MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

Computation Structures Group Memo 189

Notes on Using TOPS-20

by

Eugene Stark

This research was supported by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-75-C-0661.

February 1980

Notes on Using TOPS-20

Eugene Stark

This note provides some basic information about programming in CLU on the DECsystem 20. The first section is an introduction to using the TOPS-20 operating system and EXEC, its command interpreter. The use of the TED text editing program is described in the second section. The third section discusses how to compile, debug, and run CLU programs. This note is intended primarily for use by students in 6.170. The presentation is therefore geared toward those who may have had no prior experience with the DEC-20 or CLU.

1. Introduction to TOPS-20

1.1 Logging In and Logging Out

When you walk up to a TOPS-20 terminal, make sure that no one else is already logged in before you attempt to log in. A logged-out terminal will generally display a message like the following:

```
LOGOUT JOB 22, USER SE.STARK, TTY 17,
 AT 9-SEP-79 11:08:04, USED 0:2:30 in 0:10:20
```

Before you can log in, you must first get the system's attention by typing ↑C. For those not familiar with ASCII keyboards, ↑C is pronounced "control C", and is generated by holding down the CTRL key while typing C. After typing ↑C the system should respond with a short message identifying itself, and will then print the character @ as a prompt. At this point, you are talking to EXEC, the TOPS-20 command interpreter, and are ready to log in.

To log in, you will need a *username* and a *password.* The username and password are presented to the system by typing the keyword `login`, then a space, then your username followed by another space, and then your password, followed by a RETURN. Note that printing is turned off automatically after you type the second space, in an attempt to keep your password secret. Thus, if your username is `se.stark`, then the line you type will appear on your terminal as

```
@login se.stark
```

If everything is correct, the system will print a few lines of information that include the job number you have been assigned, and the time someone last logged in under your username. The last thing that will be printed is the @ prompt of EXEC. At this point, you are logged in, and are ready to give commands.

If something is amiss with the login sequence, the system will print a line beginning with a question mark, for example

```
?INCORRECT PASSWORD
```

or

```
?DOES NOT MATCH DIRECTORY OR USER NAME
```

If you receive the first error message, you probably made an error in typing the password, and should type the login command again. If the second error message is printed, check to see that you typed the proper username. It is *not* necessary to be concerned about distinguishing between upper- and lower-case letters.

To log out once you have logged in, type `logout` to EXEC, followed by RETURN. From now on, it will be implicitly understood that each line of input you type at your terminal must be terminated by typing RETURN. The RETURN character informs the computer that the line of input is complete and should be processed.

1.2 Giving Commands and Running Programs

After logging in, you are ready to give commands and run programs. A *command* is executed inside EXEC itself, whereas a *program* must be executed as a separate *fork* or process. Some of the more useful commands and programs will be described briefly later in these notes. Both commands and programs are invoked by typing their names to EXEC. To obtain a complete list of all command names, type a `?` to EXEC. The `?` character is almost universally interpreted by system programs as a request for help. If you need prompting as to the type of input the system requires at any point, or you would like a list of available options to be printed, `?` is the easiest way to get this information. The `help` command will also provide you with brief documentation on some system programs.

To run a system program, simply type its name. For example, `ted` invokes TED, a text editing program. If you are executing a program or command, and want to return to EXEC command level, you should type ↑C twice. If the program you are running is currently awaiting input from your terminal, then it is actually only necessary to type one ↑C to return control to EXEC. However, two ↑Cs will always interrupt the program immediately, regardless of whether it is waiting for input or not. "Panic mode", that is, typing more than two ↑Cs, is no more effective than just two, although it may be more satisfying. Note that many programs, for example TED, have more graceful ways to return to EXEC than by ↑C. If the program you are running has such an escape, it is probably better to use it than ↑C. A *program* that has been halted with ↑C can usually be continued by typing `continue` to EXEC; however interrupted *commands* cannot be continued.

1.3 Special Characters

You can generate several characters that are interpreted specially by TOPS-20, which allow you to correct typing mistakes and perform various other functions. To erase the last character you typed, press the DELETE key (RUBOUT on some terminals). On a CRT terminal, the offending character will be removed from the screen, and the cursor moved back one space. If an entire line is a disaster, typing ↑U (control U) will delete it. Typing ↑W if you are at EXEC command level causes the contents of the last field of a command to be erased. If you are running a program that sends a lot of output to your terminal, and you don't care about seeing the output but you want the program to run to completion, the ↑O character may be used as a toggle to turn printing off and on until the program has completed, at which point printing is resumed normally. When you are running nearly any program besides TED, typing ↑T will print information about the CPU time used by the program. It is useful mainly for telling if a program is running, or has somehow gotten stuck. If your terminal is in "page" mode (see description of the `terminal` command below), output to your terminal will be stalled when the screen is filled up. To see the next page of output, type ↑Q.

