Conditional Commands 6

It is important to be able to specify the sequencing of a lesson *conditionally.* We might like to jump past some material on the condition that the student has demonstrated mastery of the concept and needs no further practice. Or we might like to take the student to a remedial sequence conditionally (the condition being poor performance on the present topic). Or, which help sequence we offer might be conditional on the number of times help has been requested. All of these examples imply a need for *conditional* sequencing or branching statements, where the condition may be specified by calculations involving the status of the student.

The usefulness of conditional branching is not limited to the sequencing of major lesson segments, but extends to many calculational or display situations. For example, we might need to -do- conditionally one of several possible subroutines in the course of presenting a complex display to the student. This chapter will show you how to perform these and similar conditional operations.

Here is an example involving a *conditional* -do- statement:

```
unit    setup
calc    N⇐-1
jump    home
*
```

(Continued on the next page.)

```
unit     home
next     home
at       2Ø1Ø
do       N,neg,uzero,One,utwo
at       1215
write    N equals ⊲s,N⊳.
calc     N⇐N+1
*
unit     neg
write    Unit "neg".
*
unit     uzero
draw     21Ø,26Ø;2Ø6Ø;2Ø1Ø
*
unit     One
circleb  5Ø,Ø,27Ø
*
unit     utwo
write    Unit "two".
```

The new element is the *conditional* -do- statement in unit "home". If N is negative, that statement is equivalent to "do neg". If N is zero, the statement is equivalent to "do uzero", and so on. The statement:

```
do   N,neg,uzero,One,utwo
```

is equivalent to:

```
do   neg     if N is negative
do   uzero   if N is zero
do   One     if N is 1
do   utwo    if N is 2 or greater
```

Note that unit "utwo" will come up repeatedly because it is the last unit named in the conditional -do- statement. The list of unit names can be up to 1ØØ long:

```
do   N,neg,uzero,One,utwo,dispone,
     zon,zip,figure,ultima
```

If N is 7 or greater, this statement is equivalent to "do ultima".

The "conditional expression" (N in this case) can be anything. It can be as complicated as "3x − 5 sqrt(N)" and can even involve assignments as in "N⇐35−x". The value of the expression is *rounded to the nearest integer* before choosing a unit from the list of units. If the *rounded* value is negative, the *first* unit in the list is chosen. For example, if the expression is −.4, it rounds to *zero,* in which case the *second* unit in the list is chosen.

In a conditional -do- each unit named may involve the passing of arguments:

```
do   3N−4,circ(25,75),box(45),x,flag,circ(1Ø,3Ø)
             neg        Ø     1  2     ≥3
```

So far we have encountered the following sequencing commands: -do-, -jump-, -next-, -next1-, -back-, -back1-, -help-, -help1-, -lab-, -lab1-, -nextnow-, -data-, -data1-, and -base-. When the tag of such a command is just a single unit name (e.g., in a statement like "help   uhelper"), we say it is "unconditional". To make a "conditional" statement out of any of these, we follow the same rule: state the conditional expression, followed by a list of unit names. So we might have:

```
data   N−5,zonk,q,zap,zing,x
expression negative zero one two three or greater
```

Here, as in unconditional pointer-associated statements, "q" means the "data" pointer is cleared so that the DATA key is disabled. This can be used to cancel the effect of an earlier -data- command in this main unit. (Remember that all the unit pointers are cleared when we start a new main unit.) The unit name "x" has the special meaning "don't do anything!" In the example shown, if the condition (N−5) is three or greater, this -data- command has no effect at all and we "fall through" to the next statement without affecting the "data" pointer. Similarly, if a unit name in the conditional -do- discussed above is replaced by "x", no unit will be done for the corresponding condition and we "fall through" to the next statement.

This "x" option is extremely useful. Consider the following situation:

```
jump   correct−5,x,done
(then show the next item)
```

If (correct−5) is negative (that is, the student has made fewer than 5 correct answers), we "fall through" to the presentation of the next item. If, however, the student has 5 or more correct, the condition (correct−5) will be zero or greater and we jump to unit "done".

