

# More About Judging

# 8

The previous chapter described the array of major response-judging features of the TUTOR language. We can now discuss the judging process in more detail, after which we will see how to treat responses that don't quite fit the categories of the previous chapter.

## Stages in Processing the -arrow- Command

The following is a summary of the several stages of processing involved when there is an -arrow- command.

- Stage 1 The -arrow- command is executed. The arrow is displayed on the screen, and a marker is set to remember the unit and location within the unit of this -arrow- command. Regular processing continues until a judging command is encountered, at which point there is a wait while the student types a response.
- Stage 2 The student presses NEXT or otherwise completes his or her response. TUTOR uses its -arrow- marker to start judging at the statement following the -arrow- command. Only judging commands are executed; all regular commands are skipped. Execution of a -specs- command sets a -specs- marker to remember the unit and location within the unit of this -specs- command.


- Stage 3 Some judging command terminates judging and successive regular commands are executed until a judging command is encountered, which ends this regular processing, even if we are several levels deep in -do-s. There is *no* “undoing”. An -arrow- or -endarrow- will also halt this regular processing without permitting “undoing”. (If no judging command terminates the judging phase, the end of a unit with no more “undoing” to do; an -endarrow-; or another -arrow- will end Stage 3 and make a “no” judgment.)
- Stage 4 If the -specs- marker has been set, regular processing begins at the statement following the last -specs- command encountered. (The -specs- marker is cleared.) This processing terminates in the same way as the regular processing of Stage 3. If the judgment is not “ok,” the -arrow- is not satisfied. The student must erase part or all of the response and enter a different response, which initiates Stage 2 again.
- Stage 5 The *search* state is initiated if there is an “ok” judgment. TUTOR again uses the -arrow- marker to start processing at the statement following the -arrow- command, this time in a search for another -arrow-. Only -join-s are executed, all other commands (regular or judging) are skipped during this search state. If an -arrow- command is encountered, TUTOR begins Stage 1 for this additional -arrow-. If an -endarrow- command is encountered, the search state ends and regular commands are processed. If neither -arrow- nor -endarrow- is encountered, the student can press NEXT to go on to the next main unit, having satisfied all the -arrow-s.

This all sounds rather complicated, written out in this way, but in most practical cases this structure turns out to be quite natural and reasonable. It is, nevertheless, useful to look at some unusual cases to further clarify the various processing stages.

### Repeated Execution of -join-

The following is an example of the repeated execution of a -join- in regular, judging, and search states (remember that -join- is similar to -do-):

unit	multy
calc	i←0

	arrow	1514
	join	$i \Leftarrow i+1, \text{ansdog}$
	endarrow	
	at	2514
	show	i
	*	
	unit	ansdog
	answer	dog
	write	Bowwow!

The conditional `-join-` has only one unit listed, so we will always join unit “ansdog” no matter what value the expression ( $i \Leftarrow i+1$ ) has. Upon first entering unit “multy”, we do the `-calc-`, the `-arrow-`, and the `-join-`, all in the regular state. This terminates at the `-answer-` command to await a student response. Note that  $i$  is now 1, due to the assignment ( $i \Leftarrow i+1$ ) contained in the conditional `-join-`. Suppose the student types “cat” and presses NEXT. TUTOR starts at the statement following the `-arrow-` and executes the `-join-` in the judging state (incrementing 1 to 2 in the process). No match is found for “cat”, so the student must give another response. Suppose the student now enters “dog”. TUTOR again starts judging just after the `-arrow-` and again executes the `-join-` (thus incrementing  $i$  to 3). This time there is a match to “answer dog” which changes the state from judging to regular. The “write Bowwow!” is executed, and the end of unit “ansdog” causes TUTOR to “undo” back into unit “multy”, where the `-endarrow-` signals the end of the statements associated with the `-arrow-`. Since we received an “ok” judgment, we are ready to search for any other `-arrow-s` that might be in unit “multy”. We return to the `-arrow-` one last time, this time in the search state. The `-join-` is executed to see whether there is an `-arrow-` command in unit “ansdog”, with the incidental result that  $i$  gets incremented to 4. No `-arrow-` is found in unit “ansdog” and we “undo” into the `-endarrow-` command, which changes us from search state to regular state. The `-at-` and `-show-` are executed and we get “4” on our screen, due to the quadruple execution of the `-join-`.

Aside from illustrating some consequences of the processing rules, this example should emphasize that using the assignment symbol ( $\Leftarrow$ ) in a conditional `-join-` may have unexpected results. Note that `-join-` is the *only* command with these properties, due to the fact that it is the only command executed in regular, judging, and search states. It is important that `-join-` be universally executed in this way so that you can join judging commands in the judging state and even `-arrow-` commands in the search state, not just regular commands in the regular state.

## Judging Commands Terminate Regular State

The rule that a judging command terminates the processing of regular commands is an important and general rule. We have seen that this must be true upon first encountering an -arrow- (the first judging command after the -arrow- makes TUTOR wait for a student response, since that judging command needs a response to work on). Let's see another instance of the rule:

```

.
.
.
arrow    1518
answer   dog
write     Bowwow
wrong     cat
write     Meow
wrong     horse
.
.
.


```

If the student says "dog", he or she gets a reply "Bowwow" and regular processing stops at the "wrong cat" because -wrong-, a judging command, terminates the regular state. Similarly, if the student response is "cat", the statement "write Meow" is the only regular statement which is executed. The judging commands delimit those regular commands associated with a match of a particular judging command. This delimiting effect is achieved because:

- 1) Regular commands are skipped in the judging state; and
- 2) The processing of regular commands ends whenever a judging command is encountered.

Now let's consider a slightly modified sequence:

```

.
.
.
arrow    1518
 join    dogcat
write     Meow
wrong     horse
.
.
.