1.4 Recognition Input

If you are at EXEC command level, another way to get assistance from the system is by pressing the ESC key (ALT on some terminals). Typing this character invokes a system feature called *recognition input.* When ESC is pressed, the system attempts to complete a partially entered command field. The basic rule is, in order for the system to complete your partial input, it must be able to unambiguously determine what it was that you intended to type. If it is unable to do this, your terminal will "feep", and nothing further will be printed. If the system is able to complete the field, the input that it supplies will added to the command line, entirely as if you had typed it yourself. If you are not happy with the system-supplied input, you may use DELETE, ↑U, or ↑W to edit it. When you are satisfied with the way the command line appears, typing a RETURN causes the command to be executed. Note that ESC by itself will *never* initiate execution of the command, so the best way to get a feel for how recognition input works is just to try it out. If you decide not to execute a command line, a ↑U will erase it.

As an example of the use of recognition input, suppose you were attempting to log in, and you typed `log` and then an ESC. The system would then realize that you wanted to type `login`, and would complete the typing of that command, and in addition supply some "guide words" to indicate what you should type next. The command line would then appear as follows:

```
@logIN (USER)
```

For clarity, the original input `log` is shown here in lower case, and the system-supplied input is printed in upper-case. Note that the ESC character does not print. The system is now waiting for you to type your username.

If you had typed `lo` and then an ESC, the system would not be able to determine unambiguously that you intended to type `login`. Its only response in this case is to cause your terminal to "feep". To see what other commands also start with `lo`, type a `?`.

```
@lo? COMMAND, ONE OF THE FOLLOWING:
LOAD    LOGIN   LOGOUT
@LO
```

As you can see, in response to the `?`, the system will print a list of the possible commands, and will then retype the partially entered command line and await further input.

When you become more accustomed to the system, you will find it convenient to use the *abbreviated input* feature of EXEC. This means that you need only type as much of a command keyword as is necessary to distinguish it from any other command. For example, when logging in, it is only necessary to type the first three letters of `login`, thus

```
@log se.stark
```

is entirely acceptable to the system.

1.5 The File System

The TOPS-20 file system is organized as follows: At the level of coarsest division, the file system is composed of one or more *structures*, which correspond roughly to magnetic disk drives. Each structure is divided into a number of *directories*. Each directory in turn can contain a number of *files*, which are the actual units of information storage. At any given time, each user is said to be *connected* to a single directory. When you log in, you will automatically be connected to a directory of your own, where you may store your work. As you create, delete and modify files, you will find it necessary to perform certain "housekeeping" operations on your directory, so that you do not exceed a maximum amount of storage space, called your *quota*. More will be said about this below.

Each file has an associated *filename*, which serves as a unique identifier. An example of a complete filename is:

```
ps:<se.stark>sorted_list.clu.4
```

The first field is called the *device field* and is terminated with a colon `:`. The name of the structure on which the file resides is placed in this field. In this case, the structure is `ps:`,

for "public structure". The second field is called the *directory* field, and is always enclosed in angle brackets. In this case the name of the directory is `<se.stark>`. The name of your private directory is your username enclosed in angle brackets.

The remaining three fields of the filename are separated by periods ., and specify the name of the file within the directory. The first of these fields, in this case `sorted_list`, is a name that is usually chosen to be descriptive of what is in the file. The second field is in general chosen to indicate what *type* of information is in the file. In this case, `clu` indicates that the file contains a CLU program. The last field must be numeric, and indicates the *generation number* of the file. It is usually not necessary to use a complete filename to access a file. Whenever you omit the device field and the directory field in a filename, the system automatically uses as default the directory to which you are currently connected. The generation number field is almost always omitted -- this is discussed in more detail below.

