represents a two-way (binary) choice such as yes or no, or true or false, or

up or down (anything with two possibilities). A number is positive or negative and these two possibilities can be represented by one bit of information. Numbers themselves can be represented by bits corresponding to yes or no. Let us see how any number from zero to seven can be represented by three bits corresponding to the yes or no answers to just three questions. Suppose a friend is thinking of a number between zero and seven and you are to determine it by asking the fewest possible questions to be answered yes or no. Suppose the friend's number is 6:

- a) Is it as big as 4? Yes.
- b) Is it as big as 4+2? Yes.
- c) Is it as big as 4+2+1? No.

From this you correctly conclude that the number is 6. You determined that the number was made up of a 4, a 2, and no 1. You might also say that the number can be represented by the sequence "yes,yes,no"!

As another example, try to guess a number between zero and 63 chosen by the friend. Suppose it is 37:

- a) Is it as big as 32? Yes.
- b) Is it as big as 32+16? No.
- c) Is it as big as 32+8? No.
- c) Is it as big as 32+4? Yes.
- d) Is it as big as 32+4+2? No.
- e) Is it as big as 32+4+1? Yes.

So the number is 37, or perhaps "yes,no,no,yes,no,yes". Try this questioning strategy on any number from zero to 63 and you will find that six questions are always sufficient to determine the number. The strategy depends on cutting the unknown range in two each time (a so-called "binary chop").

Conversely, any number between zero and 63 can be represented by a sequence of yes and no answers to six such questions. What number is represented by the sequence

yes,yes,no,yes,no,yes?

This number must be built up of a 32, a 16, no 8, a 4, no 2, and a 1. $32+16+4+1$ is 53, so the sequence represents the number 53.

Because a yes or no answer is the smallest bit of information we can extract from our friend, we say any number between zero (six nos) and 63 (six yeses) can be represented by six bits. If on the other hand we know the number is between zero and seven, three bits are sufficient to describe

the number fully. Similarly, numbers up to 15 (2^4-1) can be expressed with four bits, and numbers up to 31 (2^5-1) with five bits. Each new power of two requires another bit because it requires another yes/no question to be asked.

This method of representing numbers as a sequence of bits, each bit corresponding to a yes or no, is called "binary notation" and is the method normally used by computers. Whether a computer bit represents yes or no is typically specified by a tiny electronic switch being on or off, or by a tiny piece of iron being magnetized up or down. A TUTOR variable contains *sixty* bits of yes/no information and could therefore be used to hold a *positive integer* as big as $(2^{60}-1)$, which is approximately 10^{18} , or 1 followed by 18 zeros. What do we do about *negative integers*? Instead of using all sixty bits we could give up one bit to represent whether the number is positive or negative (again, a two-way or binary bit of information) and just use 59 bits for the magnitude of the number. In this way we could represent positive or negative *integers* up to $\pm(2^{59}-1)$, which is approximately plus or minus one-half of 10^{18} .

But what do we do about bigger numbers, or numbers such as 3.782 which are not integers? The scheme used on the CONTROL DATA® PLATO computer is analogous to the scientific notation used to express large numbers. For example, 6.02×10^{23} is a much more compact form than 602 followed by 21 zeros, and it consists of two essential pieces: the number 6.02 and the *exponent* or *power* of ten (23). Instead of using 59 bits for the number, we use only 48 bits and use 11 bits for the exponent. Of these 11 bits, one is used to say whether the exponent is positive or negative (the difference between 10^{+6} , a million, and 10^{-6} , one-millionth). The remaining ten bits are used to represent exponents as big as one thousand ($2^{10}-1$ is 1023, to be precise). The exponent is actually a power of two rather than ten, as though our scientific notation for the number 40 were written as 5×2^3 instead of 4×10^1 . That is, instead of expressing the number 40 as 4×10^1 , we express it as 5×2^3 , putting the 5 in our 48-bit number and the 3 in the 11-bit exponent storage place. In this way we split up the 60 bits as:

- 1 bit for positive or negative number
- 1 bit for positive or negative exponent
- 10 bits for the power of two
- 48 bits for the number

The 48-bit number will hold an integer as big as $(2^{48}-1)$, which is about 2.5×10^{14} . If we wish to represent the number 1/4, the variable will have a number of 2^{47} and an exponent of -49 :

$$2^{47} \times 2^{-49} = 2^{-2} = 1/4$$

That is, the 48-bit number will hold a large integer, 2^{47} , and the exponent or power of 2, will be -49 . The complicated format just described is that used by the PLATO computer when we calculate with variables v1 through v150. It automatically takes care of an enormous range of numbers by separating each number into a 48-bit number and a power of two. This format is called “fractional” or “floating-point” format because non-integral values can be expressed and the position of the decimal point floats automatically right or left as operations are performed on the variable.