Logical Expressions

The last example can be written in an alternative form which improves the readability:

```
jump   correct<5,x,done
```

This says "fall through if correct is less than 5, otherwise jump to done". The condition (correct<5) we call a "logical expression" because it has only two possible values, "true" (−1) or "false" (∅), whereas numerical expressions can have any numerical value. Since a logical expression can have only two values (−1 if true, or ∅ if false) it is pointless to list more than two unit names after the condition.

Actually, because of rounding, the form "jump   N<5,x,done" is more precise than the form "jump   N−5,x,done". Suppose that N is 4.8. Then "N<5" is true (−1), which rounds to −1, which implies "x". But "N−5" is −∅.2, which rounds to zero, which implies "done". Such differences appear whenever you have variables which can have non-integer values.

Here is another example:

```
do   c−b,far,near,far
```

The above will do unit "near" if c and b differ by no more than ∅.5, since (in that case) "c−b" will lie between −∅.5 and +∅.5, which rounds to zero. On the other hand:

```
do   c=b,same,diff
```

will do unit "same" only if c and b are *equal.* The condition "c=b" is true (−1) only if c is equal to b.

There are six basic logical operators: =, ≠, <, >, ≤, and ≥, which mean equal, not equal, less than, greater than, less than or equal, and greater than or equal. The statement "do a≠b,diff,same" is equivalent to "do a=b,same,diff". These comparison operators consider two numbers to be equal if they differ by less than one part in $1Ø^{11}$ (relative tolerance) or by an absolute difference of $1Ø^{-9}$, whichever is larger. This is done to compensate for small roundoff errors, inherent to computers, due to their very high but not infinite precision. One consequence is that all numbers within $1Ø^{-9}$ of zero are considered equal by these logical operators. If it is necessary to test very small numbers, scale up the numbers: 1ØØØa<1ØØØb can be used if a and b are larger than $1Ø^{-12}$ (since multiplying by 1ØØØ brings the quantities up above the $1Ø^{-9}$ threshold).

You can mix logical expressions with numerical expressions in many effective ways. For example:

```
calc  x⇐1ØØ−25(y>13)
```

gives "x⇐125" if y is greater than 13 ("y>13" if true is −1) or it gives "x⇐1ØØ" if y is less than or equal to 13 ("y>13" if false is Ø). To clarify this, suppose that y is 18 or y is 4:

```
y=18                y=4
1ØØ−25(y>13)        1ØØ−25(y>13)
1ØØ−25(18>13)       1ØØ−25(4>13)
1ØØ−25(−1)          1ØØ−25(Ø)
1ØØ+25              1ØØ−(Ø)
125                 1ØØ
```

In these applications it would be nice if "true" were +1 rather than −1, but the much more common use of logical expressions in conditional branching commands dictates the choice of −1 (since the first unit listed is chosen if the condition is negative).

You can combine logical expressions. For example:

```
[(3<b)  $and$  (b<5)]
```

is true (−1) only if *both* conditions (3<b) and (b<5) are true. In other words, b must lie between 3 and 5 for this expression to have the value −1. Similarly,

```
(y>x)  $or$  (b=2)
```

will be true if *either* (y>x) is true *or* (b=2) is true (or both are true).

Finally, you can "invert" the truth of an expression:

```
not(b=3c)
```

is true if (b=3c) is *not* true. This complete expression is equivalent to "b≠3c".

The combining operations and, or, and "not" make sense only when used in association with logical expressions (which are −1 or ∅). For instance, [b>c and 19] is meaningless and will give unpredictable results. (If you have done a great deal of programming, you might wonder about special bit manipulations, but there are separate operators for masking, union, and shift operations, as discussed in Chapter 10.)

The Conditional -write- Command (-writec-)

A very common situation is that of needing to write one of several possible messages on the screen. For example, you might like to pick one of five congratulatory messages to write after receiving a correct response from the student:

```
unit    congrat
randu   N,5              $$ let TUTOR pick an integer from 1 to 5
at      1215
do      N-2,ok1,ok2,ok3,ok4,ok5
*
unit    ok1
write   Good!
*
unit    ok2
write   Excellent!
*
unit    ok3
write   I'm proud of you.
*
unit    ok4
write   Hurray!
*
unit    ok5
write   Great!
```