Generation numbers are a feature that help you protect yourself against accidentally destroying information in files you wanted to save. You may take advantage of this feature by *not* including the generation number field in a filename. The system then uses some simple rules to fill this field in for you. The generation number supplied by the system depends upon whether a read or write operation is being performed. Suppose the filename you typed was `sorted_list.clu`. Since both the device field and the directory field have been omitted, the system uses the connected directory as the default. The system then searches this directory to determine the highest generation number associated with any file named `sorted_list.clu`. If a read operation is being performed, then this number is used. Otherwise, if a write operation is being performed, then the generation number is this maximum number plus one. If there are no files named `sorted_list.clu`, an error message is printed in the case of a read operation, and generation number `1` is used in the case of a write.

Thus, if you never specify a generation number in a filename, you will always read from the latest version of a file. In addition, you will never write over an old file, but will always create a new version. However, it is possible to override this mechanism by explicitly specifying generation numbers. The main disadvantage of always creating a new version every time a file is written is that lots of old versions tend to accumulate and use up valuable space in your directory. One of the "housekeeping operations" you will need to perform periodically is the deletion of outdated versions of files. The `delete` command, described below, has options that are convenient for this.

1.6 Some Useful Commands

The 'Help' Command

The `help` command may be used to obtain brief documentation on various system programs. Simply typing `help` causes a short message explaining the use of the `help` command itself to be printed. `Help *` prints a list of the programs for which help is available. If `prog` is one of the programs in this list, then `help prog` prints the help for that program.

The 'Directory' Command

You may obtain a listing of the names of files in your connected directory by typing `directory`. Typing `directory <dir>` lists the names of files in directory `<dir>`. Longer listings containing more information about the files may be printed with the alternative commands `fdirectory`, `tdirectory`, and `vdirectory`, for "full directory", "time-ordered directory", and "verbose directory", respectively.

The 'Type' Command

Issuing the command `type filename` prints the contents of the file `filename` on your terminal. The command `type filename1,filename2,filename3` prints the contents of all three files.

'Copy' and 'Rename'

It is often useful to make a copy of a file, and the `copy` command serves this purpose. To create a file `prog2.clu` which is a copy of `prog1.clu`, simply type

```
@copy prog1.clu prog2.clu
```

The system will print a message telling you what it did:

```
PROG1.CLU.2 => PROG2.CLU.1 [OK]
@
```

If you simply wanted to change the name of a file, without making a copy, then `rename` should be used in place of `copy`.

'Delete', 'Undelete', and 'Expunge'

To delete a file you no longer have any use for, type `delete filename`. If no generation numbers are present in `filename` all generations of the file are deleted. You may delete specific generations of a file by explicitly specifying the generation number.

On TOPS-20, asking for a file to be `deleted` simply causes the filename to stop appearing in directory listings. It does *not* release the storage area associated with the file, nor does it cause the information stored within the file to be destroyed. In fact, once deleted, a file may be restored by typing `undelete filename`. However, deleted files are vulnerable to invocations of the `expunge` command. When you type `expunge` to EXEC, any deleted files in your directory really go away, and cannot thereafter be restored via `undelete`. It is to your advantage to `expunge` your directory occasionally, since otherwise your quota of storage will eventually be used up by deleted files.

As was mentioned above, as you work, new generations of files are continually created. Unless you explicitly delete the old generations, you will quickly run out of storage space. One way to do this is to laboriously type each of the filenames, including generation number, that you want to delete. A more efficient way is to use a special option of the `delete` command for this purpose. The `delete` command, as well as many other commands, allows you to enter a number of *subcommands*, which simply select various bells and whistles. To enter subcommands, terminate your command line with a comma ,:

```
@delete *.clu,
@@
```

The filename `*.clu` indicates that you wish the `delete` command to operate upon all CLU programs, subject to the subcommands which you will enter in response to the `@@` prompt. To find out what subcommands are available, type a question mark.

```
@delete *.clu,
@@? CONFIRM WITH CARRIAGE RETURN
 OR ONE OF THE FOLLOWING:
DIRECTORY       EXPUNGE         FORGET          KEEP
@@
```

The `expunge` subcommand causes all files deleted as a result of this command to be immediately expunged. The `keep` option is the one that is useful for deleting old generations. Suppose you are interested in saving the latest version of a file and one backup. You would then type

```
@@keep 2
@@
```

which signifies that, for each CLU program in your directory, you wish to keep the two versions with the highest generation numbers. To exit subcommand mode and execute the command, type a RETURN. If you decide that you do not want to perform the deletion, type ↑C. If you do perform the deletion, the system will tell you what files it is deleting.

```
MAIN_PROG.CLU.3 [OK]
SORTED_LIST.CLU.10,11 [OK]
@
```

If you are certain that you have deleted the proper files, then an `expunge` is reasonable.

```
@expunge
PS:<SE.STARK> [12 PAGES FREED]
@
```

The system indicates that twelve *pages* (a unit of storage equal to 512 36-bit words) have been returned to the system freelist.

