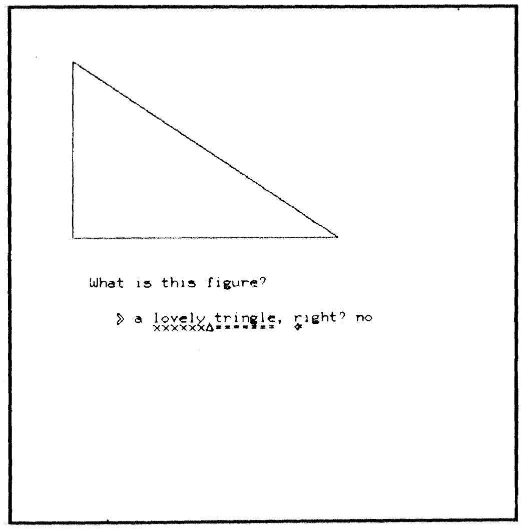
Judging Student Responses 7

You now know quite a bit about how to express (in the TUTOR language) your instructions to PLATO on how to administer a lesson to a student. You may not have realized it, but in the process you have learned a great deal about the fundamental concepts of computer programming. You can calculate, produce complex displays, and construct rich branching structures. You have studied aspects of initialization problems, you have seen the importance of subroutines, and you have looked at some stylistic aspects of good programming practice such as defining variables, placing unit pointer commands at the head of main units, etc. With this solid background you are now ready for a detailed look at how to accept and judge student responses.

In Chapter 1 you saw a common type of judging situation in which you simply listed the anticipated responses after an -arrow- statement, together with the display or other actions to be performed depending on the particular response. Let us see how TUTOR actually processes these judging commands. We will consider a slightly different version of the "geometry" unit. Remember that in the -answer- and -wrong- statements, parentheses enclose synonyms, and angle brackets enclose ignorable words.

```
unit     geometry
draw     51Ø;151Ø;154Ø;51Ø
arrow    2Ø15
at       1812
write    What is this figure?
answer   <it,is,a> (right,rt) triangle
write    Exactly right!
wrong    <it,is,a> square
write    Count the sides!
```



```
What is this figure?

  > a lovely tringle, right? no
      xxxxxxΔ======== ≡≡≡    ◆
```

Fig. 7-1.

The order of the initial statements has been changed slightly. TUTOR starts executing this main unit by drawing the triangle. TUTOR next encounters the -arrow- command, places an arrowhead at position 2Ø15, and *notes where this -arrow- command is* (the second command in unit "geometry"). TUTOR then executes the -at- and -write- to display the text: "What is this figure?"

Finally, TUTOR reaches the -answer- command. This "judging" command is *useless* at this time because the student *has not entered a response.* There is nothing more that can be done but wait for the student to type a response and enter it by pressing NEXT. We call commands which operate on the student's response "judging" commands (such as -answer- and -wrong-). Other commands, such as -draw-, -at-, -write-, and -calc-, are called "regular" commands. We see that TUTOR must stop executing regular commands when a judging command is encountered. (This assumes the presence of an -arrow- command. An -answer- or other judging command without a preceding -arrow- is meaningless.)

When the student presses NEXT to enter his or her response, TUTOR looks at its notes and finds that the -arrow- was the second command in unit "geometry". TUTOR starts looking just beyond there for judging commands to process the student's response. It *skips* the regular commands -at- and -write- since these are not judging commands and are of no use at this point. It encounters the -answer- command and compares the student response with the specifications given in the tag of the -answer- command.

If there is not an adequate match, TUTOR goes to the next command looking for a judging command that might yield a match. In this case, the following command is a regular command (-write-) which is skipped. Next there is a -wrong- judging command, and if there is no match to the student's response, TUTOR keeps judging. At this point, there is a -write- regular command which is skipped.

Finally, we come to the end of the unit without finding a matching judging command and must give a "no" judgment to this response (and possibly mark up the response with underlining and X's if the response is fairly close to that specified by the -answer- command). (See Figure 7-1.) The process of starting immediately after the -arrow- in the "judging state" will be repeated each time the student tries again with a revised response.

If, on the other hand, the response adequately matches the -answer- statement, TUTOR has found a match and can terminate the execution of judging commands. It switches to processing regular commands with the result that the following "write Exactly right!" will be executed. (This regular command is skipped unless a match to the -answer- flips TUTOR out of the "judging state" into the "regular state".) Then TUTOR, *in the regular state,* comes to a judging command (-wrong-) which terminates the processing. TUTOR finishes up by placing an "ok" beside the student response. (Similarly, a match to the -wrong- would flip TUTOR to the regular state to execute the regular statement "write Count the sides!")

When the -arrow- is finally "satisfied" by an "ok" judgment, TUTOR returns one last time to the -arrow- and searches for any other -arrow- commands in the unit. In this search it skips both regular and judging commands. In our particular example no other -arrow- is found, so all arrows (one) in the unit have been satisfied. After the student has read our comment, he or she presses NEXT and proceeds to the next main unit.

It may seem wasteful to you that TUTOR keeps going back to the -arrow- only to skip over the regular commands preceding the first judging command. It turns out that skipping a command is an extremely fast procedure, and that keeping a single marker (the location of the -arrow- command within the unit) greatly simplifies the TUTOR machinery.

In the example, the replies "Exactly right!" or "Count the sides!" would be displayed at location 2317, three lines below the response on the screen. This standard positioning can, of course, be altered by an -at- statement. Here is another illustration:

```
unit     canine
at       21Ø5
write    Name a canine:
```

(Continued on the next page.)

```
arrow    23Ø8
answer   dog
write    A house pet.
answer   wolf
write    A wild one!
wrong    cat
write    A feline!
```

Suppose the student enters "wolf" as his response. TUTOR initiates the "judging state" just after the -arrow-. The first -answer- (dog) does not match, so TUTOR stays in the judging state and skips the "write A house pet." There is a match to the following "answer wolf", so judging terminates and the regular state starts. The "write A wild one!" is executed, not skipped. Next, TUTOR encounters a "wrong cat", and since -wrong- is a judging command, this terminates the regular state. The student gets an "ok" judgment. TUTOR searches for another -arrow- but does not find one, so the student has successfully completed the unit. (Various units of this kind are illustrated with animated diagrams in the on-line "aids" available on PLATO.)

This method of processing judging and regular commands yields a readable programming structure, with judging commands delimiting the regular commands used to respond to the student. We have spent time discussing the details in order to simplify our later descriptions of the various types of judging commands used to match, modify, or store student responses.

It is important to point out that the -do- and -goto- commands are *regular* commands. They are, therefore, skipped over during the judging state and during the search state (looking for a possible additional -arrow- after an arrow has been satisfied). There is another command, -join-, which works much like -do- except that the -join- command is universally executed whether TUTOR is in the regular state, the judging state, or the search state. In particular, it is possible to -join- units containing judging commands, whereas a -goto- or -do- is incapable of accessing other units in the judging state (since these regular commands are skipped). Although the -do- command acts essentially like a -join-, it is, nevertheless, a regular command and is skipped during the judging and search states. Only the -join- command itself has the unique characteristic of being performed in all states (regular, judging, and search).

It is frequently useful to handle more than one response in a unit. Let's ask "Who owned Mount Vernon?" and (after receiving a correct response) ask in what state it is located *but stay on the same page:*

```
unit     wash
at       812
```

```
  write    Who lived at Mount Vernon?
  arrow    1Ø15
⎧ answer   <George,G> Washington
│ at       112Ø
⎨ write    Great!
│ wrong    Jefferson
│ at       1112
⎩ write    No, he lived at Monticello.
  arrow    1715
⎧ at       1512
⎨ write    In what state is it located?
⎩ answer   (Va,Virginia)
```

If you say "Jefferson" the -wrong- is matched. Regular commands are executed until you run into the second -arrow-, which ends the range of the first -arrow-. In other words, when you are working on one -arrow-, the next -arrow- is a terminating marker. If you say "Washington", the student gets the "Great!" comment. Since the -arrow- is now satisfied, TUTOR starts at the first -arrow- searching for another -arrow-. In this search state, all commands other than -join- are skipped (-join- may be used to attach a unit that contains another -arrow-). A second -arrow- is encountered, which changes the search state into the regular state. The arrowhead is displayed on the screen and the location of this -arrow- within the unit is noted. The regular commands following this second -arrow- are processed to display the second question. The final -answer- command stops this processing to await the student's response.

There is another way to do this which is probably more readable:

```
    unit      wash
    next      wash
    at        812
    write     Who lived at Mount Vernon?
    arrow     1Ø15
  ⎧ answer    <George,G> Washington
  │ at        112Ø
  ⎨ write     Great!
  │ wrong     Jefferson
  │ at        1112
  ⎩ write     No, he lived at Monticello.
☞   endarrow
    at        1512
    write     In what state is it located?
    arrow     1715
  { answer    (Va,Virginia)
```

The -endarrow- command defines the end of commands associated with the first -arrow-. Note that -endarrow- changes the search state to the regular state. One benefit of this form is that the second arrowhead appears on the screen *after* the text of the second question, which often seems more natural.

It may seem rather abrupt that the "Great!" and "In what state is it located?" both appear on the screen at the same time. It might be better to let the student digest the reply before presenting the second question. We might insert a -pause- (with the tag "keys=all") just after the -endarrow-. Now TUTOR waits for you to press a key, (which signals that you want to go on) before presenting the next question.

The -endarrow- command is quite useful even in units which contain only one -arrow-:

```
.
.
.
arrow      1213
answer     dog
write      Bowwow!
answer     wolf
write      Howl!
wrong      cat
write      Meow.
endarrow
calc       y⇐37+y
circle     1ØØ,25Ø,25Ø
```

The commands following the -endarrow- will be executed only after the -arrow- is satisfied, whether it be by the response "dog" or "wolf". So this is a convenient way to finish up the unit.