```


unit	dogcat
answer	dog
write	Bowwow
wrong	cat

Supposedly, the “join dogcat” will act as though the statements of unit “dogcat” were inserted where the -join- is, which should make this modified version equivalent to the earlier version. Indeed, the rule that a judging command terminates the processing of regular commands does make the two versions equivalent, as we will show. Remember, in this discussion, that -join- is the same as -do- except for the universal nature of -join-.

Suppose the student types “dog”. We start just after the -arrow-, in the judging state. The -join- is executed and we find a matching “answer dog” which ends judging and puts us in the regular state. The “write Bowwow” is executed. The statement “wrong cat” is encountered next. The judging command -wrong- stops the processing of regular commands and also prevents coming out of the joined unit. Even though we are one level deep in -join-s, TUTOR will *not* “unjoin” and the “write Meow” which follows the “join dogcat” will *not* be executed. What *will* happen is just what happens in the earlier version: we have an “ok” judgment which causes the search state to be initiated at the -arrow- (there was no -specs-). Thus, the two versions operate in identical manners because the -join- acts like a text insertion. Note that a response of “cat” will get a reply “Meow” because there is no judging command following the “wrong cat” (and a normal “undo” is performed at the end of unit “dogcat”).


This last example illustrates the importance of the rule “a judging command terminates the regular state.” It is this rule which insures that -join- (or -do-) will act like a text insertion.

In the discussion of the -goto- command in Chapter 6, we saw that a -goto- in a done unit destroys the strict text insertion character of the -do-. This is true in the present context as well. Suppose we insert a -goto- in unit “dogcat” (any -goto- will do, we’ll use a “goto q”):

unit	dogcat
answer	dog
write	Bowwow
 goto	q
wrong	cat

The student enters “dog” and we do unit “dogcat” where the match to “answer dog” flips us from the judging to the regular state. The regular

commands `-write-` and `-goto-` are executed. (Note that `-goto-`, like `-do-`, is only regular whereas `-join-` is universal, being executed not only in regular but in judging and search states.) The execution of the `-goto-` prevents TUTOR from encountering the “wrong cat” which previously terminated the regular state. We have run out of things to do in unit “dogcat” and are one level deep in `-do-s`. TUTOR, therefore, “undoes” and executes the “write Meow” which follows the “join dogcat”! The student will see “BowwowMeow” on the screen. If, on the other hand, we replace the “join dogcat” with the statements contained in unit “dogcat” we would have:

```
.  
. .  
. .  
    arrow    1518  
    answer   dog  
    write    Bowwow  
 goto      q  
    wrong    cat  
    write    Meow  
    wrong    horse  
    .  
    .  
    .
```


and a response of “dog” would merely cause “Bowwow” to appear on the screen, *not* “BowwowMeow”.

We have again seen that a `-goto-` in a done unit can cause the `-join-` operation to behave differently from a text insertion. We get different effects depending on whether we `-join-` such a unit or put that unit’s statements in place of the `-join-` statement. You can avoid confusion by not using `-goto-` commands in “done” or “joined” units which contain `-arrow-` commands or judging commands.

### The `-goto-` is a Regular Command


Since the `-goto-` command is a regular command, it is skipped in the judging and search states. Here is a sequence of commands which illustrates the fact that the `-goto-` is skipped in the *judging* state:

```
.  
.  
.
```

	arrow	1612
	goto	dogcat
	*	
	unit	dogcat
	answer	dog
	write	Bowwow
	wrong	cat

When the `-arrow-` is first encountered, an arrow is displayed on the screen at 1612. TUTOR continues in the regular state and executes the `-goto-`. The `-answer-` in unit “dogcat” ends this regular processing to await the student’s response. Suppose the student types “dog” and presses NEXT. TUTOR starts judging just after the `-arrow-`, skips the regular `-goto-` command, and finds no judging commands at all. The student’s response gets a default “no” judgment. The `-goto-` should be replaced by a `-join-` so that unit “dogcat” will be attached in the judging state.

Similarly, the following is an erroneous sequence which illustrates the fact that the `-goto-` command is skipped in the *search* state:

.		
.		
.		
	arrow	1612
	specs	bumpshift
	answer	dog
	goto	another
	wrong	cat
	*	
	unit	another
	arrow	2514
	answer	wolf

The student responds to the first `-arrow-` with “dog” and matches the “answer dog”, which switches the processing from the judging state to the regular state. The `-goto-` is executed, and in unit “another” we encounter an `-arrow-` command. This `-arrow-` command terminates the regular processing just as a judging command would. The `-specs-` marker was set, so we will now execute any regular commands following the `-specs-` command (there are none in this example). Since the student’s response was “ok”, the search state is now initiated. TUTOR starts at the “arrow 1612” looking for another `-arrow-` command. The `-specs-`, `-answer-`, `-goto-`, and `-wrong-` are skipped in the search state, and we come to the end of the unit without finding an `-arrow-`. Thus the `-goto-` did not

succeed in attaching a second -arrow-. If the -goto- is replaced by a -join-, the “wrong cat” will be associated with the *second* -arrow- (2514). This is due to the text insertion nature of the -join-, which interposes the statements of unit “another” between the “answer dog” and the “wrong cat”. One correct way to write this sequence is shown below:

```

.
.
.
arrow      1612
specs      bumpshift
answer     dog
wrong      cat
endarrow
goto       another    $$ or “do another”
*
unit       another
arrow      2514
answer     wolf

```

The -goto- or -do- placed after the -endarrow- will not cause any problems because the search state has been completed, and the -endarrow- flips us from the search state to the regular state.