Sometimes this format is not suitable, particularly when dealing with strings of characters. The `-storea-` and `-pack-` commands place ten alphanumeric characters into each variable or “word” (a computer variable is often called a “word” because it can contain several characters). We simply split up the sixty bits of the word into ten characters of six bits each, six bits being sufficient to specify one of 64 possible characters, from character number zero to character number 63 ($2^6 - 1$). In this scheme character number 1 corresponds to an “a”, number 2 to a “b”, number 26 to a “z”, number 27 to a “0”, number 28 to a “1”, etc. A capital D requires *two* 6-bit character slots including one for a “shift” character (which happens to be number 56) and one for a lower-case “d” (number 4). The `-showa-` command takes such strings of 6-bit character codes and displays the corresponding letters, numbers, or punctuation marks on the student’s screen.

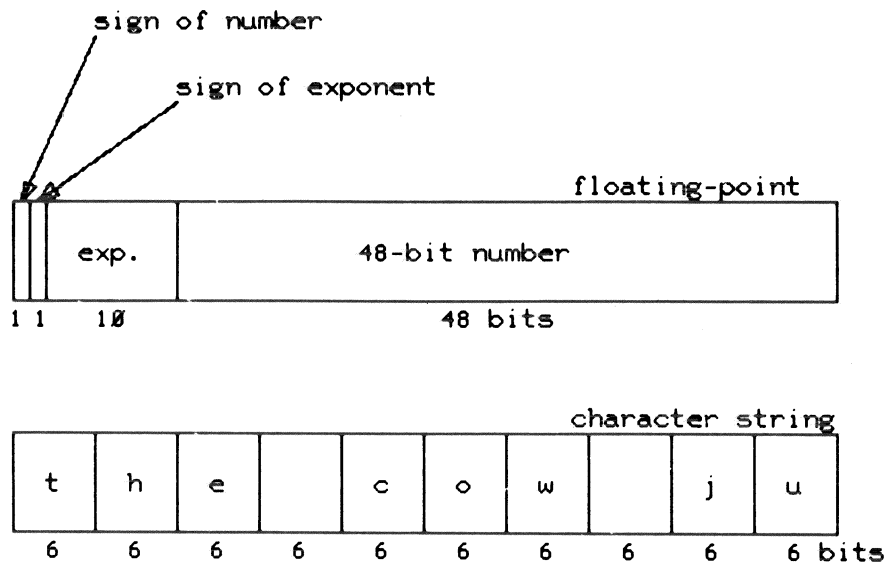


Fig. 10-1.

Nonsensical things happen when a `-showa-` command is used to display a word which contains a floating-point number. The two sign bits (for the number and for the exponent) and the first four bits of the exponent make up the first 6-bit character code. The last six bits of the exponent are taken as specifying the second 6-bit code. Then the remaining 48 bits are taken as specifying eight 6-bit character codes. Small wonder that using a `-showa-` on anything other than character strings usually puts gibberish on the screen. On the other hand, using a `-show-` with a character string gives nonsense: the floating-point exponent is made up out of pieces of the first and second 6-bit character codes, the 48-bit number comes from the last eight character codes, and whether the number and the exponent are positive or negative is determined by the first two bits of the first character code. (See Fig. 10-1)

So far we have kept numerical manipulations (`-calc-`, `-store-`, `-show-`) completely separate from character string manipulations (`-storea-`, `-showa-`). The reasons should now be clear. It is sometimes advantageous, however, to be able to use the power of `-calc-` in manipulating character strings and similar sequences of bits. For such manipulations we would like to notify TUTOR *not* to pack numbers into a variable in the useful but complicated floating-point format. This is done by referring to "integer variables":

n1,n2,n3-----n149,n150

The integer variable `n17` is the same storage place as `v17`, but its internal format will be different. If we say "`calc v17←6`", TUTOR will put into variable number 17 the number 6, expressed as 6×2^{45} with an exponent of -45 , so that the complete number is $6 \times 2^{45} \times 2^{-45}$, or 6. If on the other hand we say "`calc n17←6`", TUTOR will just put the number 6 into variable number 17. (See Fig. 10-2.) Since the number 6 requires only three bits to specify it, variable 17 will have its first 57 bits unused (unlike the situation when we refer to the 17th variable as `v17`, in which case both the exponent and the magnitude portions of the variable contain information).

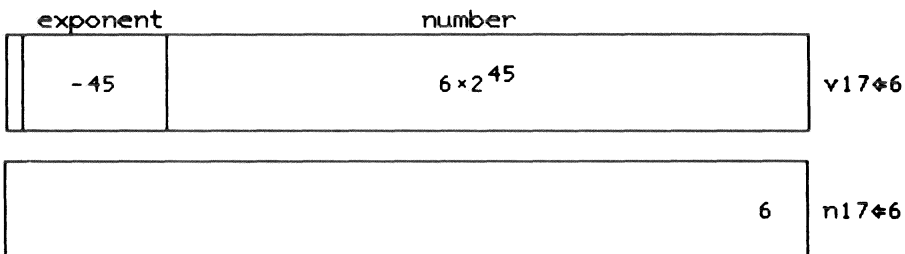


Fig. 10-2.