While it is possible to -join- or -do- units which contain -arrow- commands, two seemingly arbitrary rules *must* be followed or you will get unpredictable results:

1) A unit attached by -join- or -do- which contains one or more -arrow- commands *must* end with an -endarrow- command (possibly followed by regular commands).
2) This attached unit must not contain any -goto- commands.

If you violate either of these rules, strange things will happen because TUTOR may "undo" from this unit several times (during judging, while processing regular commands, or in the search state).

If you follow these two rules, the -join- or -do- will act like a text-insertion device whereby your program will act as though you had inserted the attached unit where the -join- or -do- was. We will discuss these rules in more detail in Chapter 8.

Student Specification of Numerical Parameters

The -answer- and -wrong- commands make it easy to specify a list of anticipated responses each of which (due to the specification of synonymous and optional words) can allow the student considerable latitude in the way he or she phrases his or her response. However, in some cases there can be no list of anticipated responses and a different technique must be used. For example, you might ask the student to specify a rocket's launch velocity and use his or her number to calculate and display the rocket's orbit. Or you might ask the student for his or her name for later use in personalized messages such as "Bill, you should look at Chapter 5." In such cases, all you can anticipate is that the response will be a number or a name, but you can't possibly list all possible numbers or names.

Here is an example of such a situation. We will provide the student with a desk calculator accessible on the DATA key. In the desk calculator mode the student can type complicated expressions (such as "$2+6^3$") and receive the evaluated result. (Students also have access to a similar calculator mode by typing "TERM-calc", which is a built-in PLATO feature.)

```
unit    mainline
data    desk
at      3Ø2Ø
write   Press DATA for calculator
.
.
.
unit    desk
next    desk        $$ for repeated use
at      1713
write   Type an expression.
        Press BACK when finished.
arrow   1915
store   eval        $$ Be sure to define "eval".
ok                  $$ Accept all responses.
write   The result is ⊲s,eval⊳.
```

The -store- command will evaluate the student's expression (e.g., "13sin3Ø°") and store the result in "eval" (in this case, the number 6.5). The -store- command is a judging command because it operates on the student's response and can be executed only after the student initiates judging by pressing NEXT. The -ok- command is a universal -answer- which matches all responses, and unconditionally flips TUTOR from the judging state to the regular state. In this example, it accepts any response and enables the following -write- to display the evaluated result.

Note that a student need not use parentheses with functions. For example, sqrt25, cos6Ø°, arctan3 are all legal. However, such expressions are illegal in a -calc-. In a moment we'll see another way in which TUTOR is more tolerant of students than of authors.

What if the response cannot be evaluated, such as "$(-3)^{1/2}$" or "19/" or "(3+5)))"? In this case, the student will get a "no" judgment. To see how this works, let's insert a -write- statement after the -store-:

```
.
.
.
store   eval
write   Cannot evaluate!
ok
.
.
.
```

Notice that this new -write- is normally skipped because the -store- leaves us in the judging state. But, if the student's expression cannot be evaluated, -store- makes a "no" judgment and *switches* us from the judging state to the regular state. TUTOR then executes the "write Cannot evaluate!", after which it encounters a judging command (-ok-) which stops the regular processing. Note that -store- terminates judging only on an error condition, whereas -answer- terminates judging only on a match, and -ok- *always* terminates judging.

You can tell the student precisely (in a -writec- statement) what is wrong with his or her expression by use of the *system* variable "formok". This variable is −1 if the student's expression can be evaluated but takes one of several positive integral values for specific errors such as unbalanced parentheses, bad form, unrecognized variable name, etc. The variable "formok" is defined automatically to perform this function. (If you yourself define "formok=v3" you override the system definition and you won't get these features.) The particular values assumed by "formok" can be obtained through on-line documentation at a PLATO terminal.

You can also give the student some storage variables. Let's define a couple of variables for the student:

```
define  student        $$ special define set
        bob=v3Ø,cat=v31
```

Place these defines *ahead of everything else* in the lesson. Suppose you do a -calc- to assign bob⇐18 and cat⇐3. If the student types "2bob" he gets 36. Or he can type "bobcat" and get 54, whereas bobcat would be illegal in a -calc- where you would need bob×cat or bob(cat). Only names defined in the set of definitions labeled "student" may be used by the student in this way. Attempted use (by the student) of names in your other sets of defines will give a value of "formok" corresponding to "unrecognized variable name".

We have discussed a desk calculator, but clearly the store/ok combination will work in any situation where we let the student choose a number. Another good example is in an index of chapter numbers:

```
unit    table
base
term    index  $$ or access by means of shift-DATA,
at      1218         $$ as in Chapter 5
write   Choose a chapter:
          1)  Introduction
          2)  Nouns
          3)  Pronouns
          4)  Verbs
arrow   1822
long    1            $$ get one digit; don't wait for NEXT
store   chapter
no
jump    chapter,x,x,intro,unoun,pron,verb,x
write   Pick a number between 1 and 4.
```

(As previously mentioned in Chapter 5, it would be better to execute the -base- command only after deciding to jump, so that the student could still use the BACK key to return to the original unit.) The -long- command following an -arrow- (but preceding any judging commands) sets a limit on the length of the student's response. The "long 1" is particularly useful here because the student need not press NEXT but has only to press the single number to begin the judging process. (For -long- of greater than 1 there must be an accompanying "force long" statement or else a NEXT key is required.) The -long- command must precede any judging command since the "long" specification is needed *before* the student starts typing (whereas we proceed past the judging command only *after* the student enters a response). The -long- command may be thought of as a kind of modifier of the -arrow- command, in the sense that the -arrow- sets a default maximum response length which is overridden (or modified) by the following -long- statement.

The -no- in this index unit is similar to an -ok- command in that it unconditionally terminates judging, but the -no- command makes a "no" judgment. If "chapter" is a number from 1 to 4, the -jump- will take the student to his chosen chapter. (Since -jump- erases the screen the "no" will not be seen.) If, however, "chapter" is not in range, we fall through the -jump- to an error message, and there will be a "no" next to the response (and the student must try again).

Student Specification of Non-Numerical Parameters

Now that we have seen how to let the student specify a number, let's see how to ask the student to tell us his or her name or nickname to permit us to communicate by name:

```
unit    meet
at      1215
write   Hello, my name is Sam Connor.
        What's your name?
arrow   162Ø
long    8           $$ limit to 8 characters
storea  name        $$ define "name" earlier
ok
write   Pleased to meet you, ◁a,name▷!
```

The -storea- command is a judging command which will store *alphabetic* information as distinguished from numeric information. The ◁a,name▷ is the embedded form of the statement "showa name" which will display alphabetic information. This unit will feed back to you any name you give it. Notice that you can't enter a name of more than 8 characters because of the -long- command. TUTOR stores a capital letter as a "shift" character plus the lower-case letter, so if capitalized, the name must be shorter because a capital letter counts as *two* characters. (Insert a "force long" statement anywhere before the -storea- if you would like judging to start upon hitting the -long- limit, without having to press NEXT.)

A statement of the form "storea name,3" will store just the first three characters of the student's response. You can get and keep a character count of the length of the student's name, including "shift" characters, by referring to the system variable "jcount", which is a count of the number of characters in the copy of the student response used for judging—hence the "j". With these facts in mind, change the -storea- to:

```
storea  name,(namlng⇐jcount)
```

This will store the whole response and save the length. Be sure to define both "name" and "namlng", but do *not* define "jcount" or you will override TUTOR's definition of its function. Also, to show the precise number of characters, change the embedded -showa- to:

```
⊲a,name,namlng⊳
```

The reason for saving the present value of "jcount" in "namlng" is that "jcount" will change at each -arrow- in the lesson, whereas throughout the lesson you will repeatedly use "showa name,namlng" or ⊲a, name,namlng⊳ to call the student by name. So, you want "namlng" to keep the name length. Incidentally, a -showa- with only a single argument (such as "showa name") will show ten characters, which is the number of characters (including shift characters) that will fit in one of your variables.

It is possible to store alphabetic information which is longer than ten characters. Change the "long 8" to "long 2∅". Suppose you've defined "name=v24." In this case, you must make sure that you are *not using v25,* and change your defines if necessary. The 2∅-character name will need both v24 *and* v25 since each variable can hold only ten characters. With these changes it is possible to enter a long name (e.g., Benjamin Franklin, which is 19 characters counting shift characters).

Difference Between Numeric and Alphabetic Information

When we were studying the desk calculator unit, we defined a variable "bob=v3∅" for the student. Suppose the student responds with the word "bob". If we use a numeric -store-, we will get the number presently contained in v3∅, which might be 529.3. If we use an alphabetic -storea-, we will get the string of characters "bob" which is simply a name and nothing more. Perhaps the distinction is most easily seen with an example, which you should write and try out at a PLATO terminal:

```
define  student
        bob=v1
define  ours,student           $$ include "student" set of defines
        name=v2,num=v3
unit    test
calc    bob⇐π                  $$ π means 3.14159 ........
arrow   1815
store   num
storea  name
ok
write   num=◁s,num▷
        name=◁a,name,jcount▷
```

Consider various responses. For example, "2bob" should give a *numeric* 2π (6.2832) and an *alphabetic* "2bob". Most often, we speak of "alphanumeric" information (letters and numbers) in the latter case. The response "3−4/5" yields a numeric 2.2 and an alphanumeric "3−4/5".