Considerations of this kind suggest that some care must be exercised when using -join- or -do- to attach units containing -arrow- commands. To avoid unpredictable results follow these two rules:

- 1) A unit attached by -join- or -do- which contains one or more -arrow- commands *must* end with an -endarrow- command. This insures that the unit will end and “undo” in the regular state. (It is permissible to have regular commands following the -endarrow-.)
- 2) The attached unit containing one or more -arrow- commands *must not* contain any -goto- commands. (A -goto- can make TUTOR fail to see the -endarrow- or a judging command so that a premature “undo” occurs.)

If these two rules are followed, the -join- or -do- will act precisely as though you had inserted the statements of the attached unit where the -join- or -do- was. Here are examples of good and bad forms:

		<u>GOOD</u>		<u>BAD</u>	
		unit	response	unit	response
...		answer	apple	answer	apple
join	response	do	newton	goto	newton ( <u>Don't</u> use -goto- here)
...		wrong	pear	wrong	pear



GOOD (continued)  
 write Wrong fruit.  
 endarrow

BAD (continued)  
 write Wrong fruit.  
 (Do use -endarrow- here)

### Interactions of -arrow- with -size-, -rotate-, -long-, -jkey-, and -copy-

When an -arrow- command is performed, several things happen. An arrow character is displayed on the screen, cuing the student to enter a response. A note is made of the unit and location within that unit of the -arrow- command so that TUTOR can return to this marked spot when necessary. Even the trail of -do-s (and/or -join-s) which brought TUTOR to this -arrow- command is saved, so that each restart at the -arrow- will be at the appropriate level of -do- relative to the main unit. The current settings of -size- and -rotate- are saved, to be restored each time so that you can write a size-3 reply to a student's incorrect response without affecting the size of his or her corrected typing. In other words, response-contingent settings of -size- and -rotate- are temporary, whereas in other circumstances they are permanent until explicitly changed:

```
.
.
.
.
size      2
rotate    Ø
arrow      1718
answer    dog
size      4
rotate    3Ø
write     Woof!
answer    wolf
endarrow
at        2218
write     This is in size 2, rotate Ø.
.
.
.
```

The last writing appears in size 2, rotate Ø despite the size 4, rotate 3Ø, that were contingent on the student's response, "dog." When the search state is initiated, the original size and rotate settings are restored.

Similarly, if “dog” had been judged wrong, the student’s revised typing would have been in size 2, not 4, because the original size and rotate are restored before waiting for the student’s revised input.

Executing an -arrow- command has other important initialization effects:

- 1) A default response limit of 150 characters is set. The student cannot enter a response longer than 150 characters (including “hidden” characters such as shift-codes and superscripts). This can be altered by following the -arrow- command with a -long- command to change this to as much as 300. If this is a “long 1,” judging will commence as soon as the student types one character. If more than 1 is specified, the student is prevented from entering more characters and must press NEXT to initiate judging, unless a “force long” statement has appeared in the unit.
- 2) A default specification of “judging keys” is set. In most cases, the NEXT key is solely responsible for starting the judging process. However, there are two other possible ways to begin judging: (1) hitting the limit with a “force long”; or (2) if there is a “long 1”, typing one character will begin judging. This can be altered by following the -arrow- command with a -jkey- command to specify *additional* judging keys (NEXT is *always* a judging key). One example is “jkey data,help” which would make the DATA and HELP keys equivalent to the NEXT key at this arrow.
- 3) A default specification is set to disable the COPY key. The -arrow- command can be followed with a -copy- command to specify a previously stored character string to be referenced with the COPY key. An example is “copy v51,v3”, where v51 is the start of the character string and v3 is the number of characters. This way of specifying a string of characters is the same as the scheme used with -storea- and -showa-.

Some explanation of the COPY and EDIT keys is required. The EDIT key is *always* available for the student to use in correcting his or her typing. Pressing the EDIT key the first time erases all typing, after which each press of the EDIT key brings back the typing one word at a time. This makes it easy to make corrections and insertions without a lot of retyping. Each press of the COPY key, on the other hand, brings in a word from the character string specified by the -copy- command, as opposed to bringing in the student’s own typed words with the EDIT

key. One example of the use of the COPY key is seen in the PLATO lesson editor. In this case, you as an author can use the COPY key in insert or replace mode to bring in portions of a preceding line without having to retype. The COPY key must be specifically activated by a -copy- command, but the EDIT key is always usable, unless you specify a -long- greater than the normal limit of 150. (To use the EDIT key on responses longer than 150 characters requires you to furnish an edit buffer through an -edit- command.)

The -long-, -jkey-, and -copy- commands all override default specifications set by the -arrow- command. They can be thought of as modifiers of the -arrow- command. If they are to have an effect on the student's first response, they not only must follow the -arrow- command but must *precede* any judging commands:

```

.
.
.
arrow 1518 $$ sets default values
{ jkey  help
  copy  cstring,ccount } These commands alter the default values.
{ long  15
  -specs- or -answer- or -store- or any other judging command

```

If -jkey-, -copy-, or -long- came *after* the first judging command, the -arrow- defaults would hold for the first response because the modifying command would not have been executed yet.

### Applications of -jkey- and -ans-

Use of the -jkey- command is well illustrated in the case of providing help to the student (through the HELP key) *without leaving the page*. (This is an alternative to the more commonly used -helpop- command described in Chapter 5.) If giving help requires an entire screen display, or a whole sequence of help units, it is best to use a -help- command to specify where to jump if the student presses HELP. The screen is then erased automatically to make room for the help page (unless the original base unit had an "inhibit erase" in it). On the other hand, sufficient help might consist merely of a brief comment or some additional line-drawings on the present page. A convenient way to provide such help without leaving the page is:

```

.
.
.
arrow      1815
jkey       help
answer     cat
no
write      Hint: it meows . . .

