Doing Calculations in TUTOR

4

You can make TUTOR calculate things for you. For example:

```
        .
        .
        .
at      12Ø1
write   Who is buried
        in Grant's tomb?
arrow   12Ø1+3Ø8
        .
        .
        .
```

The -arrow- statement, as written, is completely equivalent to "arrow 15Ø9". Or consider this:

```
circle  (41²+72.6²)^½
```

The radius of the circle will be taken to be the square root of the sum of 41 squared and 72.6 squared.

Just about any expression that would have made sense to your high school algebra teacher will be understood and correctly evaluated. For example:

Expression	TUTOR Evaluation
$3.4+5(2^3-3)/2$	15.9
2×3+8	14 (NOT 22)
sin(3Ø°)	Ø.5 (See Appendix C for other functions.)
$49^{1/2}$	7
(4+7)(3+6)	99
$6/5\times1Ø^{-3}$	12ØØ (NOT $1.2\times1Ø^{-3}$)

If your high school algebra is rusty, we remind you that "2×5+3" means "(2×5)+3" which is 13, *not* "2×(5+3)" which is 16. The rule is that multiplication is "more important" than addition or subtraction and gets done *first.* If you are unsure at some point, you may use parentheses around several portions of your expression to make the meaning unambiguous.

A similar point holds true for division, which is considered "more important" than addition or subtraction. "8+6/2" means "8+(6/2)" which is 11, *not* "(8+6)/2" which would be 7. The only ticklish point is whether multiplication is more or less "important" than division. TUTOR agrees with most mathematical books and journals that multiplication is more important than division, so that "6×4/3×2" means "(6×4)/(3×2)" which is 4. Note that this means that TUTOR considers "1/2(6+4)" to be "1/(2(6+4))" which is Ø.Ø5, *not* "(1/2)(6+4)" which would be 5. Again, when in doubt use parentheses. You could write ".5(6+4)" if you wish, which is unambiguous.

Experience has shown that students tend to write algebraic responses according to these rules, and making TUTOR conform to these rules facilitates the correct judging of student algebraic and numerical responses.

Having seen how expressions are handled, we can introduce "student variables" which may be used to hold numerical values obtained by evaluating expressions. These stored results can be used later in the lesson. As an example, a "variable" might hold the student's score on a diagnostic quiz, and this score could be used later to determine how much drill to give the student. The storage place is called a "variable" because what it holds may *vary* at different times in the lesson. Another variable might count the number of times the student has requested help, in which case the number which it holds would vary from Ø to 1 to 2, etc.

There are 15Ø "student variables" which can be used for storing up to 15Ø numerical values. These "student variables" are unimaginatively called:

v1, v2, v3, . . . v148, v149, v15Ø.

Later in this section we will learn how to give variables names (such as "radius," "wrongs," "tries," "speed," etc.) which are appropriate to their particular usage in a specific lesson. But first, we will look at variables using their primitive names: v1 through v15∅.

These variables are called *student* variables because each of the many students who may simultaneously be studying your lesson has his or her own private set of 15∅ variables. You might use variable v23 to count the number of correct responses on a certain topic, which will be different for each student. If there are forty students working on your lesson, TUTOR is keeping track of forty different "v23's", each one different. This is done automatically for you, so that you can write the lesson with one individual student in mind, and v23 may be considered simply as containing that individual student's count of correct responses. Thus, one student might be sent to a remedial unit because the contents of his variable number 23 show that he did poorly on this topic. Another student might be pushed ahead because the contents of her variable 23 indicate an excellent grasp of the material. It is through manipulation of the student variables that a lesson can be highly individualized for each student.

Variables are useful in building certain kinds of displays. Let's see how to build a subroutine which can draw a half-circle in various sizes, depending on variables which we set up. In order to specify the size of the figure and its location on the screen, we must specify a center (x and y) and a radius. We let variables v1 and v2 hold the horizontal x and vertical y positions of the center, and we let variable v3 hold the value for the radius.