Consider the following sequence:

```

calc    n17←6
.
.
.
at      1223
showa  n17,10

```

This will cause an “f” (the 6th letter in the alphabet) to appear on the screen at location 1223. The first 9 character codes in n17 are zero, and these zero or “null” codes have no effect on the screen or screen positioning. Indeed, a “showa n17,9” would display nothing since the “6” is in the *tenth* character slot. If we use “show n17”, we will only see a “6” on the screen. The integer format of n17 alerts -show- not to expect a floating-point format.

If we say “calc n23←5.7”, variable n23 will be assigned the value 6. *Rounding* is performed in assigning values to integer variables. If truncation is desired, use the “int” function: “n23←int(5.7)” will assign the integer part (5) to n23. Indexed integer variables are written as “n(index)” in analogy with “v(index)”.

The -showa- and -storea- commands may be used with either v-variables or n-variables. These commands simply interpret any v- or n-variable as a character string. This is the reason why we were able to use -showa- and -storea- without discussing integer variables.

It is possible to shift the bits around inside an integer variable. In particular, a “circular left shift”, abbreviated as “\$cls\$”, will move bits to the left, with a wrap-around to the right end of the variable. For example:

```

calc    n17←6 $cls$ 54
.
.
.
at      1223
showa  n17,1 $$ show one character

```

will display an “f” even though the -showa- will display only the first character, because the “6” has been shifted left 54 bit positions (9 six-bit character positions). A circular left shift of 54 may also be thought of as a right circular shift of 6 because of the wrap-around nature of the circular shift.

We have been using “n17” as an example, but we should actually be writing “inum” or some such name, where we have used a -define- to

specify that "inum=n17". For the remainder of this chapter we revert, therefore, to the custom of referring to variables (v or n) by name rather than number. Also, if we want the character code corresponding to the letter "f" we should use "f" rather than 6. For example:

```
calc inum←"f" $cls$ 54
```

is equivalent to but much more readable than:

```
calc n17←6 $cls$ 54.
```

The quotation marks can be used to specify strings of characters. For example:

```
calc inum←"cat"
```

will put these numbers in inum:

| | | | | | | | | | |
|------|------|------|------|------|------|------|---|---|----|
| null | null | null | null | null | null | null | c | a | t |
| ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | ∅ | 3 | 1 | 2∅ |

Fig. 10-3.

A "showa inum,1∅" will display "cat". Notice, particularly, that using quotes in a -calc- to define a character string puts the string at the *right* ("right adjusted"), whereas the -storea- and -pack- commands produce *left*-adjusted character strings. It is possible to create left-adjusted character strings by using *single* quote marks: inum←'cat' will place the "cat" in the first three character positions rather than the last three.

Let us now return to our early example of the number 37 expressed as the sequence of six bits "yes,no,no,yes,no,yes". If we let 1 stand for "yes", and ∅ for "no", we might write this sequence as:

1∅∅1∅1

which stands for:

$$(1 \times 32) + (\emptyset \times 16) + (\emptyset \times 8) + (1 \times 4) + (\emptyset \times 2) + (1 \times 1) = 32 + \emptyset + \emptyset + 4 + \emptyset + 1 = 37$$

or even more suggestively:

The TUTOR Language

$$(1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 32 + 0 + 0 + 4 + 0 + 1 = 37$$

(Note that 2^0 equals 1.) Writing the sequence in this way is analogous to writing 524 as:

$$(5 \times 10^2) + (2 \times 10^1) + (4 \times 10^0) = 500 + 20 + 4 = 524$$

In other words, when we write 524 we imply a “place notation” in base 10 such that each digit is associated with a power of 10: 5×10^2 , 2×10^1 , 4×10^0 . Similarly, rewriting our yes and no sequences as 1 and 0 sequences, we find that the string of ones and zeros turns out to be the place notation in base 2 for the number being represented.

Here are some examples. (1001_2 means 1001 in base 2.)

$$\begin{aligned} 1001_2 &= 2^3 + 2^0 = 8 + 1 = 9 \\ 1100_2 &= 2^3 + 2^2 = 8 + 4 = 12 \\ 110101_2 &= 2^5 + 2^4 + 2^2 + 2^0 = 32 + 16 + 4 + 1 = 53 \\ 1000001_2 &= 2^6 + 2^0 = 64 + 1 = 65 \end{aligned}$$

This base 2 (or “binary”) notation can be used to represent any pattern of bits in an integer variable, and with some practice you can mentally convert back and forth between base 10 and base 2. This becomes important if you perform certain kinds of bit manipulations.