The 'Information' Command

The purpose of the `information` command is to supply you with miscellaneous information about the status of a wide variety of objects in the system. The system object you will be most concerned with is your directory, and `information` will supply you with two types of information about it. The command

```
@information (ABOUT) directory
```

allows you to find out your quota of disk storage. In the above command, ESC was pressed after typing `information`, and the system has responded by printing the "guide word" `(ABOUT)`. The information printed by the system appears as follows:

```
@information (ABOUT) directory
NAME PS:<SE.STARK>
WORKING DISK STORAGE PAGE LIMIT 100
PERMANENT DISK STORAGE PAGE LIMIT 100
NUMBER OF DIRECTORY 65
ACCOUNT DEFAULT FOR LOGIN - NONE SET
@
```

The important information is contained in the first three lines. The first line gives the name of the directory about which information is being printed. The second line indicates the maximum number of pages you will be allowed to use while you are logged in and

working. If you attempt to use disk space in excess of this limit, you will receive an error message, and will not be able to complete the write you were trying to perform. The third line indicates the amount of disk storage you may retain between logged-in sessions.

To find out how much storage you are currently using, type

```
@information (ABOUT) disk-usage
```

The response from the system looks like

```
87 PAGES ASSIGNED, 79 IN USE, 8 DELETED
100 WORKING PAGES, 100 PERMANENT PAGES ALLOWED
4303 PAGES FREE ON PS:, 146774 PAGES USED.
@
```

The first line indicates that files in your directory are using 87 pages of disk storage, of which 8 pages are contained in files that are `deleted`. The second line gives your quota information, and the third line tells how many pages are used and free on the structure `PS:`. The 4303 free pages are the total number of pages on `PS:` that are available for use by anyone, when these are gone, the disk is full.

The `information` command will also tell you about what kind of terminal options are set for you by the system. To get this information, type

```
@information (ABOUT) terminal
```

The system will respond with a list that looks like the following:

```
TERMINAL VT52
TERMINAL SPEED 300 9600
TERMINAL NO PAGE
TERMINAL WIDTH 24
TERMINAL LENGTH 80
TERMINAL LOWERCASE
TERMINAL TABS
```

The first line indicates that the system thinks the terminal is a "VT52", which is a kind of DEC terminal. The second line states that the line from the terminal to the computer is configured for transmission at 300 baud, and the line from the computer to the terminal is set at 9600 baud. The third line indicates that the terminal is not in "page" mode, and the remainder of the lines give other information, much of which is implied by the fact that the terminal is a VT52. The option you will probably be most concerned with is "page mode".

If the terminal is in "page mode", the character ↑Q is used to restart printing as was previously described. If the terminal is not in page mode, the system does not interpret ↑Q as a special character.

The 'Terminal' Command

You may change various terminal options by using the `terminal` command. To set the terminal to page mode, type

```
@terminal page
```

To get out of page mode type

```
@terminal no page
```

To tell the system that you are using a VT52, type

```
@terminal vt52
```

Other common terminal types which may be used in place of "VT52" are: (1) "HEATH" - Heath terminal; (2) "HP" - Hewlett Packard; (3) "LA36" - any DECwriter-like printing terminal, e.g. Anderson-Jacobson; (4) "FOX" - Perkin/Elmer "Fox".

You probably should not change other terminal options, especially the speeds, unless you know what you are doing. If you set the speeds to the wrong values, you can cause the terminal to become useless. The intervention of an operator will then be required to reset the terminal.