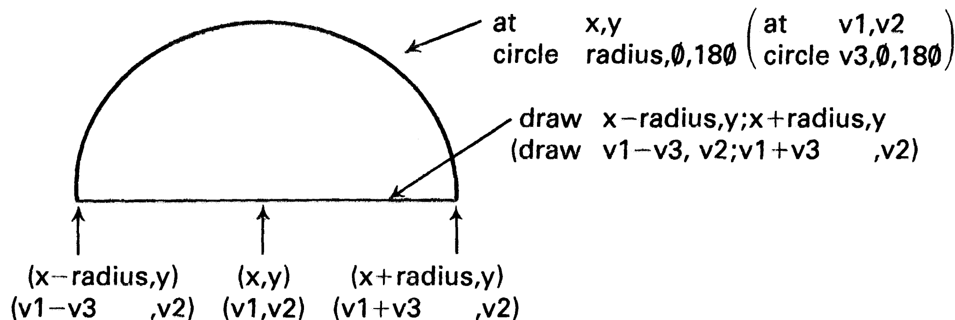


Fig. 4-1.

We can draw such a figure with the following unit:

```
unit    halfcirc
at      v1,v2
circle  v3,0,180                $$ 180 degree arc
draw    v1−v3,v2;v1+v3,v2       $$ horizontal line
```

In order to use this subroutine we might write:

```
unit  vary
calc  v1⇐150       $$ x center at 150
calc  v2⇐300       $$ y center at 300
calc  v3⇐100       $$ radius 100
do    halfcirc
calc  v1⇐v1+v3     $$ increment x center
do    halfcirc     $$ y and radius unchanged
```

The statement "calc v2⇐150" means "perform a calculation to put the number 150 in variable v2". The statement "calc v1⇐v1+v3" means "calculate the sum of the numbers presently held in variables v1 and v3, and put the result in variable v1". In the present case, this operation will store the number 250 (150+100) in variable v1 for use in the second "do halfcirc". Note that the second "do halfcirc" will use the original values of v2 and v3, which have not been changed. This unit will produce this picture:

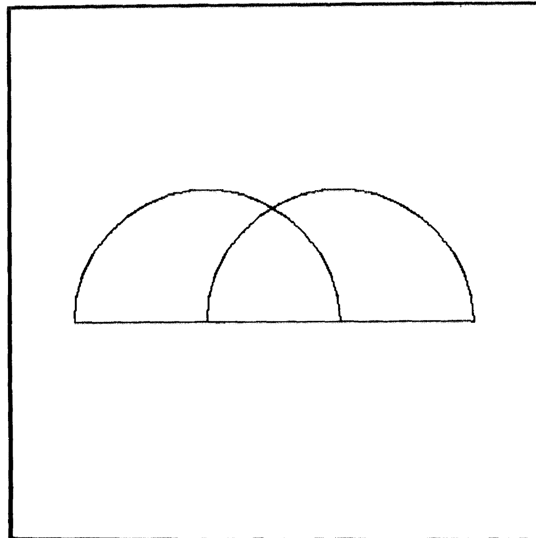


Fig. 4-2.

The ⇐ symbol is called the "assignment" symbol, because it assigns a numerical value to the variable on its left. This numerical value is obtained by evaluating the expression to the right of the assignment symbol.

A slightly more complicated example of a -calc- statement is:

```
calc   v3⇐5v2+v1
```

This statement means "multiply by 5 the number currently held in v2, add this to the number held in v1, and store the result in v3." In conversation you might read this as "calc v3 assigned five v2 plus v1" or "calc v3 becomes five v2 plus v1". Notice that it is common practice to refer simply to "v2" when we really mean "the number currently held in variable v2".