```

The statement “jkey help” makes the HELP key completely equivalent to the NEXT key. If the student presses HELP, judging is initiated, the student’s (blank) response does not match “cat”, and he or she gets “Hint: it meows . . .”. Without the -jkey- command, the HELP key would be ignored (which would be unfortunate). It is a very good idea to have the HELP key do *something* at all times so that the student can come to rely on help being available.

In this example, the student will get the same assistance whether he or she presses HELP or types “dog” followed by pressing NEXT. We could give different kinds of assistance in these two cases by changing the -write- statement to a -writec-:

```

.
.
.
arrow      1815
jkey       help
answer     cat
no
writec     key=help,Meow?,The answer is cat.

```

The system variable “key” always contains a number corresponding to the last key pressed by the student. In this case the last key will either be HELP or NEXT. If the student presses HELP, the logical expression “key=help” will be true (–1) and the student gets the reply “Meow?” But, if the student presses NEXT, then the logical expression “key=help” is false (0) and the student gets “The answer is cat.” The lower-case word “help” is defined by TUTOR to mean (in a calculational expression) “the number corresponding to the HELP key.” Other similarly defined names include next, back, and help1 (for shift-HELP).

The following is another way of writing the same sequence:

```

arrow    1815
jkey     help
no              $$ terminate judging
judge    key=help,x,continue
write    Meow?
answer   cat
no
write    The answer is cat.

```

If key=help, we “fall through” the -judge- command and write “Meow?” If the key is not equal to help (that is, the student pressed NEXT), a “judge continue” is performed to return to the judging state. The “write Meow?” is skipped since -write- is a regular command. If the response does not match “cat”, the student will get the message “The answer is cat”. As usual, there are many ways in TUTOR to do the same thing! In a particular situation one scheme may be more appropriate than another.

There is an ANS key on the keyset which is often used to let students skip through material by just pressing ANS:

```


.
.
.
arrow    1817
jkey     ans
ok
judge    key=ans,x,continue
write    The answer is cat
answer   cat
.
.
.

```

Since the ANS key generates an ok judgment here, the student will move on immediately to the next arrow or unit without having to type the correct answer. This procedure could best be utilized when the student is in the review mode. That is, you might define “review=v1”, zero it initially, and set it to -1 only after the student has gone through the material once under his or her own power. With the following structure, the student will be able to use the ANS key only when reviewing the material:

```
.  
.   
.   
arrow 1817  
do review,jans,x  
ok  
judge key=ans,x,continue  
.   
.   
.   
unit jans  
jkey ans
```

Another way to activate the ANS key for the student is to use the -ans- command with a blank tag.

```
.  
.   
.   
arrow 2123  
 ans  
write The answer is cat.  
.   
.   
.   
.
```

In the above example, the single -ans- command is equivalent to the following:

```
jkey ans  
ok  
judge key=ans,x,continue
```

The -ans- command is a judging command and must be the *first* judging command after the -arrow-. When it is first encountered, it sets up ANS to be a judging key, and it is matched only if the ANS key is pressed. If the -ans- command is used only to provide a kind of help, but not to let the student pass on to the next item, put a “judge wrong” after the -ans- command.

In many places you may do specific things in response to the ANS and HELP keys. Elsewhere in the lesson it is appropriate merely to utilize these keys so that *something* will happen when they are pressed. Just put “jkey help,ans” after each such -arrow-. The student will then

get (at least) whatever reply you give him or her after the universal -no- that catches all unrecognized responses. Certainly, every -arrow- should provide some kind of feedback to unrecognized responses or the student will become perplexed. The “jkey help,ans” will further insure that a reasonable response to the student’s input is always forthcoming. Without this -jkey- statement, nothing would happen when the student presses ANS or HELP.

An additional procedure is advisable. Often a student will press NEXT an extra time, perhaps because he or she hadn’t noticed that a response was to be typed. This blank response, consisting only of a NEXT key, will probably get judged “no” at most arrows, which requires an additional NEXT (or ERASE) to clear the “no” judgment before typing a response. This can get confusing. In most cases it is best simply to ignore blank responses by means of the statement “inhibit blanks”, which can be put in the -imain- unit (see Chapter 5). This statement causes blank-NEXT inputs to be ignored, but other blank inputs such as HELP or ANS are not ignored.

Use a -join- to insert recurring statements after an -arrow-:

```

.
.
arrow    1917
join     anshelp
answer   cat
.
.
.
unit     anshelp
inhibit   blanks    $$ or the -inhibit- could be in an -imain- unit
jkey     ans,help

```


Placing “join anshelp” after each -arrow- will insure that extra NEXT keys are thrown out (while responses involving ANS or HELP keys, will fall through to whatever reply you give to unrecognized responses). Note that you must use -join-, not -do-, to attach unit “anshelp” if you add any judging commands to that unit.

Just as the -imain- command can be used to specify a unit to be done at the beginning of each new main unit, there is an -iarrow- command (“initialize arrow”) which can be used to specify a unit to be joined after every -arrow-. With the statement “iarrow anshelp”, it is unnecessary to write “join anshelp” after every -arrow- command. Unit “anshelp” will be joined automatically after every -arrow-.

## Modifying the Response: -bump- and -put-

It is possible to delete characters from the judging copy of the student's response by using the -bump- command:


```

arrow 1812
 bump as3 $$ delete all a's,s's, and 3's
answer rdvrk

```

This -answer- will be matched if the student types "33 aardvarks" because the -bump- command reduces the judging copy of the response to "rdvrk." The original response is not altered and can be recovered with a "judge rejudge". Also, the screen display is unaffected: the student still sees "33 aardvarks" on the screen just as he or she typed it. On the other hand, all judging commands following the -bump- are affected since they all operate on the judging copy (not on the original response). For example, a -storea- following the -bump- would give you "rdvrk". Here is another example:

```

define cfirst=v1,csecond=v2
      first=v11,second=v21
unit   conson
at     913
write  Type anything, and I'll
      remove the vowels:
arrow  1309
long   100      $$ from v11 to v21 is 100 characters
storea first,cfirst<-jcount
 bump aeiou
storea second,csecond<-jcount
ok
write  You typed <a,first,cfirst>.
      Remove vowels: <a,second,csecond>.
      You used <s,cfirst-csecond> vowels.