In other words, a storea/showa combination feeds back *exactly* the alphanumeric text entered by the student. However, a -store- involves a numerical evaluation of the student's response, and a later -show- converts this numerical result into appropriate characters to display on the screen (so that you can read the result). You might interchange the "num" and "name" arguments on the -store- and -storea- commands to see the unusual things that happen if you pair -store- with -showa- (instead of -show-) or if you pair -storea- with -show- (instead of -showa-).

To sum up, if you accept numeric information with a -store-, display it with a -show-. If you accept alphanumeric information with a -storea-, display it with a -showa-.

More On -answer- and -wrong- (Including -list- and -specs-)

There are some additional features of -answer- (and -wrong-) which should be pointed out. First, -answer- will not only handle word or sentence responses but will also handle numbers:

```
answer  7 women <and> 5 men
```

This -answer- will be matched by a student response of the form "14/2 women and 3+2 men" because simple expressions such as 14/2 or 3+2 are evaluated by the -answer- command. Currently, the -answer- command will not handle very complicated numerical expressions.

(Later we will discuss the -ansv- and -wrongv- commands which handle expressions as complicated as those handled by -store- but without the sentence capabilities of -answer- and -wrong-. There are also -ansu- and -wrongu- commands which are similar to -ansv- and -wrongv- but treat scientific units on a dimensional basis.)

If the student says "37 women and 5 men," the incorrect number 37 will have xx under it, whereas the response "6.5 women and 5 men" will have the 6.5 underlined since it is nearly correct (similar to a misspelling of a word). Normally -answer- and -wrong- consider numbers off by less than 1∅% to be "misspelled." You can alter these specifications by preceding the list of -answer- and -wrong- commands with a -specs- command:

```
unit    trial
arrow   1815
specs   toler,nodiff
answer  7 women <and> 5 men
```

The -specs- command is a *judging* command which affects the operation of other judging commands which follow it. Here it has been used to specify that a "tolerance" of 1% is permitted and that "no difference will be allowed for underlining" (normally 1∅%). Having specified both "toler" and "nodiff," any expressions within 1% of 7 and 5 will be accepted, but expressions with larger discrepancies will not be underlined.

Note carefully that since -specs- is a judging command, it terminates the processing of regular commands. Among other things, this means that a -long- command must *precede* the -specs-, not follow it. If -long- comes after -specs-, TUTOR could not prevent the student from entering a longer response (since it could not see the -long- command before it paused for the student's response).

Here are some other useful applications of -specs-:

```
specs   okcap,okspell
answer  the antidisestablishmentarianism doctrine
```

This allows the student to capitalize words, and specifies that misspellings are to be considered ok. Note that if the -answer- tag contains capitalized words, the student must also capitalize those words. The "okcap" makes capitalization optional only for those words you have not capitalized. You can use -specs- to ignore extra words:

```
specs   okextra
answer  Washington
```

This states that it is ok to have extra words, so that "It was George Washington" will be an acceptable response. The following is another example of -specs- capabilities:

```
specs   noorder
answer  apples pears and peaches
```

This specifies that no particular word order is required. Note the absence of commas in the -answer- tag. (Such punctuation marks are not allowed there, but all punctuation marks are ignored in the student's response, so he or she may use commas). Also, note that "answer apples, pears and peaches" would represent two synonymous answers, and the student could respond either with "apples" or with "pears and peaches". There exists a much less powerful -exact- command (as well as other techniques) for judging particular punctuation when that is necessary. For example, it is possible to use the -change- command to redefine the comma to be a "word" rather than a punctuation mark. In that case, some otherwise unused character must be defined to take the place of the comma in specifying synonyms.

```
specs   nookno
ok
```

Here we specify that no "ok" or "no" be displayed beside the student's response, contrary to the normal situation. (As an alternative, the -okword- and -noword- commands can be used to change the words TUTOR uses from "ok" and "no" to something else.)

(For other -specs- capabilities see reference material described in Appendix A.)

Another important feature of -specs- (in addition to its use in specifying various options) is that it marks a place to return to *after* judging. Consider the following unit. You do *not* define the system variable "spell".

```
unit    presi
at      1212
write   Name one of the first three U.S. presidents.
arrow   1513
specs   bumpshift       $$ delete shift codes
at      25Ø8
writec  spell,No misspellings!,
        Underlining indicates a misspelled word.
```

```
answer   washington
write    Good old George.
answer   adams
answer   jefferson
```

Suppose the student types "WASHINGTON". TUTOR starts judging just after the -arrow- and encounters -specs-, a judging command. The tag ("bumpshift") tells TUTOR to change the response to "washington" for judging purposes. (Incidentally, this operation changes "jcount", the character count of the judging copy of the student's response, from 2Ø to 1Ø because the "shift" characters are knocked out.) Moreover, TUTOR makes a note that it encountered a -specs- command as the fourth command in unit "presi", and this marker will be used in a moment. TUTOR skips the following -at- and -writec- because regular commands are skipped in the judging state.

Next, TUTOR encounters "answer washington" which matches the student's (altered) response, and this terminates judging. The succeeding regular commands are processed as usual. In this case, there is only a "write Good old George" before we run into another judging command ("answer adams") which stops the processing.

Actually, processing has not completely stopped. It is at this point that TUTOR asks one last question: "Did I pass a -specs- command in processing this response?" The answer is yes (at the fourth command in unit "presi"). *TUTOR now processes any regular commands following that -specs- marker.* In this case, TUTOR does an "at 25Ø8" and a -writec- before finally being stopped (*really* stopped this time) by the first -answer- command.

The -writec- refers to the system variable "spell" which is true (−1) if the spelling is correct, and false (Ø) if a misspelling has been detected. The variable "spell" is −1 if there are no underlined words, but there may be X'ed words (words that are completely different).

The usefulness of the marker property of -specs- is that you can specify a central place to put messages and calculations, which should be done no matter which judging command is matched. We will see additional applications of this useful feature of -specs-. Notice that a later -specs- command will override an earlier -specs- marker in a manner analogous to the way a later -help- command overrides an earlier setting of the "help" marker. Note, too, that if no regular commands follow the -specs-, TUTOR finds nothing to do when it comes there after being nearly stopped as described above. This was the situation in our previous examples such as:

```
specs   nookno
ok
```

In this example, there are no regular commands between the -specs- and the -ok-.

Let us return for a moment to the -answer- command. We had examples involving synonyms such as (right,rt) or (Va,Virginia). A convenient way to specify synonym lists which occur frequently in a lesson is to define a -list-:

```
list   affirm,yes,ok,yep,yeah,sure,certainly
```

Here "affirm" is the title of a list of synonyms ("affirm" is not itself a member of that list). With this definition, which should be placed at the very beginning of your lesson along with your -define- statement, you can write:

```
answer   ((affirm))
wrong    maybe ((affirm))
```

These are equivalent to:

```
answer   (yes,ok,yep,yeah,sure,certainly)
wrong    maybe (yes,ok,yep,yeah,sure,certainly)
```

Note that "answer we affirm" does not imply this list of synonyms, just as a single important word by itself does not refer to a list. You can use the list equally well to specify optional words, as in:

```
answer   <<affirm>>   it is
```

Here <<affirm>> is equivalent to <yes,ok,yep,yeah,sure,certainly>. Note that <affirm> merely refers to the single word "affirm". Double marks are needed to refer to the list whose title is "affirm". You can combine references to synonym lists with individual words. For example:

```
wrong    usually (definite, (affirm))
answer   often <definite, <affirm>>
```

The following list might also be particularly useful:

```
list   negate,no,nope,not,never,huhuh
```

This covers the main capabilities of the -answer- and -wrong- commands and their associated -list- definitions. The -specs- command may be used to modify how -answer- works and also serves as a useful marker. The marker function of -specs- is not limited to -answer- but holds for any judging commands which follow it, including -ok- and -no-.

The -answer- (or -wrong-) command can nicely handle responses which involve a relatively small vocabulary of words. It is, therefore, adequate when the context limits the diversity of student responses (such as foreign language translation drills where there are only a few permissible translations of the sentence and each such sentence contains a rather small number of allowable words). The detailed markup of the response provides the student with useful feedback in such a drill.

The -answer- command is not well-suited to a more free dialog with the student where the context is broader and where the vocabulary used by the student may encompass hundreds of words. In the next section we discuss the -concept- command which can cope with more complexity.

Building Dialogs With -concept- and -vocabs-

An excellent example of a dialog is a lesson on qualitative organic chemistry analysis written by Prof. Stanley Smith of the Department of Chemistry, University of Illinois, Urbana. This lesson helps students practice their deductive skills on PLATO before they identify unknown compounds in a laboratory. Prof. Smith has PLATO randomly choose one of several organic compounds and then invites the student to ask experimentally-oriented questions aimed at identifying the unknown. Typical questions are: "what is the melting point;" "does it dissolve in sulfuric acid;" "show me the infrared spectrum;" "is it soluble in H_2O." There are over a hundred such concepts important in this simulated laboratory situation, and since each concept has many equivalent forms drawing upon a vocabulary of hundreds of words, the number of possible responses is astronomical. How can this be handled?