Fork Manipulation: The 'Reset' and 'Continue' Commands

The EXEC command interpreter allows you to control a number of processes at once. Each time you invoke a program (by typing its name), a new process, or *fork* is created to run that program. In general, a fork will stay around until it is explicitly deleted with the `reset` command. The forks currently in existence may be listed with the `information forks` command, as follows:

```
@information (ABOUT) forks
=> TED (1): KEPT, HALT AT 702032, 0:00:59.0
  CLUSYS (2): ↑C FROM IO WAIT AT 710252, 0:00:00.2
```

In this case there were two forks, TED and CLUSYS, numbered (1) and (2), respectively. Forks may be referred to by name or by number. At any given instant one fork is considered the *current* fork. In the above example, TED is the current fork, as indicated by

the => arrow pointing at the fork name. A fork is deleted by typing `reset`, followed by the fork name or number. If the CLUSYS fork is no longer needed, then it may be deleted by typing:

```
@reset clusys
```

or

```
@reset 2
```

To see that CLUSYS has really gone away, use `information forks` again:

```
@information (ABOUT) forks
 => TED (1): KEPT, HALT AT 702032, 0:00:59.0
```

If `reset` is typed without specifying a fork name or number, then the current fork is reset. The `keep` command makes the current fork immune to being reset, unless its name or number is explicitly specified in the `reset` command. TED automatically "keeps" when it starts up; this helps keep you from accidentally destroying unsaved work.

If ↑C is typed while a program is running, the program is temporarily suspended, and control is returned to EXEC. This is what happened to the CLUSYS fork in the above example, as indicated by the message `↑C FROM IO WAIT`. A suspended program may be continued by typing `continue`, followed by the name or number of the fork. If `continue` is typed without a fork name or number, then the current fork is continued.

2. Introduction to TED

This section is an introduction to TED, a display text editor program developed at MIT. People who are familiar with text editors should skip to the part on "Getting TED started."

2.1 What a Text Editor Does

A text editor can be described as a typist's assistant. Just as wizards have demons to perform the tedious details of magic, a person who types text should have an assistant to do at least some of the work. In the case of the typist, however, the assistant is the computer.

First, let's look at the equipment needed to do text editing. We need a computer that has enough storage to remember the contents of various documents, a terminal to talk to the computer, and a printer to print pages of text. When documents are in the computer, we call them *files*. The kind of terminal we will be discussing here is a *display terminal*, which has a typewriter-like keyboard and a TV-like screen. For the computer, we will be using a Decsystem-20 made by Digital Equipment Corporation.

A text editor is a program that is used to create and change text files. Some things that one can do in a text editor are:

enter new text

add new text at any point in a file

delete text from any point in a file

find occurrences of pieces of text in a file

move text from one place to another

change occurrences of pieces of text to other text

The computer has two kinds of storage: disk and fast memory. There is a great deal more room on the disk than in the fast memory. So, most of the time, files are on the disk rather than in fast memory. In TED, there is an area of fast memory, called the *buffer*, where we have a working copy of any text file that we want to edit. Therefore, we also want to have commands for copying text files from disk to fast memory (*reading*) and copying text files from fast memory to disk (*writing*). A good analogy would be if a typist had an assistant to move a document from a file drawer to the typewriter's desk and back, and every time the assistant moved a document a copy of it was made. If this were done with paper, there would soon be an office full of paper. Since it is actually done with a

computer, we don't have a paper problem.

The best way to learn TED is to experiment. For most commands, the effects of the command are immediately visible on the screen. Also, look through the documentation. When you see something that you don't quite understand, but you think that it might be useful, try it out.

2.2 Getting TED Started

After you log in, the computer will print varying amounts of stuff, then wait for you to type a command. At this point it is running a program called EXEC. Before you invoke TED, be sure that TOPS-20 knows the type of terminal you are using. To start TED, just type `ted`, followed by a carriage return. After a short time, the TED program will start up, clear the screen, and print

```
---- MAIN BUFFER:  (TYPE ↑H FOR HELP)
```

near the top of the screen. Note that `↑H` means "control-H", which is typed by holding down the CTRL key and typing `H`. If you ever need to see the help information that TED can show you, just type `↑H`. TED will print a list of items it knows about, then you will be asked to type a single character to select which help you want to see.

To leave the TED program, use the `↑@` character or the break key. This should print

```
ESCAPING TO SUPERIOR.
@
```

which is asking you for a normal EXEC command. For obscure software reasons, it is not a good idea to use `↑C` to exit TED except when `↑@` does not work.

When in EXEC, to return to TED, type

```
continue
```

2.3 Simple TED Commands

To enter text, just type normal printing characters as you would on a typewriter. To get a new line, just type carriage return. If you mistype a character, however, you can delete it immediately with the DELETE key. Then you can type the correct character. Normally, the characters that you see on the screen are actually in the buffer (except for the top two lines, which have other information).