```

Note that "cfirst" is the number of characters (including hidden characters such as shift characters) in the original response, whereas "csecond" is the number of characters after the -bump- has removed the vowels. This is a true count since "jcount" always has an up-to-date character count of the judging copy, as influenced by -bump- and related opera-



tions. (You may recall that “specs bumpshift” also affects “jcount” by removing shift characters.) Suppose the student types “Apples taste funnier”. In this case, the student will get the reply:

You typed Apples taste funnier.  
Remove vowels: Ppls tst fnnr.  
You used 7 vowels.


The reason that the word “Apples” turns into “Ppls” with a capital “P” is that a capital “A” is really a shift character followed by a lower-case “a”. With the “a” bumped out, the shift character stands next to the “p”, making a capital “P”.

While the -bump- command will delete characters, the -put- command will change particular strings of characters:

```

.
.
.
arrow 1218
put cat=dog
put rat=mouse
storea first,jcount
ok
showa first,jcount

```



All occurrences of “cat” change into “dog”, and all occurrences of “rat” change into “mouse”. Suppose the student types “Scattered cats scratch rats”. The reply will be “Sdogtered dogs scmousech mouses”!

Both -bump- and -put- are judging commands. They operate on the student’s response. Like all judging commands, they stop processing when encountered during the processing of regular commands. The -put- command has a property similar to -store- in that it can terminate judging with a “no” judgment if it cannot handle the student’s response:

```

.
.
.
arrow 1218
put cat=enormous
write Too many cats!
ok
.
.
.

```

If the student has many “cats” in his or her response, the -put- may cause “jcount” to exceed the 150-character response limit. In this case, it changes to the regular state, and the student gets the message “Too many cats!” This regular -write- command normally is skipped, since we’re in the judging state.

The following is an equivalent form of -put- which is often easier to read:

```
put   cat=dog
putd  /cat/dog/
putd  ,cat,dog,
```

All three of these statements are equivalent. The -putd- (d for delimiter) takes the first character as the delimiter between the two character strings. Other examples of its use are:

```
putd  /=/equals/  $$ convert = sign
putd  /  //       $$ remove all spaces
```

It is also possible to change variable character strings by using -putv- (v for variable):

```
putv   first,cfirst,second,csecond
      string and count      string and count
```

When you combine -put- and -bump- commands, you must be careful about how you arrange them. For example, the following sequence is nonsense:

```
bump  a
put   cat=dog
```

With all a’s bumped the -put- will not find any cat’s. Similar remarks apply to sequences of -put- commands.

The -bump- command looks for single characters, so “bump B” will not merely bump capital B’s. All shift characters will be bumped as well as lower-case b’s. In other words, “bump B” is really “bump shift-b”. If you want to eliminate only capital B’s, use “putd /B//”. This will find occurrences of the string of characters “shift-b” and replace this string with a zero-length string, thus deleting the B.


The main purpose of -bump- and -put- is to make minor modifica-

tions to the student's response and convert it into a form which can be handled by standard judging commands. For example, the word-oriented judging commands (-answer-, -match-, -concept-, etc.) cannot find *pieces* of words. Suppose that for some reason you need to look for the fragment "elect", and you don't care whether this appears in the word "selection" or "electronics" or "electoral". Do this:

```

.
.
.
arrow      1723
specs      okextra
putd       /elect/ elect /
answer     elect
.
.
.

```



The -putd- is used here to put spaces before and after the string "elect" so that it stands out as a separate word. You could also use the values of "jcount" before and after executing the -putd- to determine whether "elect" was present. The number of times it appeared could also be determined from these values. The value of "jcount" will increase by two for each insertion of two extra spaces.

## Manipulating Character Strings

The judging commands -bump- and -put- operate on the judging copy of the student's response. It is sometimes useful to manipulate other strings of characters with -pack-, -move-, and -search-. These commands are regular commands, not judging commands. Like -showa-, they operate on stored character strings, not the judging copy of the student's response. These commands are mentioned here because they are often used in association with the analyzing of student responses. In particular, the judging command -storea- can be used to get the response character string. It can then be operated on with -move- and -search-. Finally, the altered character string can be loaded back into the judging copy with the judging command -loada- (load alphanumeric; the -loada- command is precisely the opposite of -storea-). Since this section deals with a rather esoteric topic, you might just skim through it now to get a rough idea of what character string manipulations look like. If you later find a need for such operations, you should study this section again.

Here is an example of a -move- statement:

**move v3,5,v52,21,8**

This means “move 8 characters from the 5th character of the string that starts in v3 to the 21st character of the string that starts in v52.” The 21st through 28th characters of the v52 character string are replaced by the 5th through the 12th characters of the v3 character string. The v3 character string is unaffected. In other words, -move- has the form:

**move string1,start1,string2,start2,#characters moved**

If the number of characters to move is not specified, one character will be moved.