Although the context is far broader than that of a language drill, it is, nevertheless, sufficiently limited to be tractable. No attempt is made to recognize arbitrary student responses such as "cook me some apple pie." With this quite reasonable restriction, the situation can be handled by using the -vocabs- command (analogous to -list-) to define a large vocabulary (with appropriate "synonymization") associated with a list of -concept- commands (analogous to -answer-) which express the basic concepts meaningful in the context of this lesson. The following is a fragment of the -vocabs- command:

```
vocabs  labtest          $$ vocabulary must have a name
        <is,it,a,does,in,what>  $$ ignorable words
        (color,red,blue,green)  $$ word number 1 and synonyms
        (water,H2O)             $$ word number 2 and synonym
        (dissolve,soluble)      $$ word number 3 and synonym
        .
        .
        .
```

And here are a couple of the many -concept- commands:

```
.
.
arrow    1213
concept  what color
write    It is red.
concept  soluble in water
write    It's slightly soluble in water.
.
.
.
.
.
.
```

Consider what TUTOR does with "concept soluble in water". TUTOR knows that -concept- has a tag consisting of words defined by a previous -vocabs-. (As usual with such matters, the -vocabs- should be at the beginning of the lesson.) The first word in the tag is "soluble" which TUTOR finds is the *third* very important word in the vocabulary (discounting the ignorable or optional words "is,it,a," etc.). TUTOR groups synonyms together so that "dissolve", too, would be considered a "number 3" vocabulary word. The next word of the tag is "in" which TUTOR throws away because the -vocabs- command says that the word is ignorable. The next word is "water", which is in the *second* set of important -vocabs- synonyms. The net result is that "concept soluble in water" is converted to the sequence "3 2".

Now, consider a student in this lesson who types "does it dissolve in H_2O". Superficially, this looks quite different from the -concept- tag "soluble in water". However, TUTOR encounters a -concept- command which, unlike -answer-, indicates that the student's response should be looked up in the defined vocabulary. (In the case of -answer- there is no one vocabulary set because each -answer- may include various -list- references and particular words specific to that -answer-.) By a process identical to the conversion of the author's -concept- tag, TUTOR converts the student's response into "3 2". This compact form "3 2" does not match the first "concept what color" (which was converted to "1"), so, TUTOR proceeds to the next judging command, which is "concept soluble in water" or rather its converted form "3 2". This matches, so judging terminates and regular processing begins. The student gets a reply "It's slightly soluble in water."

Notice that the first -concept- encountered triggers the transformation of the student's response into the compact form suitable for looking through a very long list of concepts. If the -vocabs- contains an entry such as (five,5,cinco), the student may match this entry with "3+2", just as in an -answer- statement involving numbers.

You will have to experiment a little with this machinery in order to learn how best to manage the synonymization in the vocabulary. This does depend on the context. In an art lesson it would be disastrous to call red and blue synonyms as was done here, but it makes sense in this context (where the only concept related to color has to do with "what color is it", which means essentially the same as "is it red" or "is it blue").

You will find that the use of words not defined by -vocabs- will result in a markup indicating which words are undefined (X's will appear under these words). If your context is such that you need worry only about key words and don't care if the student asks "does it dissolve superbly in water", you might precede the first -concept- with a "specs okextra" which says that extra student words not found in the vocabulary may be ignored, as though they had been so specified in the -vocabs- tag. In that case, you need not define any ignorable words with -vocabs-, but you would write "concept dissolve water", not "concept dissolve in water" since extra *author* words are not tolerated. If you don't use "specs okextra", the student's word "superbly" will be marked (xxxxxxxx). If the student misspells a vocabulary word, that word will be underlined such as "saluble in water."

The following is an alternative and more detailed version of the heart of the dialog lesson, which illustrates several points. It is a rather complex example which brings together many aspects of TUTOR. Note particularly that the -concept- statements now are listed one after the other. The variable "unknown" is a number from 1 to 4 (associated with which compound the student is attempting to identify). The *system* variable "anscnt" is set to zero when judging starts (and when a -specs- is encountered) and it counts the number of -answer-, -wrong-, -ok-, -no-, and -concept- commands passed through. If the third such command terminates judging, "anscnt" will have the value 3. If no match is found, "anscnt" is set to −1.

```
                    .
                    .
                    arrow    1213
                    wrong    what is it
                    write    That is for you to determine!
```

(Continued on next page.)

```
specs              $$ to clear anscnt again
goto     anscnt>∅,unknown,x
writec   vocab,I don't understand your sentence.,
         The xxxx words are not in my vocabulary.
concept  what color
concept  soluble in water
concept  boiling point
.
.
.
.
unit     unknown
goto     unknown−2,reply1,reply2,reply3,reply4
*
unit     reply1
writec   anscnt,,,It is colorless.,
         It is slightly soluble in water.,
         The boiling point is 245-247° C.,
.
.
```

The statement "wrong what is it" is necessary because a "concept what is it" contains only ignorable words and would, therefore, not distinguish between "what is it" and "does it what", which also contains only ignorable words. Since -specs- resets "anscnt" to zero, "anscnt" will have the value 2 if the student's response matches the second -concept- ("soluble in water"). No regular commands follow this -concept-, so TUTOR goes right to the -specs- marker to execute the regular commands there. Since "anscnt" is greater than zero, TUTOR does a -goto- to unit "unknown", where there is a -goto- to unit "reply1" (assuming we are working on unknown number 1), which writes "It is slightly soluble in water" on the student's screen.

This structure makes it *very* easy to add a fifth unknown compound to the lesson. The -vocabs- and list of -concept- commands do not have to be changed, since the basic concepts and vocabulary are pertinent to the analysis of *any* compound. All that is necessary is to add "reply5" to the end of the conditional -goto- in unit "unknown" and to write a unit "reply5" patterned after unit "reply1". The lesson revision is completed!

What happens if the student says "it what does"? This will not match the -wrong- nor any of the -concept- commands, so "anscnt" will be −1. Therefore, the -goto- just after the -specs- will fall through to the following -writec-, which gives one of the two messages dependent on the system variable "vocab": *true* if all words are found in vocabulary, *false* if some words are not found (these words would be underscored with xxxx). In this case, the student will get the message "I don't understand your sentence", whereas if the student says "what is elephant" he will see the xxxx's under "elephant" and get the message "The xxxx words are not in my vocabulary".

That was a fairly complicated example, but the discussion is justified by the general usefulness of many of the techniques employed and by the extraordinary power such a structure yields, both in its sophisticated handling of student responses and in the ease of expansion to additional options.

Suppose the -arrow- is in unit "analysis". One way to proceed from one question to the next would be to place a "next analysis" in this unit. There is an efficient way to avoid erasing and recreating the display associated with this unit. Instead of proceeding, let's judge each response "wrong" so that we stay at this -arrow-. Replace the -specs- command with these two statements:

```
        .
        .
        .
        .
        .
        .
        specs   nookno   $$ so "no" doesn't appear
        judge   wrong
        .
        .
        .
        .
```

Despite its name, -judge- is a *regular* command, *not* a judging command. It can be used to alter the judgment made by the judging commands. In this case, TUTOR first skips over this regular command to get to the -concept- commands. If one of these commands matches the student response, TUTOR makes an "ok" judgment, but upon going to the -specs- marker TUTOR finds a "judge wrong" which overrides the earlier judgment. TUTOR keeps going, processing regular commands, and produces a message as we have seen before. The "nookno" specification prevents a "no" from appearing on the screen and the student simply sees our message. But the -arrow- has not been satisfied, so when the student presses NEXT, TUTOR erases the response and awaits a new response. Each time, the student gets a reply to his or her experimental question, and the "wrong" judgment takes us back to the -arrow-.

This is a good way to manage the screen because only a small portion of the display changes (the surrounding text and figures remain untouched). The "next analysis" re-entry to this same main unit would quickly get tiresome because of the repetitious replotting of the surrounding material.

You should now be able to use -answer-, -wrong-, and -list- in situations where the vocabulary is small and -concept- and -vocabs- where the vocabulary is large. You have seen how to use -specs- both to specify various judging options and to mark a place where post-judging actions can be centralized. You have seen one form of the regular -judge- command "judge wrong" which overrides an "ok" judgment made by an -answer- or -concept-.

Another way to get a "wrong" judgment is to use -miscon- ("misconception") commands instead of -concept- commands. Just as -wrong- is the opposite of -answer-, -miscon- is the opposite of -concept-.

There is a particularly convenient way to make different concepts equivalent, including different word orders:

```
concept   dissolve in water
          water soluble
          drop in water
write     It's soluble in H2O.
```

The "continued" -concept- specifies synonymous concepts. If the student's response matches any of these three concepts the same message will be given. Also, "anscnt" will be the same no matter which of these concepts makes the match.

Use of -vocabs- makes possible the underlining of misspelled vocabulary words (or their acceptance with a "specs okspell"), just as with the -answer- command. Similarly, "specs noorder" can be used to indicate that no particular word order is required. There is a -vocab- command which permits a larger vocabulary (at the price of giving up these spelling and order capabilities). Just as the multi-word phrase "sodium*chloride" can be used with the -answer- command, so can such phrases be specified in a -vocabs- vocabulary.

At times you may be interested mainly in root words, no matter what endings are attached. The words "walk", "walks", "walked", "walker", and "walking" can be added to a -vocabs- very simply as "walk/s/ed/er/ing", which saves you some typing effort. If you want all of these *except* for "walk" itself to be added to the vocabulary, use a double slash after the root: "walk//s/ed/er/ing".

An even more compact way to define common endings is with -endings- commands:

```
endings  ∅,s,ed,ing
endings  9,er,est
. . .
vocabs   sample
         will/∅,full//9
```

The use of the "∅" and "9" sets of endings causes the vocabulary to contain these words: will, wills, willed, willing, fuller, and fullest ("full" itself is missing, due to the double slash). An -endings- set must be identified by a number from ∅ to 9.