An important property of binary representations is that shifting left or right is equivalent to multiplying or dividing. Consider these examples:

$$\begin{array}{l} \leftarrow \text{shift left 2 places} \\ 9 \text{ \$cls\$ } 2 = 1001_2 \text{ \$cls\$ } 2 = 100100_2 = 36 \\ \text{(left shift 2 is like multiplying by } 2^2 \text{ or 4)} \end{array}$$

$$\begin{array}{l} \leftarrow \text{shift left 3 places} \\ 9 \text{ \$cls\$ } 3 = 1001000_2 = 72 \\ \text{(left shift 3 like multiplying by } 2^3 \text{ or 8)} \end{array}$$

So, a *left* shift of N bit positions is equivalent to *multiplying* by 2^N . A *right* shift of N bit positions is equivalent to *division* by 2^N (assuming no bits wrap around to the left end in a $60-N$). There exists an “arithmetic right shift”, $\$ars\$, which is not circular but simply throws away any bits that fall off the right end of the word:$

$$9 \text{ \$ars\$ } 3 = 1001_2 \text{ \$ars\$ } 3 = \overset{\text{thrown away}}{1}0\cancel{0}1 = 1.$$

This corresponds to a division by 2^3 , with truncation ($9/2^3 = 9/8$ which truncates to 1).

A major use of the 60 bits held in an integer variable is to pack into one word many pieces of information. For example, you might have 60 “flags” set up or down (1 or 0) to indicate 60 yes or no conditions, perhaps corresponding to whether each of 60 drill items has been answered correctly or not. Or you might keep fifteen 4-bit counters in one word: each 4-bit counter could count from zero to as high as 15 (2^4-1) to keep track of how well the student did on each of fifteen problems. Ten bits is sufficient to specify integers as large as 1023: you could store six 10-bit baseball batting averages in one word, with suitable normalizations. Suppose a batting average is .324. Multiply by a thousand to make it an integer (324) and store this integer in one of the 10-bit slots. When you withdraw this integer, divide it by a thousand to rescale it to a fraction (.324). When we discussed arrays we had exam scores ranging from zero to 100. The next larger power of two is 128 (2^7), so we need only 7 bits for each integer exam score. Eight such 7-bit quantities could be stored in one 60-bit word.

How do you extract a piece of information packed in a word? As an example, suppose you want three bits located in the 19th of twenty 3-bit slots of variable “spack”:

$$\text{inum} \leftarrow (\text{spack \$ars\$ } 3) \text{ \$mask\$ } 7$$

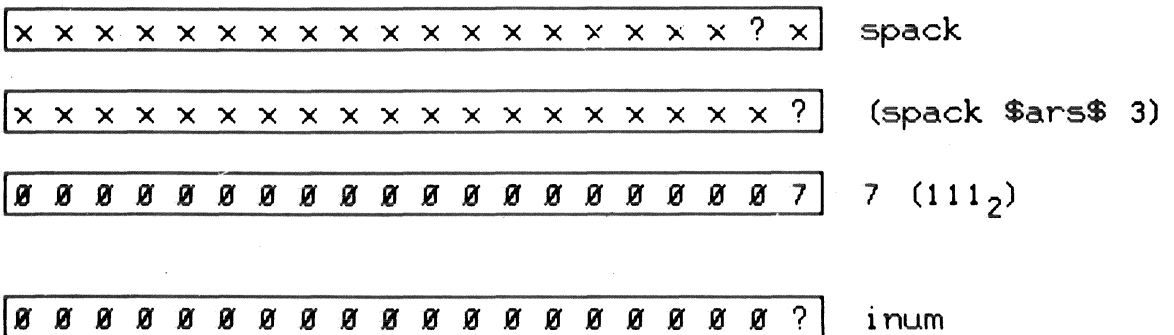
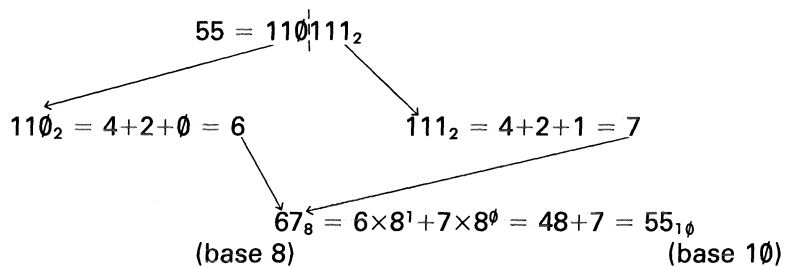


Fig. 10-4.

The number 7 is 111_2 (base 2: $4+2+1$), so it is a 3-bit quantity with all three bits “set” or “on” (non-zero). The $\$mask\$$ operation pulls out the corresponding part of the other word, the 3-bit piece we are interested in. In an expression $(x \$mask\$ y)$, the result will have bits set (1) only in those bit positions where both x and y have bits set. In those bit positions where either x or y have bits which are “reset” or “off” (0), the $\$mask\$$ operation produces a 0. We could also have used a “segment” definition to split up the word into 3-bit segments.

A 4-bit mask would be 15 (1111_2) and a 5-bit mask would be 31 (11111_2). (Again, “segment” definitions of 4 or 5 bits could be used.) You might even need a mask such as 110111_2 (or 55) which will extract bits located in the five bit positions where 110111_2 has bits set. There should be a simpler way of writing down numbers corresponding to particular bit patterns. Certainly, reading the number 55 does not immediately conjure up the bit pattern 110111_2 !