The -randu- command, "random on a uniform distribution," tells TUTOR to pick an integer between 1 and 5 and put it in N. We then use this value of N to do one of five units to write one of five messages. There is a *much* more compact way of writing this:

```
unit    congrat
randu   N,5
at      1215
writec  N−2,Good!,Excellent!,
        I'm proud of you.,
        Hurray!,Great!,
```

The -writec- command is similar to that of a conditional branching command, but the listed elements are pieces of text rather than unit names. Because -write- can be used to display any kind of text (including commas), it is necessary to use a different command name (-writec-) to indicate the conditional form of -write-, whereas in branching statements the commas separating the unit names are enough to tell TUTOR that it is a conditional rather than an unconditional form. (In conversation, "writec" is pronounced "write-see.")

You can write whole paragraphs with nice left margins, just as with the -write- command:

```
writec  N,,,Good!,Excellent!,
        I'm proud of
        you and so
        is your mother.,
        Hurray!,Great!,
```

The elements of text are set off by commas. If N is 3, the student will see a three-line paragraph, since there are no commas at the end of "of" and "so". If N is −1 or Ø, no text will be displayed, since there is no text between the first few commas. Note that "x" is not the fall-through that it is for a unit name in a conditional branching command. Here, "x" is a legitimate piece of text which can be displayed, so the ",," form is the "fall-through".

If you want commas to appear in some of your text elements, you have a problem, since the commas delimit elements. Consider this:

```
writec  N,Hello!,How are you, Bill?,Hi there!,
```

If N is zero, we will see "How are you", not "How are you, Bill?" The solution is to use a special character (↕):

```
writec  N↕Hello!↕How are you, Bill?↕Hi there!↕
```

Now, if N=Ø we will see "How are you, Bill?" While this special character (↕) is required if text elements contain commas, you may prefer to use it always, even when there are no commas. This special character is often called "the writec delimiter".

The same kinds of embedding of other commands which are permitted by -write- are also permitted with -writec-:

```
writec  2c=b,I have ⊲s,ap⊳ apples.,
        I will buy ⊲s,peachy⊳ peaches.,
```

The -writec- is affected by -size- and -rotate- commands, just like -write-.

The Conditional -calc- Commands: -calcc- and -calcs-

The effects of -writec- can be achieved by a conditional -do- and a bunch of units containing the text elements, but we have seen that this is a clumsy way to do it. We would often like to calculate one of several things based on a condition. This, too, could be done with a conditional -do- to one of several units containing the calculations, but this is cumbersome. We saw one shortcut already:

```
calc   x⇐1ØØ−25(y>13)
```

This statement is equivalent to "x⇐125" if y>13, and to "x⇐1ØØ" if y≤13. This can also be written as:

```
calcc   y>13,x⇐125,x⇐1ØØ
```

The -calcc- (pronounced "calc-see") is strictly analogous to -writec-. It indicates a list of calculations to be performed, dependent on a condition. The elements in the list are calculations rather than pieces of text or unit names.

Very often each of the calculations in the list consists of assigning a value to the *same* variable. In the example above, both calculations assign a value to the variable "x". An even shorter way to write this kind of thing is:

calcs   N−5y,bin⇐37,5.2,y^3+2,,2/N

The -calcs- (pronounced "calc-ess") will store one of five values in "bin", depending on the condition "N−5y". Note that if "N−5y" rounds to two, we do nothing. Two commas in a row (,,) indicate "do nothing" in -calcs-, -calcc-, and -writec-. Just as "x" can be a legitimate piece of text to write, so "x" might be a defined variable, which is why it cannot be used as the "do-nothing" indicator in these commands.

The Conditional -mode- Command

For completeness it should be mentioned that the -mode- command can also be made conditional:

```
mode   count−3,write,x,rewrite,erase,write
```

Here the list of elements following the condition is similar to the list of unit names in a -help- command. In this case, they are the names of the various possible screen display modes. The "x" option means "do nothing—do not change the present mode."

The -goto- Command

The -goto- command is a very mild version of the -jump- command. It does not initiate a new main unit and does not perform the initializations associated with starting a main unit (the screen is not erased, the help and other unit pointers are not cleared, and how deep we are in "do" levels is unaffected). It is most often used in its conditional form so we waited until this chapter to introduce it.