Here is an example of the use of -move-. Suppose the student types “ $x+4y = y-3$ ”, and we want to convert this into the form “ $x+4y-(y-3)$ ” before using -store- on it. Assume “str” has been defined:

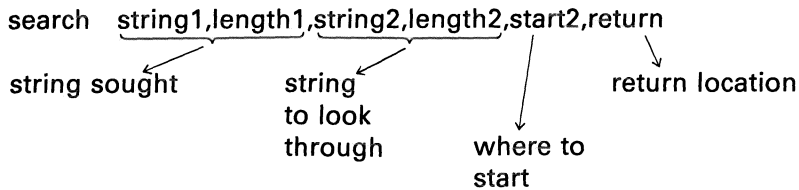
```

.
.
.
arrow 1812          $$ x+4y=y-3
putd   .=.-(.       $$ x+4y-(y-3)
storea str,jcount
ok
{move  '}',1,str,jcount+1  $$ x+4y-(y-3)
{judge continue          $$ to do judging -loada-
{loada str,jcount+1
store  result
ok
write  Subtracting the right side of
        your equation from the
        left side gives <s,result>.

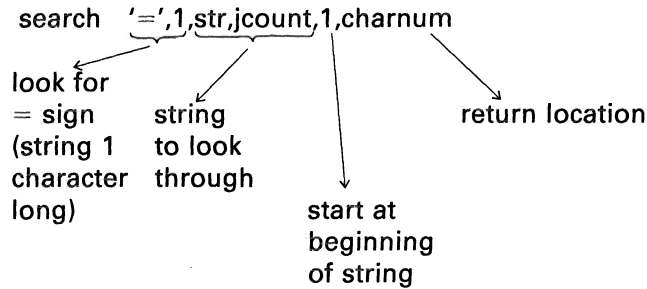
```

In the -move- command the parenthesis within single quote marks, ')', means a character string one character long consisting of a right parenthesis. Similarly, 'dog' would denote a character string consisting of d,o, and g. Character strings up to ten characters in length may be described this way, using single quote marks. The -move- command shown above moves the first character of ')', which is just a right parenthesis, to the (jcount+1)th character position in “str”. This effectively appends a right parenthesis to the student’s character string (as modified by the -putd-). The -loada- command moves the final character string into the judging copy so that -store- can operate on it. Note carefully the switches from the judging state to the regular state and back again.

The `-search-` command is used to look for occurrences of specific character strings. It has the form:



Suppose we use `-storea-` to place the unaltered student response “`x+4y=y-3`” in “`str,jcount`”. Then use:



This `-search-` command will set the variable “`charnum`” to 5, since the equal sign is the 5th character in “`x+4y=y-3`”. If the search is unsuccessful, “`charnum`” is set to -1. As further illustration of `-move-` and `-search-`, let’s rewrite our earlier sequence without the `-putd-`:

```

arrow  1812
storea  str,jcount
ok
search  '=' ,1,str,jcount,1,charnum
* Now make room for the -( :
move    str,charnum+1,str,charnum+2,jcount-charnum
*Next insert the -( :
move    '-(',1,str,charnum,2          $$ move 2 characters
* Append the ) :
move    ')',1,str,jcount+2
judge   continue
loada   str,jcount+2
store   result
ok
  
```

The `-search-` finds the equal sign. The first `-move-` moves the latter part of the string to make room for the insertion of `'-('`. The second `-move-` makes the insertion which overwrites the characters (`=y`) which were there originally. The third `-move-` appends the `'`. Normally, the `-search-` would be followed by a `"goto charnum,noeq,x"` to take care of the case where the student did not use an equal sign, in which case `"charnum"` would be `-1`.

The single quote marks can be used to specify character strings up to ten characters long. Longer character strings can be placed in variables with a `-pack-` command:

```
pack v11,v3,abcdefghijklmnopqrstuvwxyz
      ↙           ↘
string location character count
```

This packs a character string 26 characters long into `v11` and following variables. The character count (26 in this case) is placed in `v3`. Since each variable holds ten characters, `v11` and `v12` will be full while `v13` will have the last six characters. The `-pack-` command might be considered analogous to `-storea-`, since both place character strings in variables. In the case of `-storea-`, the total character count can be gotten from the system-defined variable `"jcount"`. Here is another example:

```
pack v12,v1,H2SO4
...
showa v12,v1
```

This will display `"H2SO4"` on the screen. The character count in `v1` will be ten, including three shift codes and two subscripts. The character string `H2SO4` is actually composed of shift, h, subscript, 2, shift, s, shift, o, subscript, 4. The character count portion of a `-pack-` command can be left blank, as in `"pack v12,,dog"`, the result of which could be displayed later with the statement `"showa v12"`. It is possible to embed `"show"` commands in a `-pack-` statement:

```
pack string,count,There are $(s,total) left.
```

There is also a conditional form, `-packc-`, analogous to `-writec-`:

```
packc cond,string,count,dog,cat,horse,cow
      ↙           ↙   ↘   ↘   ↘
conditional -1  0  1  ≥2
expression
```

There are other string-oriented commands. For example, `-clock-` will get the time, `-date-` gets today's date, `-name-` gets the (18-character) name the student is registered under, and `-course-` gets the course the student is registered in. These commands are used in the following illustration:

```
.
.
.
name    v1  $$ v1 and v2 for name
course  v3
clock    v4
date     v5
write    Hello! Your name is<a,v1,18>.
          You are registered in <a,v3>.
          The time is <a,v4>.
          The date is <a,v5>.
```

Suppose the student is registered as "sam nottingham" in a course "french4." It is 10:45:37 PM (22:45:37 on a 24-hour clock) on June 3, 1974. The student will receive this display:

```
Hello! Your name is sam nottingham.
You are registered in french4.
The time is 22.45.37.
The date is 06/03/74.
```

All of these commands, `-name-`, `-course-`, `-clock-`, and `-date-`, simply place the requested character/string in the specified variable for use in a `-showa-`.