A compact way of expressing patterns of bits depends on whether or not each set of three bits can represent a number from 0 to 7:



Just as each digit in a decimal number (base 10) runs from 0 to 9, so do the individual numerals run from 0 to 7 in an octal number (base 8). Octal numbers are useful only because they represent a compact way of expressing bit patterns. With practice, you should be able to convert between octal and base 2 instantaneously, and between base 8 and base 10 somewhat slower! See the table below.

| | | | |
|---------|--------|------------------------------------|-----------|
| base 10 | base 8 | These should be memorized | base 2 |
| 0 | 0 | | 0 or 000 |
| 1 | 1 | | 1 or 001 |
| 2 | 2 | | 10 or 010 |
| 3 | 3 | | 11 or 011 |
| 4 | 4 | | 100 |
| 5 | 5 | | 101 |
| 6 | 6 | | 110 |
| 7 | 7 | 111 | |

| base 10 (continued) | base 8 (continued) | base 2 (continued) |
|------------------------|-----------------------|-----------------------|
| 8 | 10 | 1000 |
| 9 | 11 | 1001 |
| 10 | 12 | 1010 |
| 11 | 13 | 1011 |
| 12 | 14 | 1100 |
| 13 | 15 | 1101 |

The conversion between base 8 and base 2 is a matter of memorizing the first eight patterns, after which translating 1101011011101_2 to octal is simply a matter of drawing some dividers every three bits:

$$\begin{array}{c|c|c|c|c} 1 & 101 & 011 & 011 & 101 \\ \hline 1 & 5 & 3 & 3 & 5 \end{array} = 15335_8$$

What is 15335_8 in base 10?

$$\begin{array}{c|c|c|c|c} 8^4 & 8^3 & 8^2 & 8^1 & 8^0 \\ \hline 4096 & 512 & 64 & 8 & 1 \end{array}$$

$$1 \quad 5 \quad 3 \quad 3 \quad 5 = 1 \times 4096 + 5 \times 512 + 3 \times 64 + 3 \times 8 + 5 = 5853_{10}$$

How about the octal version of the number 79? The biggest power of 8 in 79 is 8^2 (64), and 79 is 15 more than 64. In turn, 15 is $1 \times 8^1 + 7 \times 8^0$, so:

$$79_{10} = 1 \times 64 + 1 \times 8 + 7 \times 1 = 1 \times 8^2 + 1 \times 8^1 + 7 \times 8^0 = 117_8$$

Luckily, in bit manipulations the conversions between base 2 and base 8 are more important than the harder conversions between base 8 and base 10.

To express an octal number in TUTOR, use an initial letter “o”:

`x $mask$ o37`

will extract the right-most 5 bits from x, because $o37 = 37_8 = 011111_2$, which has 5 bits set. Naturally, a number starting with the letter “o” must not contain 8’s or 9’s.

You can display an octal number with a `-showo-` command (show octal):

`showo 39`

will display “0000000000000000000047” on the screen ($39_{10}=47_8$). The default format is twenty (3-bit) octads, corresponding to a whole 60-bit word:

```
showo 39,4
```

will display “0047”, showing just four octads.

Now that we have discussed the octal notation, it is possible to point out what happens to negative numbers:

```
showo -39
```

will display “77777777777777777730”. A negative number is the “complement” of the positive number (binary 1’s are changed to 0’s and binary 0’s are changed to 1’s). In octal, the complement of 0 is 7 ($000_2 \rightarrow 111_2 = 7_8$), and the complement of 7 is 0. In the example shown, octal 47_8 is 100111_2 , whose complement is 011000_2 , or 30_8 . Notice that the *left*-most bit (the “sign” bit) of a negative number is always set. In order for a negative number to stay negative upon performing an “arithmetic right shift”, all the left-most bits are set. So,

```
o400000000000000000003242 $ars$ 6
```

yields:

```
o77400000000000000000032.
```

Only the sign bit was set among the left-most bits before the shift (o40 is 100000_2), but after the shift the first seven bits are all set. The “circular left shift”, \$cls\$, does not do anything special with the sign bit.

It is interesting to see the bits set for floating-point numbers:

```
.
.
.
.
calc v1←3
at 1215
write pos=<o,v1> $$ o for -showo-
neg=<o,-v1>
```

will make this display:

```
pos = 17216000000000000000
neg = 6057177777777777777
```

Note that the negative number is the complement of the positive. The 48-bit magnitude (600000000000000000) represents a huge integer (6×2^{45}). The eleven bits between the sign bit and the 48-bit magnitude give the power of two (-46) by which the magnitude is to be scaled ($3 = 6 \times 2^{45} \times 2^{-46} = 6 \times 2^{-1} = 3$). A bias of 2000_8 is added to the correct exponent (-46 , or -56_8) to give an eleven-bit exponent of 1721_8 . Exponents less than 2000_8 represent negative powers and exponents greater than 2000_8 represent positive powers.