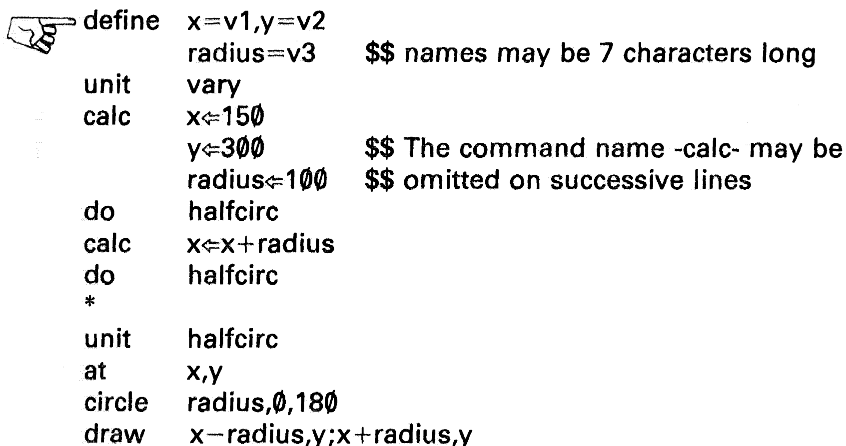
The simplest possible -calc- statement merely assigns a number to a variable, as in "calc   v2⇐15Ø". It is permissible to make more than one assignment in a -calc- statement:

calc   v3⇐v7⇐18.6^2

This will assign the value 18.6^2 to both variables v3 and v7.

Giving Names to Variables: -define-

Your programming can be made much more readable by "defining" suitable names for the student variables which you use. For example, in the units just discussed, the quantities of interest were the center (x and y) and radius of the circular arc. We should *precede* such units with a -define- statement:



```
define  x=v1,y=v2
        radius=v3     $$ names may be 7 characters long
unit    vary
calc    x⇐15Ø
        y⇐3ØØ         $$ The command name -calc- may be
        radius⇐1ØØ    $$ omitted on successive lines
do      halfcirc
calc    x⇐x+radius
do      halfcirc
*
unit    halfcirc
at      x,y
circle  radius,Ø,18Ø
draw    x-radius,y;x+radius,y
```

The -define- statement tells TUTOR how to interpret the defined names when they are encountered later in expressions. The units are now much more readable than they were when we used v1, v2, and v3.

Giving meaningful names to the variables you use is very important. After an absence of several months, you would have difficulty in remembering what you are keeping in, say, variable v26, whereas the name "tries" would remind you immediately that this variable holds a count of the number of times the student has tried to answer the question. The importance of readability is even more vital if a colleague is working with you on the material. Your associate would find it extremely frustrating to try to figure out what you are keeping in v26. So, use -define-!

There should not be any v3's or v26's anywhere in your lesson except in the -define- statement itself. Put all your definitions at the very beginning of the lesson where you will have ready reference to the variables you are using.

The only reason we started out using the primitive v-names was to establish a more concrete feeling for the meaning of a student variable. From here on we will use defined variable names. A preceding -define- statement is assumed.

WARNING: Normal algebraic notation permits expressions such as "rcosθ", but in TUTOR you must write "r×cos(θ)" or "r(cos(θ))". That is, you *must* use an explicit multiplication sign between names (either your defined names such as "r" or TUTOR-defined names such as "cos"), and you *must* place parentheses around the arguments of functions. For example, the "θ" in cos (θ) must be enclosed in parentheses.

The reason for this is that TUTOR cannot cope with the ambiguities of trying to decide whether an expression such as "abc" means "a×bc" (if there is a name "bc"), or "ab×c" (if there is a name "ab"), etc. Later, when we discuss the important topic of judging student responses, we will see that TUTOR can make reasonable guesses when treating a *student's* algebraic response and can permit the student the luxury of leaving out multiplication signs and omitting parentheses around function arguments. You, the author, are required to be more explicit, however, in separating one name from another. Notice that "17angle" is fine and TUTOR will recognize this as meaning "17×angle". But "rangle" can't be pulled apart into "(r)(angle)" because you *might* have meant "(ran)(gle)".