As you are typing, you will see a little marker moving along on the screen just ahead of what you are typing. This is called the *cursor.* It marks your current position in a document. It will appear differently on different terminals. The cursor can be moved from place to place by various commands. The most basic of them are:

↑F: to move forward one character

↑B: to move backward one character

↑N: to move to the beginning of the next line

↑P: to move to the beginning of the previous line

↑A: to move to the start of the current line

↑E: to move to the end of the current line

As you type these characters, the cursor on the screen will move. When many people are trying to use the computer, the cursor may not move as fast as you type the characters, but it should catch up in a short time.

To add text into the middle of text that you have, just put the cursor where you want to add the text and type in the text. This inserts the text, and displays the new text on the screen. So inserting text is the same as entering new text.

To delete old text, you can use either the DELETE key, or the ↑D character. The difference is that ↑D deletes forwards, and the DELETE key deletes backwards. To delete entire lines, use the ↑K character. It deletes from where the cursor is to the end of the line. It also deletes the end of the line, so the two lines are joined together.

As you are typing in, you will probably accumulate more text than can fit on a screen. You need not do anything special to make more room. The screen is used as a "window" on the buffer, so it does not matter how much text you have (well, try to keep it less than 100,000 characters in any file). TED will try to keep a window displayed around your cursor. If you still cannot see enough information, use ↑L to redisplay your screen. If that does not work, you probably need a bigger screen.

2.4 Reading, Writing, and Searching

Once you have entered a document into the buffer, you need to write it to a file on disk. Use the ↑W command to do this. When ↑W is typed, the message

```
---- WRITE BUFFER TO FILE: (DEFAULT "...")
```

will appear near the bottom of the screen, and the cursor will be placed below the message. At this point, TED is asking for a file name. If you like the default file name that TED shows you, just type carriage return. If you want to write the buffer to a different file, or there is no default, then type in the file name followed by a carriage return. When you are typing the file name, a mistaken character can be deleted by the DELETE key. If you change your mind about writing the buffer to a file, type ↑G to quit from the ↑W command, and no writing will have taken place. The text in the buffer will not be changed. The ↑G character may be used to abort any partially-completed TED command without ill effect.

To read an old file into the buffer, use the ↑R command. This command is similar to the write command, since it will ask you for a file name. You can use delete and ↑G in the same way when typing in the file name. However, when the file is read into the buffer, the old contents of the buffer will be lost.

Once you have read an old file into the buffer, you may wish to find a particular place in the file where you want to make changes. The easiest way to do this is to use the ↑S command to tell TED to search for a particular piece of text. It will ask you for the text that you want to find. Just as with the read and write commands, end the text with a carriage return. If the text is found by looking in the forward direction, TED will move the cursor there. Otherwise, TED will display a message saying that the text was not found, and the cursor will stay in the same place.

2.5 Numeric Arguments

Many TED commands make use of a numeric argument. This number can be used as a counter to indicate how many times to do something, or it can change the meaning of the command. These numeric arguments can be specified by the ↑U command. For example, with the ↑N command:

↑U↑N:	go forward 4 lines
↑U7↑N:	go forward 7 lines
↑U↑U↑N:	go forward 16 lines (4*4)

Notice that when ↑U is followed by a number, that number is used for the argument. When ↑U is used alone, it uses 4 as the argument. When two ↑U commands are used in a row, the two numbers are multiplied.

Searching can be done either forwards or backwards, depending on the numeric argument. ↑U5↑S means search forward for the 5th occurrence of some text. ↑U-↑S means search backwards for the first occurrence of some text.

Some commands are changed by the numeric argument. Some useful examples of these are:

↑U↑A:	move cursor to the start of the buffer
↑U↑E:	move cursor to the end of the buffer
↑U↑R:	insert the given file into the buffer at the cursor

If you start a ↑U argument and change your mind, use ↑G to abort the command.

Note that the numeric arugument only applies to the next command. If it is not given, TED assumes that the numeric argument should be 1.