We have encountered octal numbers (e.g., $o327$) which can be shifted left ($\$cls\$$) and right ($\$ars\$$) and complemented (by making them negative). Pieces can be extracted with a $\$mask\$$ operation. Additional bit operations are $\$union\$$, $\$diff\$$, and “bitcnt”. The “bitcnt” function gives the number of bits set in a word: $bitcnt(o25)$ is 3, because $o25$ is 010101_2 , which has 3 bits set; $bitcnt(-o25)$ is 57, since the complement will have only 3 of 60 bits *not* set; and $bitcnt(0)$ is 0. Like $\$mask\$$, $\$union\$$ and $\$diff\$$ operate on the individual bit positions, with all 60 done at once:

- x $\$mask\$$ y produces a 1 only where *both* x and y have 1's.
- x $\$union\$$ y produces a 1 where *either* x or y or *both* have 1's.
- x $\$diff\$$ y produces a 1 only where x and y *differ*.

Note that $\$union\$$ might be called “merge”, since 1's will appear in every bit position where either x or y have bits set. The $\$diff\$$ operation might also be referred to as an “exclusive” union, since it will merge bits except for those places where *both* x and y have bits set.

While $\$mask\$$ can be used to extract a piece of information from a word, a $\$mask\$$ that includes all *but* that piece followed by a $\$union\$$ can be used to insert a new piece of information.

These bit operations can be used with arrays. For example, if A, B, and C are true arrays, the statement “ $C \leftarrow A \$diff\$ B$ ” will replace each element of C by the bit difference of the corresponding elements of A and B.

Byte Manipulation

The most common use of bit manipulations is for packing and unpacking “bytes” consisting of several bits from words each of which contain several bytes. This can lead to major savings in space. If an exam score lies always between 0 and 100, only seven bits are required to hold each score, since $(2^7 - 1)$ is 127. Another way to see this is to write the largest 7-bit quantity: $1111111_2 = 177_8 = 1 \times 8^2 + 7 \times 8^1 + 7 \times 8^0 = 64 + 56 + 7 = 127$. This is one less than 2000_8 , which requires an eighth bit. We can fit

eight 7-bit bytes into each 60-bit word. Happily, TUTOR will do the bookkeeping, as we saw earlier:

```
define segment,scores=n31,7
```

This definition makes it possible to work with this “segmented” array as though it were an ordinary array:

```
calc ss←scores(3)
      scores(17)←83
      etc.
```

These refer to the 3rd and 17th bytes. The first eight 7-bit bytes reside in n31, with the last 4 bits unused. The next eight bytes are in n32, etc. The 17th byte is the first 7-bit byte in n33.

Just as it is possible to give up one bit of a 60-bit word in order to have negative as well as positive numbers, so it is possible to have both positive and negative numbers stored in a segment array:

```
define segment,temp=v52,8,signed
.
.
.
.
calc temp(23)←-95
```

With 8-bit bytes we can have numbers in the range of ± 127 . The word “signed” may be abbreviated by “s”.

Now that you understand the bit structure of a variable, you should be able to understand the table (Table 10-1) provided earlier of segment ranges and the number of segments per variable. Look at the table now and see whether you can check the entries in the table.

Vertical Segments

We might call the segments discussed so far “horizontal” segments (the segments move horizontally across each word). It is possible to define “vertical” segments (each of which occupies only part of a word): successive segments are found in the same position in successive words, rather than in different positions within the same word. As an example, “define segmentv,left=n51,1,30” defines vertical segments each occupying the left half of words n51, n52, n53, etc. Each segment

starts in bit position 1 of each word, and each segment is 30 bits long. The right halves of the words could be specified with “define segmentv, right=n51,31,30”, whose elements begin in the 31st bit position and are 30 bits wide. An “s” can be added to denote signed segments, as with horizontal segments.

Aside from the intrinsic usefulness of this kind of segmenting of words, the simpler structure permits TUTOR to process vertical segments much faster than horizontal segments, and only slightly slower than normal whole-word variables.

You can save space with true arrays by putting the elements in vertical segments. The -define- statement looks like “define arraysegv, A(10)=n5,3,12,s”. This example defines a ten-element array, with A(1) represented by a 12-bit signed segment starting in bit position 3 of n5. It is not yet possible to define a true array in horizontal segments.

Alphanumeric to Numeric: The -compute- Command

The -store- command analyzes the judging copy of the student’s response character string and produces a numerical result. This is actually a two-step process. First, the character string is “compiled” into basic computer instructions and then these machine instructions are “executed” to produce the numerical result. During the compilation process the “define student” definitions and the built-in function definitions (sin, cos, arctan, etc.) are used to recognize the meaning of names appearing in the character string. Numbers expressed as alphanumeric digits are converted to true numerical quantities. For example, the character string 49 becomes a number by a surprisingly indirect process. The character code for “4” is 31 since “z” is 26, “0” is 27, etc. The character code for “9” is 36. The number expressed by typing 49 is obtained from the formula:

$$\begin{aligned} &10(31-27)+(36-27) \text{ or } 10("4"- "0") + ("9"- "0") \\ &10(4)+(9) \\ &40+9 \\ &49 \end{aligned}$$

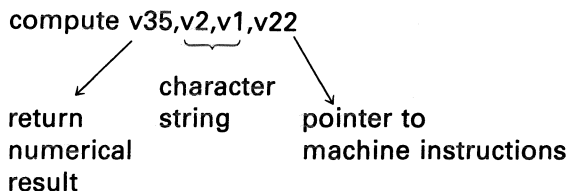
For these and similar reasons, the compilation process is ten to a hundred times slower than the execution process. Therefore, TUTOR attempts to compile the student’s response only once, while the resulting machine instructions may be used many times.