Numbering Vocabulary Words

Suppose the student is encouraged to ask questions such as "What is the capital of Alabama?" or "What is the area of Alaska?" A compact and powerful way to handle all the states is to specify a vocabulary class ("state") and number the various states. For example:

```
define   st=v1
vocabs   inquiry
         <What,is,the,of>
         (state, Alabama=1, Alaska=2, Arizona=3, . . . . . . .)
         capital, area
. . .
concept  capital of state,st⇐state
writec   st,,,Montgomery,Juneau,Phoenix
concept  area of state,st⇐state
writec   st−2↕51,6∅9↕586,4∅∅↕113,9∅9↕ . . . . . .
write    sq. mi.
```

If the student asks "What is the capital of Alaska?" the first -concept- is matched ("capital of state"), and variable "st" is assigned the value "2", since "Alaska" was given the value "2" in the vocabulary. Now "st" can be used in the following -writec- to tell the student the name of the capital (Juneau). Similarly, if the student asks "What is the area of Arizona", the second -concept- is matched, "st" is assigned the value "3", and the student is given the reply "113,9∅9 sq. mi."

We can go even further. Consider this altered version, in which the two -concept-s are combined:

```
define    st=v1,prop=v2
vocabs    inquiry
          <What,is,the,of>
          (state, Alabama=1, Alaska=2, Arizona=3, . . . . . .)
          (property, capital=1, area=2)
. . .
concept   property of state, st⇐state,prop⇐property
writec    2(state−1)+(prop−1)↕Montgomery↕51,6∅9
          Juneau↕586,4∅∅↕Phoenix↕113,9∅9↕ . . . . .
writec    prop=2↕ sq. mi.↕↕
```

Suppose the student asks about "the area of Alabama". The -concept- is matched, "st" is assigned the value "1", and "prop" is assigned the value "2". The expression "2(state−1)+(prop−1)" reduces to "2(∅)+1" or "1", which picks out "51,6∅9" from the first -writec-. Since "prop" does equal "2", the second -writec- will write "sq. mi." on the screen beside the area number. (It would be good practice for you to determine the steps that would be taken if the student asked about "the capital of Arizona.")

Synonyms, phrases, and endings can be numbered, as in this -vocabs- entry:

```
(verbs, walk=1/ed=2, stroll=1/ed=2, went*past=3)
```

According to this numbering scheme, "walk" and "stroll" are number 1 among the "verbs," "walked" and "strolled" are number 2, and the phrase "went past" is number 3.

The -judge- Command

We have encountered the regular command -judge- (*not* a judging command) and have seen how it can be used to "judge wrong" a response that had already received an "ok" judgment. The -judge- command may also be used to "judge ok" a response (disregarding what a previous judging command may have had to say). The following is a conditional form for this type of -judge- command:

```
judge   3a-b,ok,x,wrong
```

This form will either make the judgment "ok", leave the current judgment as is (the "x" option), or make the judgment "wrong", depending on the condition "3a-b".

Here is a useful example:

```
unit    negative
at      1214
write   Give me a
        negative number:
arrow   1516
store   num
write   Cannot evaluate your expression.
ok              $$ terminate judging
judge   num<Ø,ok,wrong
writec  num<Ø,Good!,That's positive!
```

We could just as well have written "judge num<Ø,x,wrong" since the original judgment was a universal "ok". (Later we will study -ansv- and -wrongv- which are also useful in numerical judging.) Note that "judge ok" and "judge wrong" do *not* cut off the following commands. In the above example, the -writec- *is* performed, even though it follows the -judge- command. The -judge- command here merely alters the judgment. If you want to cut off the following commands, you can use "judge okquit" or "judge noquit".

We have been using the -ok- or -no- commands to terminate judging unconditionally, as in the last example. It is sometimes useful to be able to switch in the other direction, from the regular state to the judging state. For example, suppose you want to count the number of attempts the student makes to satisfy the -arrow-:

```
  .
  .
  .
  .
calc    attempt⇐Ø
arrow   1518
ok
calc    attempt⇐attempt+1
judge   continue
answer  cat
   etc.
```

Judging starts just after the -arrow-. The -ok- terminates judging to permit executing the regular -calc- which increments the "attempt" counter. Then the regular -judge- command says "continue judging", which switches TUTOR back into the judging state to examine the -answer- and other judging commands which follow. If the response is finally judged "no", the student will respond again, and since judging starts each time from the -arrow-, the "attempt" counter will record each try. (Actually, system variable "ntries" *automatically* counts the number of tries, but structures similar to the structure illustrated here are often useful.)

Leaving out the -ok- and "judge continue" (which permit counting each attempt) is a common mistake. If you write:

```
calc      attempt⇐∅
arrow     1518
calc      attempt⇐attempt+1
answer    cat
```

then "attempt" will stop at one. TUTOR initializes "attempt" to ∅, then encounters the -arrow- and notes its position in the unit. Then, the following -calc- increments "attempt" to 1, after which the -answer- judging command terminates this regular processing to await the student's response. The student then enters his or her response and TUTOR starts judging. The first command after the -arrow- is the incrementing -calc-, which is skipped because it is a *regular* command and TUTOR is looking for judging commands. This will happen on each response entry, so "attempt" never gets larger than one. This explains the importance of bracketing the -calc- with -ok- and "judge continue".

A related option is "judge rejudge" which is similar to "judge continue". We have seen that "specs bumpshift" alters the "judging copy" of the response by knocking out the shift characters. The judging copy is the version of the response which is examined by the judging commands (such as -answer-). This version may differ from the student's actual response due to various operations such as "specs bumpshift". It is also possible to -bump- other characters or to -put- one string of characters in place of another. All such operations affect the judging copy *only* and do not touch the original response, which remains unmodified. The statement "judge rejudge" replaces the judging copy of the response with the original response, thus cancelling the effects of any previous modifications of the judging copy. The statement also initializes the system variables associated with judging, including "anscnt". It is, therefore, much more drastic than "judge continue", which merely switches TUTOR to the judging state without affecting the judging copy or the system variables.

Another exceedingly useful -judge- option is "judge ignore" which erases the student's response from the screen and permits him or her to type another response without first having to use NEXT or ERASE. Unlike "judge wrong", "ok", or "continue", "judge ignore" stops all processing and waits for new student input. (Even the commands following a -specs- won't be performed.) On the other hand, TUTOR goes on to the following commands after processing -judge- with tags "ok", "wrong", or "continue".

The following routine (which permits the student to move a cursor on the screen) is a good example of the heightened interaction made possible through the use of "judge ignore". We use the typewriter keys d,e,w,q,a,z,x, and c which are clustered around a 3 key by 3 key square on the keyboard, to indicate the eight compass directions for the cursor to move on the screen. These keys (shown in Fig. 7-2) have small arrows on them to indicate their common use for moving a cursor.

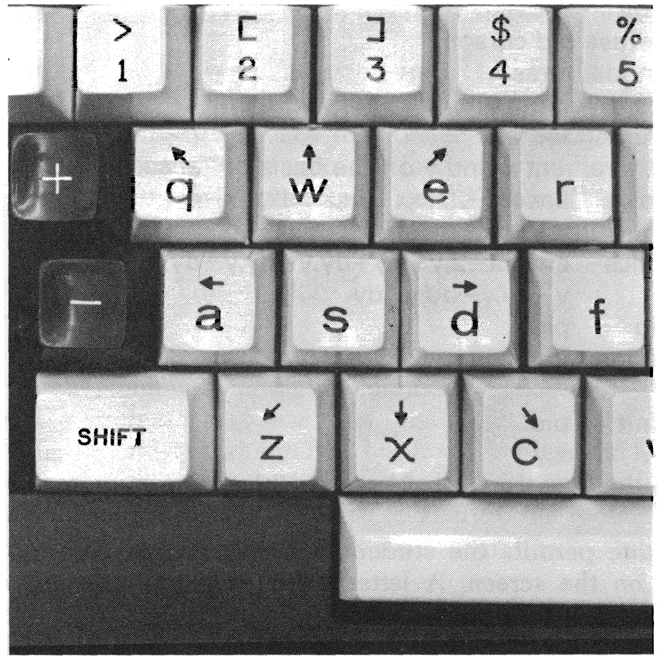


Fig. 7-2.

```
unit     cursor
calc     x⇐y⇐25Ø     $$ initialize cursor position
         dx⇐dy⇐1Ø    $$ cursor step size
do       plot        $$ plot cursor on screen
inhibit  arrow       $$ don't show the arrowhead
arrow    32Ø1
long     1
specs                $$ come here after judging
do       move        $$ -do- is a regular command
answer   d           $$ east:   anscnt=1
answer   e           $$ northeast        2
answer   w           $$ north            3
answer   q           $$ northwest        4
answer   a           $$ west             5
answer   z           $$ southwest        6
answer   x           $$ south            7
answer   c           $$ southeast        8
ignore               $$ equivalent to: { no
*                                      { judge   ignore
unit    move
*erase old cursor
mode    erase
do      plot
mode    write
*increment x and y on the basis of "anscnt"
calcs   anscnt-2,x⇐x+dx,x+dx,x,x-dx,x-dx,x-dx,
        x,x+dx
calcs   anscnt-2,y⇐y,y+dy,y+dy,y+dy,y,
        y-dy,y-dy,y-dy
do      plot
judge   ignore
*
unit    plot
at      x,y
write   +           $$ use "+" for cursor
```

This routine permits the student to move the cursor rapidly in any direction on the screen. A letter which matches one of the -answer-statements will cause the -calcs- statements to update x and y appropriately to move in one of the eight compass directions. The "long 1" makes it unnecessary to press NEXT to initiate judging, and the "judge ignore" after the replotting of the cursor again leaves TUTOR awaiting a new response. The "judge ignore" greatly simplifies repetitive response handling such as that which arises in this example. Normally, such a cursor-moving routine would be associated with options to perform some action, such as drawing a line. This would make it possible for the student to draw figures on the screen.