Repeated Operations: The Iterative -do-

With very little effort we can make a variety of designs out of our unit "halfcirc". For example:

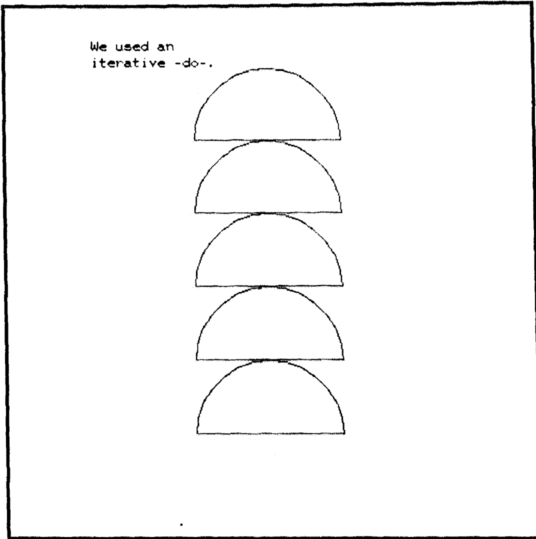


Fig. 4-3.

```
unit   stack
calc   x⇐256
       radius⇐7Ø
do     halfcirc,y⇐1ØØ,38Ø,7Ø
at     312
write  We used an
       iterative -do-.
```

The effect of the -do- statement is to set y to 1ØØ and do unit "halfcirc", then set y to 17Ø (the starting value of 1ØØ plus an increment of 7Ø) and do halfcirc again, and repeat the process until y reaches the final value of 38Ø. The format of the extremely useful iterative -do- statement is:

```
do   unitname,index⇐start,end,increment
```

In the above example, the index "y" starts at 1ØØ and goes to 38Ø in increments of 7Ø. If no increment is specified, an increment of one is assumed. For example, "do halfcirc, radius⇐1Ø1,1Ø5" will make an arc five dots wide, as in the following figure:

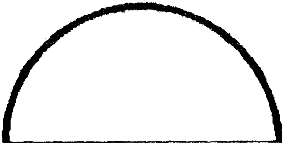


Fig. 4-4.

The iterative -do- statement also helps in making animations. The following statements will cause the half-circle to move horizontally across the screen. (See Figures 4-5a and 4-5b.)

```
unit      march
at        312Ø
write     Move figure left to right.
calc      y⇐28Ø
          radius⇐75
do        anim,x⇐1ØØ,35Ø,5Ø
do        halfcirc          $$ draw final figure
at        322Ø
write     All done.
*
unit      anim
do        halfcirc          $$ draw figure
catchup                     $$ wait for it to finish
pause     1                 $$ pause an additional second
mode      erase
do        halfcirc          $$ erase the figure
mode      write
```

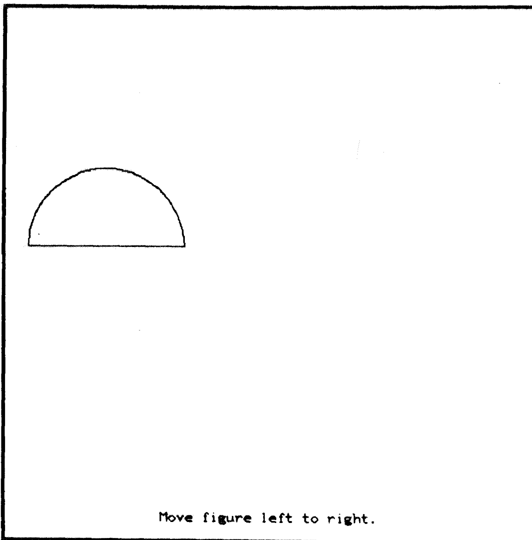


Fig. 4-5a.