The first -store-, -ansv-, -wrongv-, -storeu-, -ansu-, or -wrongu-command encountered during judging triggers compilation. All these commands following the first one simply reuse the compiled machine instructions. If a -bump- or -put- makes any changes in the judging copy, a following -store- or related command will have to recompile. Similarly, a “judge rejudge” will force recompilation by any of these commands. Note that re-execution is *always* performed even if recompilation isn't, because the student might refer to defined variables whose values have been altered.

While -store- will compile and execute from the judging copy, the *regular* -compute- command will compile and execute from *any* stored character string:

compute result,string,#characters,pointer

For example:



After compilation, the “pointer to machine instructions” contains the location of the machine instructions in a special -compute- storage area. You must zero the pointer at first to force compilation. TUTOR will then set the pointer appropriately, so that re-executions of the -compute-command can simply re-execute the saved machine instructions. Here is a unit which permits the student to plot functions of interest to him or her.

```

define      student
            x=v1
define      ours,student
            result=v2,string=v3,point=v35
origin      100,250
bounds      0,-200,300,200
scalex      10
scaley      2
*
unit        graph
next        graph
  
```


| | | |
|---------|----------------------------|------------------------|
| back | graph | |
| axes | | \$\$ display the axes |
| labelx | 1 | |
| labely | 0.2 | |
| at | 3105 | |
| write | Type a function of x: | |
| arrow | where +2 | |
| storea | string,jcount | |
| ok | | |
| calc | x←point←0 | |
| compute | result,string,jcount,point | |
| goto | formok,x,badform | |
| gat | 0,result | \$\$ draw from here |
| doto | 8plot,x←.1,10,.1 | |
| compute | result,string,jcount,point | |
| goto | formok,x,badform | |
| gdraw | ;x,result | |
| 8plot | | |
| * | | |
| unit | badform | |
| at | 3207 | |
| writec | formok, . . . | \$\$ tell what's wrong |
| judge | wrong | |

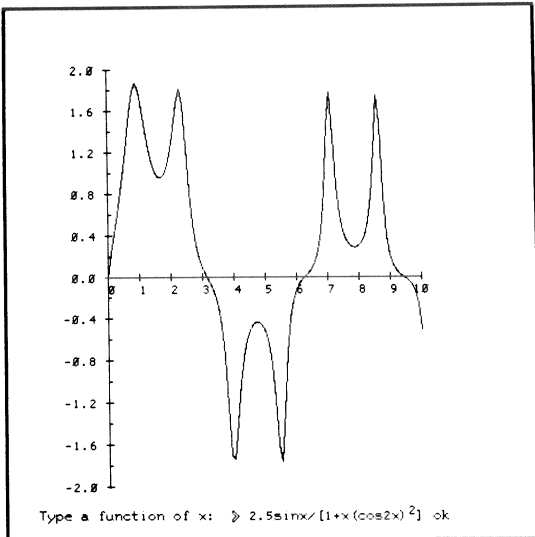


Fig. 10-5.

Different functions can be superimposed by changing the response instead of pressing NEXT or BACK. The first -compute- in this unit calculates the value of the student's function for x equal to zero. The -gat-command positions us at location (0, result) so that the first -gdraw- will draw a line starting at that point. The system variable "formok" has the value -1, if compilation and execution succeed; 0 if compilation succeeds but execution fails (due to such errors as trying to take the square root of a negative number); and various positive integral values for various compilation errors (missing parentheses, unrecognized variable names, etc.).

Note that predefined functions can be more easily plotted with a -funct- command. For example, the student could specify a value for "n", and you could plot a polynomial simply by using "funct x^n,x←0,10,.1". But, you must use -compute- if the student is permitted to try arbitrary functions of his or her own choosing.

As another example, the PLATO lesson "grait" (written by this author) permits the student to write up to fifteen statements in the grafit

The TUTOR Language

language and execute his or her program to produce graphical output (as seen in Fig. 10-6):