2.6 Extra Commands

There are only so many control characters. So if we want to type in more commands, we need to use more characters to get them. One good example of this kind of command is ↑V. To find out what commands ↑V is part of, just type ↑V?, which will display the options that are possible. Some of the more commonly used ↑V commands are:

↑V↑N:	move cursor to next screenful of lines
↑V↑P:	move cursor to previous screenful of lines
↑V↑F:	move cursor forward a word
↑V↑B:	move cursor backward a word
↑VT:	display the current date and time

2.7 Moving Text

One job that a typist must sometimes perform is cutting and pasting. This job can be done more easily by TED just by moving text around in the buffer. To move text from one place to another, first move the cursor to the place that the text occurs, withdraw the text into the *save buffer*, move the cursor to the place that the text should be, and then insert the text from the save buffer.

To withdraw text into the save area, move the cursor to the start of the text that should be withdrawn, place an invisible marker there with ↑VM, move the cursor to the end of the text, and withdraw the text using ↑VW. To copy the text from one place to another without removing it from the place it was, use ↑VS instead of ↑VW. Then move the cursor to wherever the text should be, and insert the text with ↑VI. Note that the text can be inserted as many times as desired.

We can also withdraw or save several lines by using the ↑V↑W or ↑V↑S commands. To withdraw the next 3 lines starting with the cursor into the save buffer, use ↑V3↑W. To save them only, use ↑V3↑S. Insertion still uses ↑VI. This method is sometimes easier than using ↑VM, followed by ↑VW or ↑VS.

2.8 Changing Text

One way to change text is to just delete the text that is not wanted, then insert the text that is wanted. However, this can be tedious where you want to change some or all of one item to another. For this, we use the ↑Z command.

The ↑Z command will ask for the text item to replace (the *search string*), and the text item to replace it with (the *replace string*). The cursor will move to the first occurrence of the search string, and you will be asked about what to do. Typing ? will show you what options you have. Typing Y will replace the search string with the replace string and move the cursor to the next occurrence. Typing N will leave the buffer as it is and move the cursor to the next occurrence. Typing U will replace all further occurrences of the search string with the replace string. Typing Q will quit the update.

At the end of the changes, you will be asked whether or not you really want to make the changes in the buffer. This is useful in cases where some of the changes were made in error. Answer the question with either a Y (for "yes") or an N (for "no").

2.9 Miscellaneous

When lines are too long, not all of the characters can be displayed on a single line of the screen. TED marks lines that are too long with a ! character at the end of the line. The rest of the line can only be displayed by putting the cursor just before the ! character and inserting a newline character (type carriage return). Since long lines are difficult to read and manipulate, lines should be kept relatively short.

For commands that delete large amounts of text, the text deleted is placed in a special buffer called the *restore buffer*. That text can be inserted into the buffer using the ↑V↑R command.

Since most commands are reached by typing control characters, we need a special way to insert control characters into the text when they are needed in the text. To do this, just type ↑Q followed by the control character. This works for all control characters except for ↑C, which can be inserted by typing ↑V3#.

Every so often TED will decide that enough changes have been made to the buffer, and will save the contents of the buffer to a temporary file called `_ted.save` in your directory. This is done so your work will not be lost if the machine crashes. You may lose the last 100-200 characters you typed, but most of the work will be saved. When TED is saving the buffer, it will display a message to that effect at the top of the screen. When you log out, you can delete the `_ted.save` file, since you will not be needing it.

There are many other features of TED that may prove to be useful to you. You can find these features through the ↑H command, or you can read the file `<clu>ted.help`.

3. Compiling, Running, and Debugging CLU Programs

We have seen how to use the TED text editor program to enter a CLU program. Once typed in, a program may then be compiled using the program CLU, and then loaded and run with the CLUSYS run-time support system. This section describes the use of CLU and CLUSYS.

3.1 Running the CLU Compiler

The major function of the CLU compiler is to take a file containing CLU source and produce a binary file which can be loaded into CLUSYS and executed. The CLU compiler follows the convention that the second field of the filename of source files is `clu` and of binary files is `bin`. Thus compilation of the source file `sorted_list.clu` would create the binary file `sorted_list.bin`. It is also possible to use the CLU compiler to type- or syntax-check an input file, without producing binary.