The `-clock-` command produces a *character string*. In addition, there is a system variable "clock" which may be used in calculational expressions. It holds the *number* of seconds of a daily clock to the nearest thousandth of a second, and is convenient for calculating the amount of time spent in a section of a lesson.

The `-date-` command also produces a *character string*. There is also a `-day-` command which produces a *number* corresponding to the number of days elapsed since January 1, 1973. This number of days and fraction of a day is accurate to one-tenth of a second.

The TUTOR judging commands offer a great deal of power. We have seen that the judging commands `-bump-` and `-put-` together with the regular string-oriented commands `-move-`, `-search-`, and `-pack-` can be used to change an otherwise intractable response into a form which can be handled with TUTOR judging commands. This is a useful scheme as

long as only minor modifications are required. However, if major modifications of the response are required in order to be able to use TUTOR judging facilities, it is usually simpler to “do your own judging.” That is, get the student’s response with a `-storea-` and then analyze it with string-oriented commands, together with the additional calculational machinery described in Chapter 9. You might not even want to use the built-in marker features of the `-arrow-` command, with the associated returns to the `-arrow-`, when there is a “no” judgment. In such circumstances you might write a subroutine to be used in place of `-arrow-` commands, which merely collects the student’s response:

```
unit      arrow(apos)
arrow     apos
storea   sstr,scnt←jcount
specs     nookno
ok
endarrow
```

Instead of writing “arrow 1815” with associated judging commands you would then write:

```
.
.
do arrow(1815)
calc,move,etc. to do your own judging
.
.
```

Naturally, this course of action is advisable only if you are trying to analyze responses which have a form very different from those classes of responses which can be handled well by TUTOR judging commands.

### Catching Every Key: `-pause-`, `-keytype-`, and `-group-`

Occasionally, it is useful to process individual keypresses without waiting for a NEXT key. We have already discussed such typical examples as moving a cursor and choosing a topic from an index. These examples used a “long 1” with an `-arrow-` in order to catch each keypress. There is another way to do this, involving the `-pause-` command which was introduced in Chapter 2 in connection with creating displays, particularly timed animations. As was pointed out in the discus-



sion of the -jkey- command in the present chapter, the system variable “key” contains a number corresponding to the most recent key pressed by the student. For example, if the student presses the letter “d”, the system variable “key” will have the numerical value 4 (since d is the 4th letter in the alphabet). Putting these notions together, we have the following kind of structure:

```
write  Press “d”, please.
pause
writec key≠4,You didn’t press d.,Good!
```

The blank -pause- statement (“blank” in the sense of having no tag) causes TUTOR to wait for the student to press a key. Any key will cause TUTOR to move past the -pause- to the next statement.

In the example shown, the -pause- is followed by a -writec- conditional on “key≠4”. This -writec- can be written in more readable form by replacing the “4” with a “d”:

```
writec key≠“d”,You didn’t press d.,Good!
```

Enclosing the d with (double) quote marks is taken in calculational expressions to mean the number 4. Similarly, (v3<“z”) will assign the value 26 to v3. If the student presses 0 or 1, “key” will have the numerical value 27 or 28 respectively. That is, the 26 letters are followed by the numbers 0 through 9, then come various punctuation marks. If the student presses the plus key, “key” will have the numerical value “+”, which happens to be 37.


If the student presses a capital D, “key” will have the value 64+“d”, or 68. The shifted or upper case letters have “key” value 64 greater than the corresponding lower-case letters. Caution: some common keys such as parentheses have key numbers smaller than 64 despite requiring the shift key to type them. The most commonly used characters (lower-case letters, numbers, and common punctuation marks) have key numbers less than 64, independent of whether they are typed using the shift key. As for the function keys (NEXT, BACK, HELP1, etc.), we have seen (in connection with the -jkey- command) that the corresponding key numbers are given by next, back, help1, etc., as in:

```
goto key=help1,yes,no
```

No quote marks are used for the function keys.

A more convenient way to determine which key has been pressed is to use a -keytype- command. Consider a cursor-moving procedure:

```

define    num=v5,x=v1,y=v2,dx=10,dy=10
unit      cursor
pause
 keytype  num,d,e,w,q,a,z,x,c
goto      num,cursor,x
.
.
calcs     num-1,y<=y,y+dy,y+dy,y+dy,y,y-dy,y-dy,y-dy

```

The `-keytype-` command searches through the listed keys (d, e, w, q, a, z, x, and c in this case) and, similar to the `-match-` command, sets “num” to -1 (if the key is not found in this list) or to 0, 1, 2, 3, etc. (if it is found). If the student presses d, “num” will be set to 0; if the student presses c, “num” will be 7; and if he or she presses D, “num” will be set to -1. The `-goto-` statement effectively causes all unlisted keys to be ignored.

Note that no quote marks are used in specifying keys in a `-keytype-` command. Capital letters and function keys may also be listed:

```
keytype  v3,a,A,b,B,next,data,timeup
```

While the `-keytype-` command is most often used in conjunction with a `-pause-` command, it can also be used in association with an `-arrow-` command or any time that you want to find out which key was pressed most recently. The function key `timeup` is one generated by TUTOR when a timing key is “pressed” as the result of an earlier `-time-` command or timed `-pause-` command (see Chapter 2).

Just as the `-list-` command can be used to specify a set of synonymous words and numbers for use in `-answer-` and `-match-`, so there is a `-group-` command available for specifying synonymous keys for use in a `-keytype-` command:

```

define    keynum=v23,algkey=v24
group     algebra,x,y,z
.
.
.
keytype   keynum,a,b,algebra,help
           ↓ ↓      ↓      ↓
           0 1      2      3

```

If the student presses any of the keys x, y, or z, the variable “keynum” will be assigned the value 2. An additional `-keytype-` command can be used to separate members of a group:

```

keytype  keynum,a,b,algebra,help
goto     keynum,none,ua,ub,alg,somehelp
.
.
unit     alg
keytype  algkey,x,y,z
.
.