In addition to the -judge- options discussed above, there is a "judge exit" which throws away the NEXT or timeup key that had initated judging. This leaves the student in a state to type another letter on the end of his or her response. This can be used to achieve special timing and animation effects.

To summarize, the -judge- command is a *regular* command used for controlling various judging aspects. The -ok-, -no-, and -ignore- are *judging* commands which somewhat parallel the "judge ok", "judge no", and "judge ignore" options. The "judge rejudge" and "judge continue" options make it possible to switch from the regular state to the judging state (with or without reinitializing the judging copy of the student response and the system variables associated with judging). All of these options may appear in a conditional -judge- with "x" meaning "do nothing":

```
judge   expr,no,x,ok,continue,wrong,rejudge,x,ignore,ok
```

The subtle difference between "judge wrong" and "judge no" will be discussed in Chapter 12 in the section on "Student Response Data". Basically, "judge wrong" is used to indicate an anticipated (specific) wrong response, whereas "judge no" indicates an unanticipated student response. Additional -judge- options are "quit", "okquit", and "noquit".

Finding Key Words: The -match- and -storen- Commands

The -match- command, a judging command, makes it easy to look for key words in a student's response. The -match- command will not only find a word in the midst of a sentence, but it will replace the found word in the judging copy with spaces, to facilitate the further use of additional judging commands (including -match-) to analyze the remainder of the response. Here is the form of a -match- statement:

```
match   num,dog,(cat,feline),horse,(pig,hog,swine)
              ∅       1         2          3
```

Here "num" is a variable which will be set to −1 if none of the listed words appear in the student's response, to ∅ if "dog" appears, to 1 if "cat" or "feline" is present, 2 if "horse" is in the response, etc. In any case,

-match- terminates judging, with a "no" judgment if num= −1 or an "ok" judgment otherwise. What if more than one of the words appear in the student's response? Suppose the student says:

"horse and dog"

In this case "num" will be set to 2 because in looking at the first student word we find a match (horse). The judging copy of the response is altered by replacing "horse" with spaces so that it looks like:

"   and dog"

If we were to execute the same -match- again we would get the number Ø corresponding to "dog", and the judging copy would then look like:

"   and   "

Note that -match- always terminates judging, so that a "judge   continue" is needed before another -match- can be executed. Also note that the key words are pulled out in the order in which they appear in the student's response, not in the order they appear in the -match- statement.

There are many other ways in which the -match- can be utilized. First, we can improve greatly on our cursor program:

```
.
.
.
.
inhibit  arrow
arrow    32Ø1
long     1
match    num,d,e,w,q,a,z,x,c
do       num,x,move
judge    ignore
.
.
.
.
```

Unit "move" remains unchanged except to replace (in two places) the expression "anscnt−2" by the expression "num−1" (and we can delete the "judge   ignore" in unit "move"). We see that -match is useful for converting a word to a number which represents the word's position in a list.

Another good use of -match-.is in an index:

```
unit    table
base
term    index
at      1218
write   Choose a chapter:
             a) Introduction
             b) Nouns
             c) Pronouns
             d) Verbs
arrow   1822
long    1
match   chapter,a,b,c,d
calc    chapter⇐chapter+1
jump    chapter,x,x,intro,unoun,pron,verb,x
write   Pick a,b,c, or d.
```

Notice that we must increment "chapter" by one if we want topic "a" to be chapter 1, since -match- associates ∅ with the first element in its list (−1 is reserved for the case where no match is found). If no match is found, there is a "no" judgment. (Again, -base- could come later in the unit, or at the beginning of the chapters, in which case the BACK key would still be active for returning to the place from which the index was accessed.)

These applications barely scratch the surface of -match-s capabilities. Here are some other ideas on how to use -match-:

1) Use -match- to pull out negation words such as no, not, never, etc. Then "judge continue" and use -answer- or -concept- commands to analyze the remainder of the response. You can in this way separate the basic concept from whether it is negated, with the negation information held in the -match- variable for easy use in conditional statements.
2) Use -match- to identify and remove a key-word directive before processing the rest of the information. This comes up in simulating computer compilers, in games ("move" or "capture"), etc.

A related command is -storen-, which will find a simple numeric expression in a sentence, store it in your specified variable, and replace the expression with spaces. This is particularly useful for pulling out several numbers. The -store- command will handle much more complicated expressions including variables as well as numbers, but can get only one number. For example, the student might respond to a question about graph-plotting coordinates with "32.7,38.3". These two numbers can be acquired by:

```
        .
        .
        arrow     1215
        storen    x
        write     You haven't given me numbers.
        storen    y
        write     You only gave me one number.
        answer              $$ remainder should be essentially blank
        no
        write     There should just be two numbers.
```

Like -store-, -storen- will terminate judging on an error condition (in which no number was found). In the example, the first -storen- removes and stores one number in "x" and the second -storen- looks for a remaining number to store in "y". The first -storen- will terminate judging if there are no numbers. The second -storen- will terminate judging if there is no number remaining after one has been removed. The blank -answer- will be matched if only punctuation, such as commas, remains after the actions of the two -storen-s.

Numerical and Algebraic Judging: -ansv- and -wrongv-

We have already had some experience in handling numerical and algebraic responses by using -store- to evaluate numerically the student's expression. The -ansv- (for "answer is variable") and -wrongv- judging commands evaluate the student's expression in the same way as -store- and also perform a comparison with a specified value.

The -ansv- command is useful in association with -store-. If you ask the student for a chapter number or a launch velocity of a moon rocket, it is convenient to use -ansv- to check whether his number is within the range you allow. For example:

```
        .
        arrow   1314
        store   chapter
        ansv    5,4       $$ match if in the range 5±4 (1 to 9)
        no
        write   Choose a chapter from 1 to 9.
        .
```

Another common use is in arithmetic drills:

```
define    b=v1,c=v2
unit      drill          $$ multiplication drill
next      drill
randu     b,1Ø           $$ pick an integer from 1 to 1Ø
randu     c,1Ø           $$ pick another integer
at        1513
write     What is ⊲s,b⊳ times ⊲s,c⊳?
arrow     1715
ansv      b×c            $$ no tolerance
write     Right!
wrongv    b+c
write     You added.
wrongv    b×c,1          $$ plus or minus 1
write     You are off by 1.
wrongv    b×c,2Ø%        $$ plus or minus 2Ø%
write     You are fairly close.
no
write     You are way off!
```

The drill as written will run forever. It could be modified to stop after 5 straight correct responses, or after some other criterion has been met. Note that the response "bc" or "b×c is judged "no" (unless you define these variables in the "student" set of defines). Also note that the student need not do any mental multiplication for this drill (since if the student is asked to multiply 7 times 9, he or she could respond with 7×9 which matches the -ansv-).

Let's make a change to require some multiplication on the part of the student:

```
.
.
ansv      b×c
judge     opcnt=Ø,ok,wrong
writec    opcnt=Ø,Right!,Multiply!
wrongv    b+c
.
.
```

Do not define "opcnt"! It is a system variable which counts the number of operations in the student's response. If the student says "7(5+8+3)/2" then "opcnt" will be 4 because the student's expression contains:

1) an (implied) multiplication (7 *times* a parenthesized expression);
2) two additions; and
3) a division.

In this drill we want the student to give the result with no operations, so "opcnt" should be zero ("specs noops,novars" can also be used to prevent the student from using operations or variables in his or her response).

Recall that the first -concept- command encountered will trigger the reduction of the student's response to a compact form, through the use of the -vocabs-. This compact form can be compared rapidly to all succeeding -concept- commands. Similarly, the first -store- or -ansv- or -wrongv- causes TUTOR to "compile" the student's expression into a form which can be quickly evaluated when another of these commands is encountered. It is during the compilation process that "opcnt" is set. Just as the -vocabs- list tells TUTOR how to interpret the student's words, so the "define student" set of names tells TUTOR how to treat names encountered in the compilation of a student's algebraic response. So, there are many parallels between -ansv- and "define student" on the one hand and -concept- and -vocabs- on the other.

Let's look at an algebraic example, as opposed to the numerical examples we have treated:

```
define   student
         x=v1
unit     simplify
at       1215
write    Simplify the expression
           3x + 7 + 2x − 5
randu    x         $$ pick a fraction between Ø and 1
calc     x⇐x+1     $$ change to 1 to 2 range
arrow    1418
ansv     5x+2      $$ Ø tolerance
goto     varcnt−1,toofew,x,manyvar   $$ how many x's
goto     opcnt−2,toofew,x,manyop     $$ how many operations
wrongv   5x+12
write    You should subtract 5, not add it.
no
goto     formok,x,tellerr
*
```

```
unit     toofew
write    Your expression is not sufficiently general.
judge    wrong
*
unit     manyvar
write    "x" should appear only once.
judge    wrong
*
unit     manyop
write    Not simplest form.
judge    wrong
```

Unit "tellerr" would contain a -writec- involving the system variable "formok" to tell the student precisely why his or her expression could not be evaluated. There could be several -wrongv- statements in the example to check for specific errors. The system variable "varcnt" during compilation of the student's expression counts the number of references to variables. For example, "x+3x+x+2" is numerically equivalent to (5x+2), so that this response will match the -ansv-, but "varcnt" will be 3 because "x" is mentioned three times. If both x and y were defined, the expression "2x+y+4x" would yield a "varcnt" of 3 (two x's and one y) and an "opcnt" of 4 (two implied multiplications and two additions).

In this way "opcnt" and "varcnt" may be used to distinguish among equivalent algebraic responses which differ only in form. Roughly speaking, what is usually called "simplest algebraic form" often corresponds to the smallest possible values of "opcnt" and "varcnt".