One common use of the -goto- command is to "cutoff" a unit prematurely:

```
unit   A
at     1315
write  You have now finished the quiz.
goto   score<9Ø,fair,x
size   4
at     22Ø5
write  Congratulations!
size   Ø
*
unit   B
at     1912
write  The next topic is . . . . .
.
.
.
unit   fair
at     1815
write  Your score was below 9Ø.
*
unit   blah
.
.
.
```

In this example, a score of 9∅ or better will mean that we fall through the -goto- to display the large-size "Congratulations!" A score of less than 9∅ will take us to unit "fair" to add "Your score was below 9∅" to the "You have finished the quiz" already on the screen. The -goto- does not erase the screen, nor does it change the fact that the main unit is still "A". When the student presses NEXT, he proceeds to unit "B", the main unit following unit "A". He does *not* proceed to unit "blah".

Like -do-, the -goto- command attaches a unit without changing which unit is "home", whereas -jump- changes the main unit and performs the many initializations associated with entering a new main unit (full-screen erase, clearing the help pointers, forgetting any -do-s, etc.). The main difference between -goto- and -do-, is that the -do- will normally come back upon completion of the attached unit, whereas -goto- does not come back and statements following the -goto- are normally nøt executed. (Some people like to think of the -goto- coming back to the *end* of the unit, whereas -do- comes back to the next statement.)

The relationships among main units and attached units and among -jump-, -goto-, and -do- may be clearer if you think of a lesson as being made up of a number of nodes or clusters, each consisting of a main unit and its attached units:

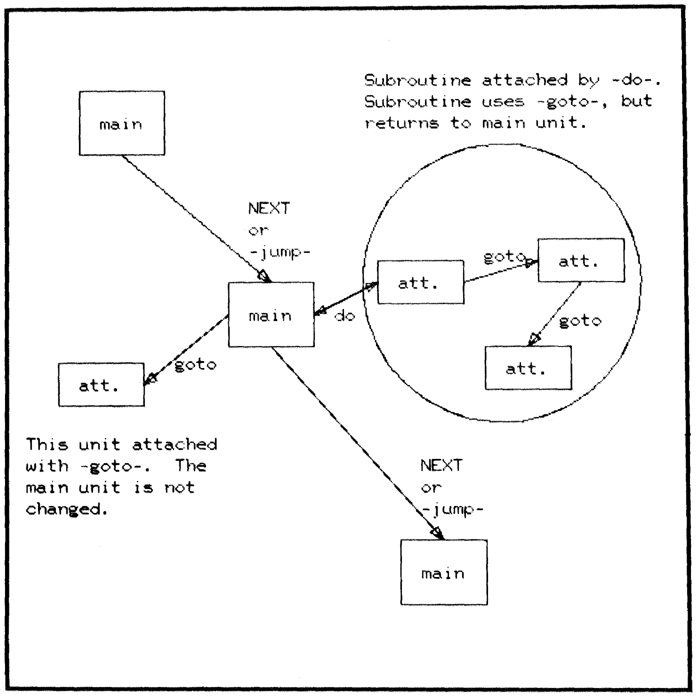

Fig. 6-1.

Movement between main units is made by pressing NEXT (or HELP, BACK, etc.) or by executing a -jump-. These main units may form a normal sequence or a help sequence (see Chapter 5). The -goto- and -do- commands attach auxiliary units to these main units.

Notice that completion of a unit reached by one or more -goto-s will cause TUTOR to "undo" one level, if one or more -do-s had intervened in reaching this unit. The reason this occurs is that whenever TUTOR encounters a -unit- command (which terminates the preceding unit) TUTOR asks "Are we at the main-unit level?" If so, we have completed processing; if not, we must "undo" to the statement immediately following the last -do- encountered. This point deserves an illustration:

```
unit    calcit
do      sum
show    total
.
.
.
unit    sum
calc    total⇐0   $$ initialize "total"
goto    addup     $$ -goto- used instead of -do-, for
*                 $$ purposes of illustration
unit    addup
.
.                 $$ a calculation of "total"
.
unit    other
```