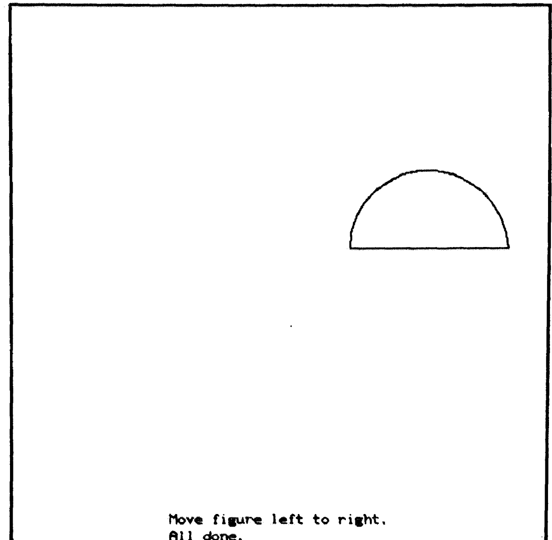


Fig. 4-5b.

We simply -do- unit "anim" repeatedly for different values of x (the horizontal position of the figure on the screen). Unit "anim" does unit "halfcirc" twice, once to draw and once to erase the figure interrupted by a one-second pause. The -catchup- command insures that a second will elapse from the end of drawing the figure on the screen until the beginning of erasing it.

Now that you have studied -define-, -calc-, and -do-, you have learned the basic techniques of how to tell PLATO what calculations you want performed. We have applied these tools to a variety of display generation problems, and we will later use calculations for controlling sequencing in a lesson and for judging responses. Hopefully, you have gained added insight into the value of a subroutine. Notice how many different ways we have used the single unit "halfcirc"!

Showing the Value of a Variable

We have learned how to calculate and how to store results in variables. How do we show these results on the screen? Suppose we perform this calculation:

```
calc   y⇐5sqrt(37)   $$ or, y⇐5×37½ ; "sqrt" means square root
```

How do we later show the value of y? Assume we have defined y. Perhaps we could use this:

```
write   y
```

No, that won't work; that will just put the letter "y" on the screen. The -write- command is basically a device for displaying non-varying text, not for showing the value contained in a variable. We need another command:

```
show   y
```

This will show the value of y in an appropriate format (-show- picks an appropriate number of significant figures and will use a scientific format such as 6.7×10^{13}, if the number is large enough to require it). By using -show- instead of -write-, you tell TUTOR that you want the stored value to be shown rather than just the characters in the tag.

The -show- command will normally choose 4 significant figures, so that a typical display might be "−23.47". You can specify a different value by giving a second "argument" (arguments are the individual pieces of the tag of a statement):

```
show   y,8   $$ 8 significant figures
```

The arguments of the -show- command can, of course, be complicated expressions:

```
show   1∅+3∅cos(2angle),format+2
```

In fact, it is a general rule that you can use complicated expressions anywhere in TUTOR statements. For example, "draw 5rad +225,34L;123−L²,28L"!

Here is a short program which uses -show- to display a table (see Figure 4-6) of square roots of the integers from 1 to 15:

```
define  N=v1
unit    roots
at      31∅            $$ write titles for the two columns
write   N
at      325
write   N½
do      root,N⇐1,15
*
unit    root
at      41∅+1∅∅N
show    N
at      425+1∅∅N
show    sqrt(N)
```

N	$N^{1/2}$
1	1
2	1.414
3	1.732
4	2
5	2.236
6	2.449
7	2.646
8	2.828
9	3
1∅	3.162
11	3.317
12	3.464
13	3.6∅6
14	3.742
15	3.873

Fig. 4-6.

The last statement could also be written as "show N½". This technique of making tables, including the use of the -do- index (N) to position the displays (as in "at 425+1ØØN") is an important and powerful tool.

There are other commands for displaying variables: -showe- (exponential), -showt- (tabular), -showa- (alphanumeric), -showo- (octal), and -showz- (show trailing zeroes). These are described in detail in the reference material mentioned in Appendix A.