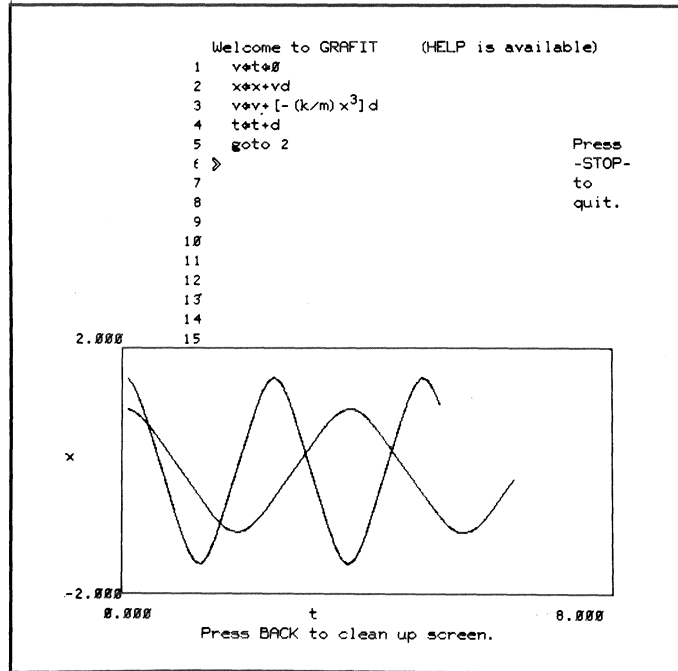


Fig. 10-6.

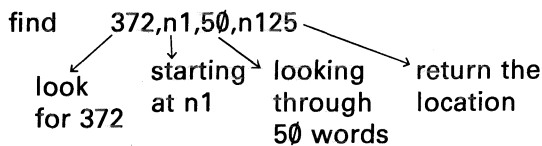
This student's program calculates the motion of a mass oscillating on the end of a non-standard spring. The two curves are the superposition of running the program twice with different values of the parameters. The heart of this lesson is a loop through a -compute- command with string, character count, and point all being indexed variables. The index is the line number, from 1 to 15. Each student response is analyzed using a -match- command looking for keywords such as "goto". Then the rest of the response is filed away with a -storea- into the string storage area corresponding to that line number. The 15 pointer variables are zeroed in the "ieu" (initial entry unit) to insure that when the student returns to a PLATO terminal after several days TUTOR won't be confused over whether the strings have been recently compiled or not. Also, whenever the student changes one of his or her statements, the corresponding pointer is zeroed in order to force recompilation of the altered character string. The student can press DATA to initialize parameters, LAB to specify what variable to plot against what variable, and HELP for a description of the grafit language. The student define set defines all 26 letters as variables the student can use.

Note that even though *s*, *i*, and *n* have been defined in the student define set, the student can use the “sin” function. The reason that the student’s “sin” is not interpreted as $s \times i \times n$ is that TUTOR looks for the longest possible name in a string of characters typed by the student. One difference between the handling of student expressions and author expressions is that students cannot reference system variables such as “where”, “anscnt”, or “data” (the numerical value of the DATA key). If you want the student to be able to use “where”, define it in the student define set as “where=where”. While authors are discouraged from using primitive names such as *v47* (except in a -define- statement), students are not permitted to use primitives at all. This is done to protect the author’s internal information. Similarly, students cannot use the assignment symbol (\Leftarrow), except in a -compute-, unless there is a “specs okassign”.

It should be mentioned that while -compute- converts alphanumeric information into a numerical result, there is an -itoa- command that can be used to convert an integer to an alphanumeric character string. Most often, however, the -pack- command with embedded -show- commands will be used to convert non-integer as well as integer values to the corresponding character strings.

The -find- Command

The -search- command discussed in Chapter 8 is character-string oriented and will locate ‘dog’ even across variable or word boundaries: the “d” might be at the end of one word and the “og” at the beginning of the next word. The -find- command, in contrast, is word oriented. It will find which word contains a certain number or character string:



If *n1* contains 372, *n125* will return the value 0; if *n2* is the first word which contains 372, *n125* will be 1; etc. If none of the 50 words contains 372, *n125* will be set to -1. Notice that in -search- the return is 1, not 0, if the string is found immediately. This is due to the fact that in character strings we start numbering with character number 1. On the other hand, here the first word is $n(1+0)$.

Do not use *v*-variables in the first two arguments of -find- because -find- makes its comparisons by integer operations. The first argument can be a character string such as ‘dog’ or “dog”. You can look at every 3rd word by specifying an optional increment:

```
find "cat",n1,50,n125,3
                        }
                        optional
```

This will look for "cat" in n1, n4, n7, etc., and n125 would be returned 0, or 3, or 6, etc. Negative increments can be used to search backwards from the end of the list.

You can also specify that a "masked equality search" be made:

```
find "cat",n1,50,n125,1,o777700
                        }      ↘ mask
                        not optional
```

In this case, n125 will be zero if [(n1 \$diff\$ "cat") \$mask\$ o777700] is zero. The mask specifies that only a part of the word will be examined. The increment must be specified, even if it is one, to avoid ambiguity.

There is a `-findall-` command which will produce a list of all of the locations where something was found, rather than producing locations one at a time.

The `-exit-` Command

Suppose you are seven levels deep in `-do-`s. That is, you have encountered seven nested `-do-` statements on the way to the present unit. The statement `"exit 2"` will take you out two levels. The next statement to be executed is the statement which follows the sixth `-do-`. A blank `-exit-` command (blank tag) takes you immediately to the statement following the first `-do-`. (Such operations are occasionally useful.) Notice that encountering a unit command at the end of a done subroutine will cause an automatic `"exit 1"`. It is superfluous to put `"exit 1"` at the end of a unit, since this effect is automatic.