In unit "calcit" we -do- "sum", which initializes "total" and does a -goto- to unit "addup", where some kind of calculation is performed. When we run out of work (by encountering a -unit- command at the end of unit "addup"), TUTOR asks whether there was a -do-. There *was* a -do-, so control passes to the statement following the last -do-, which is "show total". All of this is perfectly reasonable and useful, but it should be pointed out that this property of the -goto- (that it preserves the required information to permit "undoing") has an odd side-effect. The presence of a -goto- in a done unit causes an exception (the *only* exception) to the description of -do- as a text-insertion device. Except for this case, the effect of a -do- is equivalent to inserting all the statements, contained in the done unit, in place of the -do- statement. But suppose we replace our -do- with the statements contained in unit "sum". We would have:

```
unit    calcit
calc    total⇐0 }
goto    addup   } in place of "do   sum"
show    total
*
unit    addup
.
.
unit    other
```

Now the -goto- cuts off the rest of unit "calcit", and the -show- will not be performed, in contrast with the case where we used a -do-. So, the presence of a -goto- in a done unit causes a (useful) exception to the text-insertion nature of -do-.

Here is a summary of the basic properties of the -goto- command:

1) -goto- may be used to attach units with none of the initializations associated with -jump-;
2) statements which follow the -goto- will not be executed (like -jump- and unlike -do-);
3) a -goto- in a done unit does *not* cut off statements following the original -do- statement, which is an exception to the normal text-insertion nature of -do-.

Additional aspects of -goto- (in relation to judging student responses) are discussed in Chapter 8.

It is often convenient to cut off a unit with a -goto- in the form shown in this example:

```
unit    cuts
goto    expression,x,zonk,empty,x,empty
write   We fell through . . .
.
.
.
unit    empty
*
unit    zonk
.
.
.
```

Note that unit "empty" has nothing in it but serves merely to have a place to go to in order to cut off the end of unit "cuts". This is such a common situation that TUTOR provides an empty unit named "q" (for quit). The previous -goto- can be written as:

```
goto    expression,x,zonk,q,x,q
```

The statement "goto q" means go to an empty unit. The special meaning of "q" here makes it illegal to have your own unit named "q", just as it is not possible to name a unit "x". Since "do empty" can be rendered by the equivalent "do x", the statement "do q" (or a conditional form) is given the special interpretation of acting like a "goto q". The use of "q" in a -goto- statement is somewhat different from the use of "q" in a -help- statement. You will recall from Chapter 5 that "help q" means to quit specifying a help unit, by clearing the -help- pointer.

The -goto- can be used in association with the -entry- command to skip over statements:

```
        .
        .
        .
        calc    b⇐Ø
        goto    3f>5,leavit,x
        calc    b⇐f/2
                f⇐Ø
        entry   leavit
        .
        .
        .
```

If 3f is greater than 5, we skip over intervening statements to entry "leavit". The -entry- command is equivalent to a special -goto- plus a -unit-:

```
  .
  .
  .
 { special goto   leavit }
 { unit           leavit } equivalent to (entry leavit)
  .
  .
  .
```

So, unlike a -unit- command, -entry- does not terminate a unit but merely provides a named place to branch to. Its equivalence to a special hidden -goto- followed by a -unit- command means that an entry is completely equivalent to a unit, except for not terminating the preceding statements. For this reason it is possible to use an entry name with -do-, -jump-, -help-, etc.

The conditional -goto- is often used for repetitive operations similar to those carried out with -do-. Here are two versions of a subroutine to add the cubes of the first ten integers:

```
-do-                          -goto-
unit  add                     unit   add
calc  total⇐Ø                 calc   i⇐1
do    add2,i⇐1,1Ø                    total⇐Ø
*                             goto   add2
unit  add2                    *
calc  total⇐total+i³          unit   add2
                              calc   total⇐total+i³
                                     i⇐i+1
                              goto   i≤1Ø,add2,x
```

The last two statements in the -goto- example could be combined as:

```
goto   (i⇐i+1)≤1Ø,add2,x
```

For the simple task of adding ten numbers, the -do- form is certainly easier to construct, but situations occasionally arise where it is easier to construct a repetitive loop using a conditional -goto-.