Although -write- is basically designed for non-variable text, combinations of text and variables occur so often that TUTOR makes it easy to "embed" a -show- command within a -write-:

```
write   The area was ⊲s,13.7w,6⊳ square miles.
```

The embedded "s" indicates a -show- command and the remainder "13.7w,6" is its tag. Other permissible abbreviations include "o" (showo), "a" (showa), "e" (showe), "t" (showt) and "z" (showz). The above -write- statement is equivalent to:

```
write   The area was
show    13.7w,6
write   square miles.
```

Passing Arguments to Subroutines

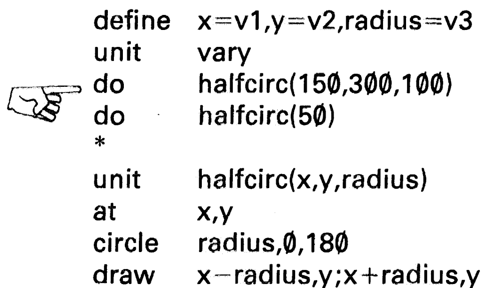
When you write "show 13.7w,6", you are passing two pieces of information to the -show- command. You are giving two numerical "arguments" (13.7w and 6) to the TUTOR machinery that performs the -show- operations. Similarly, we created a half-circular arc with "circle radius,Ø,18Ø" in which we passed three arguments to the TUTOR circle-making machinery. Sometimes certain arguments are optional. For example, "show 13.7w" will use a default second argument of 4 (significant figures), and omitting the last two arguments in a -circle- command ("circle radius") will cause a full circle to be drawn rather than an arc. When we pass one argument to the -at- command ("at 1215"), we mean coarse grid; when we pass two arguments ("at 125,375"), we mean fine grid.

This notion of passing arguments to TUTOR commands, with some arguments optional, also applies to your own subroutines, such as unit "halfcirc". The "halfcirc" subroutine needs three arguments (x, y, and radius) to do its job. We passed these arguments by assigning values to variables and letting "halfcirc" pick up those values and use them:

```
define   x=v1,y=v2,radius=v3
unit     vary
calc     x⇐15Ø
         y⇐3ØØ
         radius⇐1ØØ
do       halfcirc
calc     radius⇐5Ø
do       halfcirc
*
unit     halfcirc
at       x,y
circle   radius,Ø,18Ø
draw     x−radius,y;x+radius,y
```

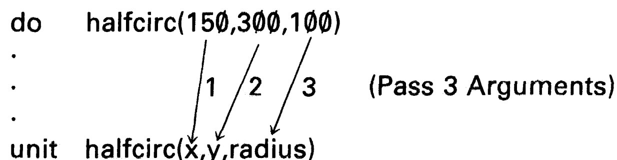
Notice that the second -do- will use the original "x" and "y", since these variables have not been changed. It is as though we passed only one argument ("radius") to the subroutine.

TUTOR permits another way of writing this sequence which looks similar to the way one passes arguments to the "built-in subroutines" (-show-, -circle-, -at-, etc.):



```
define   x=v1,y=v2,radius=v3
unit     vary
do       halfcirc(15Ø,3ØØ,1ØØ)
do       halfcirc(5Ø)
*
unit     halfcirc(x,y,radius)
at       x,y
circle   radius,Ø,18Ø
draw     x−radius,y;x+radius,y
```

The statement "unit halfcirc(x,y,radius)" tells TUTOR that when this unit is done as a subroutine, arguments are to be passed to it. The statement "do halfcirc(15Ø,3ØØ,1ØØ)" tells TUTOR to pass the listed arguments to the "halfcirc" subroutine for its use. The arguments are passed in the order listed:



These variables are now set for use in the subroutine. It is precisely as though we had assigned values to "x", "y", and "radius" by using -calc-. If some arguments are omitted, these variables are not transferred:

```
do      halfcirc(235,,85)
.              |    |
.              |1   |2     (Pass 2 Arguments)
.              ↓    ↓
unit    halfcirc(x,y,radius)
```

In this case the variable "y" has not been assigned a new value, so it retains the value it had, which was 3ØØ. (The value of "y" could have changed if "halfcirc" itself altered it. For example, if we append "calc y⇐75" to the end of unit "halfcirc", "y" would now be 75, although it was originally passed the value of 3ØØ by the first -do- statement during the making of the first display.)