There are some minor technical points in the preceding example. For example, -randu- with only one argument produces a fraction between ∅ and 1. If this should happen to be very close to ∅ then "x" would be unimportant in the expression (5x+2), so it seems better to add one and give "x" a value between 1 and 2, which is comparable to the other quantities in the expression. We could have used the two-argument form (e.g., "randu x,8") to pick an *integer* value for "x". However suppose TUTOR chooses the integer 2 for "x". In this case, a student who happens to give "12" as his or her response will match the -ansv- by accident since 5x+2 = 5×2+2 = 1∅+2 = 12. On the other hand, with TUTOR picking a *fraction*, the student would have to type something like "8.93172462173" to accidentally match the -ansv-. This just won't happen. You would have to type different numbers 24 hours a day for hundreds of years to match accidentally. If you want even more security against an accidental match, just change the value of "x" and check again. In skeleton form, here is a way to do it:

```
ansv     5x+2
goto     varcnt−1,toofew,x,manyvar
goto     opcnt−2,toofew,checkup,manyop
wrongv   5x+12
.
.
unit     checkup
randu    x          $$ new value of x
calc     x⇐x+1
judge    continue
ansv     5x+2       $$ try again
.
.
.
```

A further check is that we require exactly one "x" and exactly two operations.

There is a way to give detailed feedback to the student in case his or her expression is not algebraically equivalent to the desired expression (5x+2). Suppose the student's incorrect expression is "6x+2", and that you have done a -storea- to save the response and a -store- to evaluate it for some *integer* value of x. Then ask the student this question:

```
.
.
.
write   What is the numerical value of
          3(◁s,x▷)+7+2(◁s,x▷)−5?
```

If x is 4, this will appear on the screen as:

```
What is the numerical value of
  3(4)+7+2(4)−5?
```

Many students can handle a numerical example even if an algebraic example gives them trouble, so this student is likely to reply correctly, either with or without some help, that this expression gives 22. You can then reply to the student with this statement (assuming the student's alphanumeric response is in "string" and its value is in "result"):

```
write   But your expression, ⊲a,string,count⊳,
        gives ⊲s,result⊳ in this case.
```

If the student's response was "6x+2", with a value of 26 (if x is 4), this appears on the screen as:

```
        But your expression, 6x+2,
        gives 26 in this case.
```

The student now sees that his or her expression "6x+2" does not give the value 22 which it should in the case where x is 4. You have fed back the student's own expression, evaluated for a particular case where the student can see there is a conflict. (In other words, anything the student says may be used against him or her.) Here is an opportunity for the student to learn, by example, a useful technique in simplifying complicated expressions: try some numerical cases for which you know the results and see whether they agree with the simplified expression.

It is possible to judge equations as well as expressions. Suppose we ask the student to simplify the equation "4x+3=x+12y−5". A suitable response might be "12y=3x+8" or "x=(12y−8)/3". Every time the student enters a response, let TUTOR pick a random value for the independent variable x, and calculate the corresponding value of the dependent variable y: y⇐(3x+8)/12. Consequently, any correct equation will be true (with value −1), and an incorrect equation will be false (with value ∅). Here is a unit embodying these concepts:

```
define  student,x=v1,y=v2
unit    equate
at      1215
write   Simplify the equation
          4x+3=x+12y−5
arrow   1718
ok
randu   x              $$ random x on each judging
calc    x⇐x+1
        y⇐(3x+8)/12          $$ y depends on x
judge   continue
ansv    −1             $$ logical true
do      ident
wrongv  ∅              $$ logical false
write   That is false.
```

(Continued on the next page.)

```
no                          $$ anything else
write     Give me an equation!
*
unit      ident
calc      y⇐3.72y           $$ change y arbitrarily
judge     continue
 wrongv   −1                $$ should not now be true
write     That is an identity!
ok
judge     varcnt>2,wrong,ok
writec    varcnt>2,Not simplified.,Fine.
```

If the student writes "3+4", this expression has the numerical value 7, so the reply is "Give me an equation!"

If the student writes "3=4", this expression has the numerical value ∅, since it is logically false, and the reply is "That is false."

If the student writes "$3^2+5=17-3$", which is equivalent to 14=14, TUTOR replies "That is an identity!" The student's response *is* true (14 *does* equal 14), so that this true relationship has the value −1 which matches the -ansv- statement. A "do ident" follows, where the dependent variable y is changed so that y no longer bears the correct relationship to x. If the student's response had been a correct simplification of the given equation, his or her expression would no longer be true (−1), since y is no longer the correct function of x. In the case of "$3^2+5=17-3$", however, changing y has no effect and the value is still −1, which matches the -wrongv- statement in unit "ident". The student gets the message "That is an identity!"

Only if the student enters an equation which is not an identity will he or she get an "ok" judgment. Note the check on "varcnt". There could also be a check on "opcnt".

To summarize, -ansv- and -wrongv- are extremely powerful commands for algebraic or numeric responses, particularly in association with variables defined in the "define student" set. The system variables "opcnt" and "varcnt" give you additional information about the *form* of the response.

CAUTION: Since TUTOR performs multiplications before divisions (unless parentheses intervene), a student response of "1/2x" is taken to mean "1/(2x)", whereas the student might have in mind "(1/2)x". It is important to warn your students of this convention at the beginning of a lesson which uses algebraic judging. Scientific journals and most textbooks follow this same convention, but many students are unaware of this. Usually, printed materials use the forms $\frac{x}{2}$ or $\frac{1}{2}x$ or $\frac{1}{2x}$. These forms avoid the ambiguities that arise from the slash (/) or quotient sign

(÷) used on a single typewritten line. It is hoped that eventually TUTOR will make it easy for students to type fractions with the horizontal bar rather than with the slash or quotient sign. Until then, it is important to point out this convention to your students.

Handling Scientific Units: -ansu-, -wrongu-, and -storeu-

Suppose you want to ask the student for the density of mercury. A correct answer would be "13.6 grams/cm^3", but there are many equivalent ways to write the same thing. For example, the student might write "13.6×1Ø$^{-3}$kg/ (.Ø1 meter)^3" or "13.6 gm-cm^{-3}", and both of these responses are equivalent to "13.6 grams/cm^3". TUTOR provides a convenient way not only to judge such responses appropriately, but to give the student specific feedback if he or she makes specific errors (such as omitting the units or giving the right units but the wrong number).

The TUTOR scheme is based on the judging performed by human instructors when grading exam questions involving numbers and units. The instructor makes two separate checks, one for the numerical value and the other for the *dimensionality* of the units. The dimensionality of density is (mass)^1 (length)$^{-3}$, and it is the powers (1,−3) that we are interested in as well as the number 13.6. All of the equivalent correct responses listed above have a numerical value of 13.6 (in the gram-cm system of units) and a mass-length dimensionality of (1,−3). The -storeu- command (-store- with units) can be used to get the numerical part and the dimensionality if we define the units appropriately:

```
define  student        $$ units will be used by student
☞       units,gm,cm  $$ can define up to 1Ø basic units
        gram=gm,grams=gm,kg=1ØØØgm  $$ synonyms
        meter=1ØØcm,cc=cm³
define  mine,student        $$ include student define set
        num=v1,dimens(n)=v(1+n)$$ see "Arrays", Chapter 1Ø
unit    dense
at      1215
write   What is the density of mercury?
        (Include units!)
arrow   1618
☞storeu num,dimens(1)
write   Cannot evaluate.
no
```

(Continued on the next page.)

```
goto    num≠13.6,badnum,x
goto    dimens(1)≠1,badmass,x
goto    dimens(2)≠−3,badleng,x
judge   ok
write   Good!
```

We will go to a unit "badnum", "badmass", or "badleng" (not shown here) if there is something wrong with number, mass, or length. The -storeu- command has two variables in its tag. The first variable will get the numerical part of the student's response, and the second (dimens(1) in this case) is the starting point for receiving the dimensional information. Here are some examples of what will end up in num, dimens(1), and dimens(2) for various student responses:

student response	num	dimens(1)	dimens (2)
13.6 grams/cm^3	13.6	1	−3
13.6	13.6	∅	∅
13.6 cm-gm^2	13.6	2	1
13.6 kg/1∅cm	136∅	1	−1

Notice (in the third example) that a minus sign preceding a unit name is taken as a dash meaning multiplication, not subtraction. Note in the last example that "kg" brings in a factor of 1∅∅∅ relative to the basic unit (gm). Note also that, as usual, TUTOR does multiplication before doing division so that the "1∅ cm" is all in the denominator, with the result that we have (length)$^{-1}$. Similarly, "1/2 kg" will be taken to mean 1/(2 kg), *not* (1/2) kg. As mentioned earlier, it is best to point out this matter to the student at the beginning of the lesson.

Like -store-, the -storeu- judging command will flip TUTOR to the regular state (with a "no" judgment) if it cannot evaluate the student's response. The system variable "formok" can be used in a -writec- to tell the student *why* his or her response can't be evaluated. One example characteristic of responses involving units is "5 grams + 3 cm", which is absurd. You cannot add masses and lengths, and -storeu- will give up. On the other hand, the student can say "65 cm + 2 meter" and -storeu- will set num to 265, dimens(1) to ∅ (no mass), and dimens(2) to 1. As another example, "cos(3cm)" is rejected, but "cos(3cm/meter)" is accepted. The argument of most functions must be dimensionless. (Exceptions are "abs" and "sqrt".)