Except for not changing how many levels deep in -do-s we are, -goto- is quite similar to -do-. Although the feature is seldom used, it is even possible to pass arguments to a subroutine with a -goto-:

```
goto   zonk(12,25)
```

Arguments may also be passed in a conditional -goto-:

```
goto   3N−4,alpha(2+count),x,beta(15,2N),q
```

The Conditional Iterative -do-

The conditional and iterative -do- can be combined so that, on each iteration, the conditional expression selects which unit to do this time:

```
do   N+3,ua,ub,uc,ud,i⇐1,12
          ↓   \   \   \
         neg   Ø   1   ≥2
```

For each value of i (from 1 to 12), the expression "N+3" is evaluated, which determines which subroutine will be done. For example, if "N+3" is ∅, the above statement is equivalent to "do ub,i⇐1,12". Usually a conditional iterative -do- is used in situations where the conditional expression ("N+3") is not changing, but doing one of the subroutines *can* change N so that a different subroutine is used on the next iteration. The following is an example of such manipulations:

```
do   i–2,ua,ub,uc,ud,i⇐1,4
```

In the first case, where i is equal to 1, the condition "i–2" is –1, so we do "ua". Then i is incremented to 2, and we do "ub", etc. This is, therefore, equivalent to the sequence:

```
do  ua
do  ub
do  uc
do  ud
```

As usual, the specified units can involve the passing of arguments.

In a conditional non-iterative -do- the unit names "x" and "q" mean "don't do anything" and "goto q" respectively. In a conditional iterative -do-, "x" means "don't do anything on this iteration," and "q" means "quit doing this statement and go on to the next statement." In other words, "x" means "fall through to the next iteration," while "q" means "fall through to the next TUTOR statement." For example:

```
do      i–2,ua,x,q,ud,i⇐1,4
show    i
```

will display the number "3". For i equal to 1 we do "ua"; for i equal to 2 we do nothing; for i equal to 3 we quit and go on to the following -show- statement.

The -if- and -else- Commands

Suppose you want to do one set of statements if x is greater than y, and a different set of statements. One way to do this, as we have seen, is to put the two sets of statements in two different units and write "do x>y, unita, unitb". Another way to perform these operations is to use -if- and -else- commands:

```
              if      x>y
Done if x>y { .       calc    Z⇐5y
            { .       draw    x,Z;x+1ØØ,Z+1ØØ
              else
Done if x≤y { .       at      x,y
            { .       circle  5Ø
              endif
```

The statements between the -if- and -else- commands are performed only if x is greater than y, and the statements between the -else- and -endif- commands are performed otherwise. The tag of the -if- command must be a logical expression (one that has values −1 or Ø). The tag of the -else- command must be blank. The -endif- command identifies the end of the sequence.

Note that the statements bracketed by -if-, -else-, and -endif- must be indented, with an initial period identifying them as indented statements. (It is possible that the details of this indenting format may change. Consult on-line PLATO aids for up-to-date information.)

When do you use a conditional -do-, and when do you use -if- and -else-? This depends mainly on the number of statements involved. If there are few statements to be performed, -if- and -else- is probably more readable. But, if "unita" and "unitb" are long subroutines, the conditional -do- is the more convenient form.

There doesn't have to be an -else-:

```
if      x>y
.       calc    Z⇐5y
.       draw    x,Z;x+1ØØ,Z+1ØØ
endif
```

This will do the -calc- and -draw- only if x is greater than y.

There is also an -elseif- for specifying an additional condition:

```
              if      x>y
Done if x>y { .       calc          Z⇐5y
            { .       draw          x,Z;x+1ØØ,Z+1ØØ
```

```
                    elseif  x>.5y
Done if x>.5y     { .       at      1225
but x not         { .       write   This paragraph will be
greater than      { .               displayed only if x is
y                 { .               not greater than y but
                  { .               is greater than .5y.
Done if             else
neither of        { .       at      1225
the above         { .       write   x is less than .5y!
is valid            endif
```

It is possible to have additional levels of indented -if- structures:

```
                    if      a=b     $or$  b>3
                    .       calc    x⇐b+2
                    .       if      count<8
A second          { .       .       at     2513
level of          { .       .       write  Two levels!
indenting         { .       else
                  { .       .       do     subr
                    .       endif
                    else
                    .       at      912
                    .       show    x
                    endif
```

The text "Two levels!" will appear on the screen if (a=b or b>3) and if (count<8).