```

Some particularly useful `-group-` definitions are built-in. Without specifying these definitions with your own `-group-` commands, you can (in a `-keytype-` command) refer to these groups in the following ways:

```

alpha    all 52 lower-case and upper-case letters
numeric  0 through 9
funct    function keys (next,help,etc.)

```

An example of the use of these built-in groups might be “`keytype v45,funct,a,b,c`”. You can also use previously defined or built-in groups to define new groups:

```

group  mine,a,b,c,help
group  ours,mine,d,e,f
group  all,A,B,C,ours,numeric,funct

```

It is important to note that if you use a `-pause-`, the key pressed will not cause the associated character to appear on the student’s screen. You are in *complete* control. You may write something on the screen or not, as you choose. Only if you use an `-arrow-` will the standard key display take place (with the associated ERASE and other standard typing features available). Similarly, if you press HELP, you will not automatically branch to a unit specified by a previous `-help-` command, because a blank `-pause-` gives you *every* key, function key or not.

There is a variant of the `-pause-` command which is usually more useful than the blank `-pause-`. You can define which keys are to be accepted, and all other keys will be ignored:

```

.
.
next    umore
help    discuss
data    tables

```

(Continued on the next page.)

```
.  
. pause keys=d,D,next,term,help,help1  
. .  
. .
```

Any key not listed here is completely ignored, as though the student had not pressed it. Of the function keys listed, the HELP key will take the student to unit “discuss”, since you have already specified what you want the HELP key to do. Note that this is not possible with a blank -pause- which catches *all* keys. Similarly, what the TERM key will do has been predefined (the student will be asked “what term?”). But the DATA key will be ignored since it is not listed in the -pause- statement, and the student cannot reach unit “tables” with the DATA key until he or she has passed the -pause-. Pressing d, D, NEXT, or HELP1 will take the student past the -pause-. The NEXT key is rather special here in that the preceeding specification “next umore”, unlike “help discuss”, tells TUTOR what to do when the present main unit has been *completed*. Thus, pressing NEXT here takes us past the -pause- instead of branching us immediately to a different unit as HELP does.

You may prefer *not* to ignore the HELP key *nor* to use it to access unit “discuss”. In this case, the statement “help discuss” must *follow* the -pause- statement, or a “help q” must precede the -pause- in order to quit specifying a help unit.

## Touching the Screen

Most PLATO terminals have “touch panels” which make it possible for the student to respond by touching the screen. For example, a language drill might show the student pictures of various animals and ask the student to point to the dog. You need a way to tell at which part of the screen the student pointed. This is most easily done with -pause- and -keytype- statements, as in the following example:

```
pause keys=touch  
keytype num,touch(1215),touch(100,200)
```

The first statement, using the built-in group “touch”, tells PLATO to expect a touch input. The -keytype- statement will set “num” to 0, if the student touches as close as possible to screen location “1215”; will set “num” to 1, if the student touches near location “100,200”; and will set

“num” to -1, if the student touches the screen elsewhere.

How close the student must be to location “1215” or location “100,200” depends on the resolution or fineness of the touch panel. Most touch panels cover the screen with a 16 by 16 grid of square touch areas. Each square is 32 dots by 32 dots in size, or 4 characters wide by 2 characters high. If the square touched by the student overlaps location “1215” or location “100,200”, TUTOR will consider that the student has pointed at that place.

You can define larger regions of the screen. For example:

```
keytype num,touch(1215;8,4),touch(100,200;64,32)
```

In this case, the -keytype- statement will set “num” to 0 if the student touches somewhere within a box whose lower left corner is at “1215”, whose width is 8 characters, and whose height is 4 characters. The variable “num” will be 1 if the student touches within a box whose lower left corner is at fine-grid location “100,200”, whose width is 64 dots, and whose height is 32 dots. The touch-panel square touched by the student must overlap one of your rectangles in order for TUTOR to consider that a rectangle has been touched.

You can abbreviate “touch” by “t” and write “t(1215)” instead of “touch(1215)”.

In addition to the pause-keytype combination, you can also use a -touch- judging command with an -arrow-. See the PLATO on-line “aids” for details.

## Summary

In this chapter we have discussed, in some detail, the marker properties of the -arrow- command. The -arrow- command as we have seen serves as an anchor point which TUTOR clings to until the -arrow- is satisfied by an “ok” judgment (at which point a search is made for additional -arrow- commands). We looked at some cases involving the repeated execution of -join- in regular, judging, and search states, and of the *non*-execution of -goto- in the judging and search states. We have also looked at other side-effects of the -arrow- command, including initializations associated with -size-, -rotate-, -long-, -jkey-, and -copy-.

In addition, we have seen how the -bump- and -put- commands can be used to change a student’s response into a form more easily handled by the standard judging commands. This is particularly useful when only slight changes are necessary.

In Chapter 7 we saw how to store numeric and alphanumeric

responses for later processing (-store- and -storea-). These capabilities make it possible to “do your own judging” in those cases where the standard judging commands are not suitable. The basic TUTOR judging commands provide a great deal of power but cannot handle all possible situations. Fortunately, there is always the possibility of performing calculations on a stored student response, which means that TUTOR is open-ended in its judging power. The regular commands -search- and -move- can be used to manipulate stored character strings. (In Chapter 10 you will find discussions of “segments” and “bit manipulations” which permit you to use the -calc- command to perform additional operations on character strings.) We have also discussed how to handle input from the student by collecting each key with a -pause- command, then using -keytype- (aided by -group-) to make decisions on a key-by-key basis. We have learned, also, how to use similar techniques to determine where the student had touched the screen.