A related difficulty faces students unless they are specifically warned about "3+6 cm" being rejected by -storeu- (although it looks reasonable in context to the human eye). As far as -storeu- is concerned, however, the student is trying to add 3 "nothings" to 6 cm, and the units do not have the same dimensionality. For -storeu- this is as improper as "3 kg + 6 cm". Unfortunately, until -storeu- and TUTOR become more sophisticated, it will be necessary to give explicit instructions to the students that:

1) Multiplications are done before divisions (unless parentheses intervene), so that 1/2 kg does *not* mean (1/2) kg.
2) Responses such as "3 + 6cm" must be written rather as "(3+6)cm".

Note that these rules also apply in scientific journals and almost all textbooks, but your students may not be consciously aware of these standard rules. Given only these standard conventions, -storeu- will correctly handle an enormous variety of student responses.

While -storeu- can be used to get the number and dimensionality, the -ansu- and -wrongu- commands are primarily used to check for specific cases. Let us modify our sample unit to use these commands, which are like -ansv- and -wrongv- except for checking for correct units:

```
  .
  .
arrow    1618
storeu   num,dimens(1)
write    Cannot evaluate!
ansu     13.6 gm/cm3,.1
write    Good!
wrongu   13.6,.1
write    Right number, but give the units!
wrongu   (num)gm/cm3,.1
write    Right dimensionality, but wrong number!
wrongv   13.6,.1
write    Right number but wrong dimensionality.
no
writec   dimens(2)=-3,Length ok.,Length incorrect.
```

The -ansu- will make a match only if the dimensionality is correct and the -wrongu- checks for 13.6 (mass)$^{\varnothing}$ (length)$^{\varnothing}$, that is, no units given at all. The second -wrongu- looks for a number equal to (num), and finds it since it is the number the student gave (as determined by -storeu-). Therefore, this -wrongu- will match if the number is not 13.6 but the dimensionality is correct. The -wrongv-, unlike -wrongu-, is only concerned with the numerical element rather than the dimensionality. It is used here to check for responses such as "13.6 cm".

The -exact- and -exactc- Commands

It is occasionally useful (in special cases) to use a command *less* powerful than -answer- to judge a response. Suppose you are teaching the precise format required on some business form, and you want the student to type "A   B   C" *exactly*, with three spaces between the letters. A match to "answer   A   B   C" would occur no matter how the student separates the letters. One space, four spaces, a comma or a semicolon (any of these punctuations) are permissible separators as far as -answer- is concerned. Normally, this flexibility is beneficial to students because it keeps them from getting too hung up on petty details. If, however, it is the details that are important in a particular response, use an -exact- command. In the present case, the statement "exact   A   B   C" will be matched only if the student types *exactly* that string of characters: A, space, space, space, B, space, space, space, C.

The -answer- command does not permit punctuation marks in its tag, so that a response such as "a:b" must be judged with an -exact- command if the colon is important. While punctuation marks cannot appear in the tag of the -answer- command, the student can use them in a response. The -answer- command will treat all punctuation marks that the student uses as being equivalent to *spaces.* (As an alternative, the -change- command can be used to redefine the colon to be considered a "word" and not just as a punctuation mark, in which case the -answer- command can be used.)

It should be emphasized that it is easy to misuse the -exact- command. The student should normally be given considerable latitude in the form of his or her response, such as is permitted by the -answer-, -concept-, and -ansv- commands. The -exact- command should be used sparingly, and only for short responses. It may be important for the student to know the exact format of something that is as long as:

```
3 No. 6 screws/516-213-86xq-4:   New Orleans
```

In this case, it would certainly be preferable to have the student pick this correct form out of a displayed set of samples than to ask him or her to type it exactly. (Then, all the student would need to say is that item number 3 is the correct form.)

There is also a conditional form of the -exact- command, -exactc-. (The conditional -answer- command is called -answerc-.) In the case of the conditional form of the -do- command, the presence of commas tells TUTOR that the statement is conditional, so a -doc- command name is not needed. But -write-, -answer-, and -exact- may have tags which include commas, so the conditional command names must be different (-writec-, -answerc-, -exactc-).

The -answerc- Command: A Language Drill

The conditional -answer- command, -answerc-, may be used to create vocabulary or translation drills. Here is a sample unit which will give the student practice with Esperanto numbers:

```
unit     espo
next     espo
at       1812
write    Give the Esperanto for
randu    item,5                      $$ pick an integer from 1 to 5
at       2ø15
writec   item-2,one,two,three,four,five
arrow    2113
answerc  item-2;unu;du;tri;kvar;kvin $$ note semicolons
```

Each item in the -answerc- can be as complicated as the tag of an -answer- command. For example, "answerc select↕ <it,is,a> (right,rt) triangle, <it,is,a> three*sided (polygon,figure)↕↕ circle,ring" will accept either "rt triangle" or "three sided polygon" if "select" is −1, will accept nothing if "select" is zero, and will accept "circle" or "ring" if "select" is one or more. Note that items must be separated by a semicolon or by the -writec- delimiter. There is also a conditional -wrong- command, -wrongc-.

You might write yourself a similar unit to drill yourself on historical dates, capitals of nations, etc. The drill just shown has three defects: (1) it never ends; (2) you may see the same item two or three times in a row; and (3) no help is available if you get stuck. Let's revise the sample unit to have the following characteristics: it should present the five items in a random order but without repeating any item; any items missed will then be presented again; the student may press HELP to get the correct answer.

We will be using a random sequence of non-repeating item numbers such as:

4,2,1,5,3.

This is called a "permutation" of the five integers. The following sequence is another permutation:

2,5,3,1,4.

You can see that there is a large number (12∅) of different permutations of five integers. Correspondingly, there is a large number of different permutation sequences for presenting the drill to the student. Such sequences of non-repeating integers are quite different from the sequences we get from repeated execution of our "randu item,5", which produces sequences (with some integers repeating and some not showing up for a long time) such as:

3,2,4,4,1,5,1,2,4,3,5,5,2,etc.

We need some way of asking TUTOR to produce a permutation for us, rather than the kind of sequence produced by -randu-. This is done by telling TUTOR to set up a permutation of 5 integers ("setperm 5") from which to draw integers ("randp item") until the sequence is finished (indicated by "item" getting a value of zero). The -setperm- command actually sets up *two* copies of the permutation, and the "remove item" statement can be used to remove an integer from the second copy. (The -randp- draws integers from the *first* copy.) If we -remove- only those integers corresponding to items correctly answered on the first try, the second copy will contain only the difficult items (after completing the first pass over the five items). At this time, we can use -modperm- (which has no tag) to modify the first copy by shoving the second copy into the first copy. Having replenished the first copy with the difficult items we can use -randp- to choose these again.

Here is a form of the drill incorporating these ideas:

```
unit      begin
setperm   5                 $$ set up two copies of a permutation
jump      choose
*
unit      choose
calc      attempt⇐∅         $$ initialize number of attempts
randp     item              $$ pick an integer
jump      item>∅,espo,x     $$ jump if first copy not empty
modperm                     $$ use second copy if first copy empty
randp     item
jump      item>∅,espo,x     $$ jump if second copy not empty
```

```
at        2115
write     Congratulations!
          You finished the drill.
end       lesson               $$ end the lesson
*
unit      espo
next      choose
help      esphelp
at        1812
write     Give the Esperanto for
at        2Ø15
writec    item-2,one,two,three,four,five
arrow     2113
answerc   item-2;unu;du;tri;kvar;kvin
goto      attempt>Ø,q,x
remove    item                 $$ remove item if correct on first attempt
no
calc      attempt⇐attempt+1
*
unit      esphelp
calc      attempt⇐attempt+1    $$ count HELP as an attempt
at        1613
writec    item-2,unu,du,tri,kvar,kvin
end
```

We want to remove an item only if the student gets it right on the first try, which means "attempt" should be zero. The "goto attempt>Ø,q,x" means "goto a fictitious, empty unit 'q' if attempt is greater than Ø, else fall through." If we fall through, we remove the item ("remove item"). We increment "attempt" on each try (and also when help is requested) so that if the student has to see the answer, the item is not removed and will be seen again. Note that the student *is* required to type the correct response and cannot see this answer while he or she types, which gives the student additional practice on the difficult items.

Summary

This chapter has demonstrated an array of techniques for judging various types of student responses. There are -answer- and -wrong- (aided by -list-) for handling sentences composed from a relatively small vocabulary of words. There are -concept- and -miscon- (supported by -vocabs-) to handle dialogs involving a large vocabulary. The -match- and

-storen- commands can be used to pull out pieces of a student's response. The -storea- and -store- commands allow the student to specify alphanumeric or numeric parameters. There are -ansv-, -wrongv-, -ansu-, and -wrongu-, aided by "define student", for judging numerical and algebraic responses. The -exact- and -exactc- commands can be used when it is important that the response take a particular *precise* form. The -specs- command permits you to exercise various options associated with these commands and also provides a convenient marker of centralized post-judging processing. The *regular* -judge- command offers additional control over the judging process.

The construction of randomized drills using -setperm-, -randp-, -remove-, and -modperm- (and featuring the conditional commands -answerc- and -wrongc-) was also illustrated in this chapter.

It is hoped that you will read over this chapter occasionally in the course of writing curriculum materials. The TUTOR judging capabilities are extremely rich (because of the wide range of student responses that must be handled in order for lesson material to be successful). Reread appropriate sections of this chapter at a later time, when you need the details. For now it is sufficient to know what is available, and roughly in what form. You may find it helpful to think of the judging commands introduced in this chapter as making up two major classes: those used for handling words and sentences (-answer-, -answerc-, -list-, -concept-, -vocabs-, -match-, -storen-, -storea-, and -exact-), and those used for handling numbers and algebraic expressions (-ansv-, -define-, -ansu-, -store-, and -storeu-).