Arguments to be passed need not be simple numbers. Each argument can be a complicated expression. The expressions are evaluated, then passed in order:

```
do      halfcirc(3.4radius−25,radius+25y,2ØØ+y)
.                  1            2          3
.
.
.
unit    halfcirc(x,y,radius)
```

It is as though we had written:

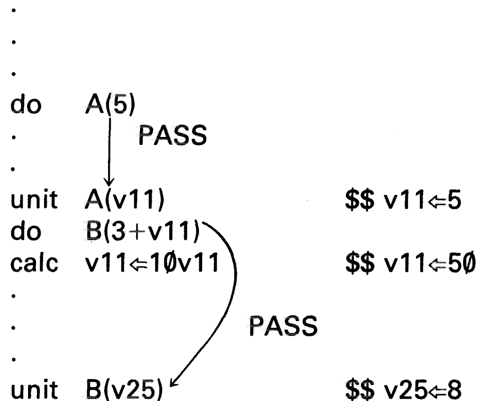
```
calc  arg1⇐3.4radius−25
      arg2⇐radius+25y
      arg3⇐2ØØ+y
      x⇐arg1
      y⇐arg2
      radius⇐arg3
```

Just as the -at- command handles its arguments differently depending on the number of arguments (one for coarse grid and two for fine grid), so it is possible for your subroutines to do such things. There is a TUTOR-defined "system variable" named "args" which always contains the number of arguments passed the last time a subroutine was done. By "system variable" we mean a variable separate from the student variables

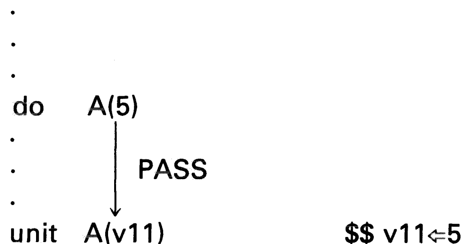
(v1 through v15Ø) whose contents are assigned by TUTOR rather than by you. You do not define system variables; they are already defined for you. (Indeed, if you say "define args=v3", you will override TUTOR's definition of the meaning of "args", so that "args" will mean "v3" rather than "the number of arguments passed to a subroutine".) In Chapter 6 (Conditional Commands) you will see how you could do a variety of things in a subroutine (conditional on the value of "args") which are similar to the kinds of things the -at- command does.

Our subroutine "halfcirc" uses three student variables: v1, v2, and v3, defined as "x", "y", and "radius". Another subroutine could use the same variables for carrying out its work, but it must be kept in mind that -do-ing this subroutine will affect v1, v2, and v3, since arguments will be passed.

Suppose one subroutine uses another, with "nested" -do-s like this:



Variable v11 ends up with the value 5Ø. It is advisable to use different variables in the two subroutines. Here unit A uses v11 and unit B uses v25. It can lead to confusion or even logical errors if B also uses v11 to do its work, since -do-ing B will affect the value of v11 used by A. Here is the structure to be avoided:



```
do    B(3+v11)  \
calc  v11⇐10v11  )         $$ v11⇐80
.               /
.              /  PASS
.             /
unit  B(v11) <             $$ v11⇐8
```

Now variable v11 ends up with the value 8∅ rather than 5∅. This is due to the effect on v11 of the "do B(3+v11)" statement, which assigns the value of 8 to v11 by passing the argument to unit "B".

This concludes our discussion of calculations for now. We can calculate, save results, use them to make displays, and show the values. In the next section, we will use calculations in association with guiding the sequencing of a lesson.