The most convenient way to invoke the CLU compiler is from within TED. To start up a CLU compiler process, type ↑\K (control-backslash, then K) to TED. You will then be asked whether you wish to keep a new CLU inferior process. Answer this question with a Y (for "Yes"). If you later decide to kill this CLU inferior (unlikely), use ↑\K followed by N. Compilation is then performed by reading the file you wish to compile into TED and using one of the various ↑\ commands to process the file. A list of the most useful commands is given below:

↑\K	Keep or kill a CLU inferior process
↑\H	Type-check the current file without producing binary
↑\C	Compile the current file and produce binary
↑\A	Redisplay the error messages from the last compiler run
↑\I	Reformat the CLU program in the buffer to conform to indentation standards

When the compiler starts up for the first time, its internal tables are empty. As each file is processed, the compiler constructs tables of interface specifications for the various modules. These tables are used for type-checking purposes. Because you have indicated that you wish to *keep* the inferior CLU compiler process, the interface information is not deleted after each file is processed, but is retained inside the compiler. It is important to know that the compiler makes only a *single* pass through a file. Although this one pass is sufficient for the compiler to produce binary, it is not in general sufficient for complete checking of all inter-module references. Type-checking is important, and failure to do such checking can result in rather obscure run-time errors. The way to get complete type-checking is to process all files once with the ↑\H to construct the internal tables, and then to process the files again using ↑\C to produce binary.

Output from the CLU compiler is collected by TED and displayed on your screen when the compilation is finished. If there were errors, line numbers are given to indicate the line(s) of the program containing the errors. To get to a specific line in your program, for example line 5, type ↑V5L (control-V five L). Even though doing this erases the error messages from the screen and redisplays your program, you may review the error messages at any time by typing ↑\A. This feature of TED makes it easy to find and fix a number of errors at a time, without writing down line numbers.

It is perfectly reasonable to split a large program up so that each module occupies a separate file. If you do this, then modification of a single module requires only recompilation of that module; you do not have to recompile the entire program. This will improve turn-around for you and for others on the system.

The CLU compiler may be invoked directly from EXEC by typing `clu`. To find out the syntax of the command line it expects, type ? followed by RETURN. It is generally not convenient to run the CLU compiler this way unless you have a large number of files to compile.

3.2 Using CLUSYS to Debug and Run CLU Programs

Successful compilations of CLU programs produce binary (`.bin`) object files. To run these files, it is first necessary to invoke the CLU run-time support system by typing `clusys` to EXEC. CLUSYS may also be invoked from within TED; this is described in more detail later on. Starting CLUSYS causes it to enter a command input loop called the *listen loop*. When CLUSYS is in the listen loop expecting a command, it prints a ":" as a prompt. To get out of CLUSYS and return to EXEC, type ↑C.

The listen loop of CLUSYS consists of the following actions:

1. Read an input line from the TTY. The DELETE key may be used to delete the last character typed, ↑X and ↑U delete the entire line, ↑R redisplays the current line, and ↑L redisplays the current line after clearing the screen.

2. The input line is evaluated as a command. Legal commands are described below.

3. The results of this evaluation are printed on the TTY. If errors occur, messages to this effect are printed.

4. Steps 1-3 are repeated.

3.2.1 CLUSYS Commands

Commands to CLUSYS are intended to be roughly a subset of the legal CLU expressions, augmented with a means of *binding* objects to identifiers. When an expression is given as a command to CLUSYS, it is evaluated, and any result object(s) or error messages are printed, in a fashion described in more detail below. For a complete description of the syntax of CLUSYS commands that is concise to the point of being cryptic, refer to the file `<clu>clusys.intro`.

The set of expressions recognized by CLUSYS includes CLU literals, CLU identifiers, module names, invocations, record constructors and array constructors, and type specifications. Also a legal expression in CLUSYS is an assignment statement of the form:

<identifier_list> = <expression>

This assignment statement has properties of both the CLU assignment statement, and the CLU equate. Specifically, if evaluation of the expression on the right-hand side of the = produces an object, that object is then bound to the identifier on the left-hand side, exactly as in the CLU assignment statement. Note that if the expression on the right-hand side produces a list of objects, then these objects are simultaneously bound to the list of identifiers on the left. If the expression on the right evaluates instead to a type, then this type is bound to the identifier on the left, much as in a CLU equate.

3.2.2 Identifiers

The set of legal identifiers in CLUSYS includes the set of legal CLU identifiers. In addition, CLUSYS allows the use of the `%` character in identifiers. The reason for this is that various options of CLUSYS are associated with identifiers that begin with `%`. For example, if the command

```
%time = 1
```

is typed, CLUSYS will print out the CPU time used after each subsequent command is executed. To turn this option off, type

```
%time =
```

Since `%time` is not a legal CLU identifier, its value cannot be changed from within a CLU program, but only in the listen loop of CLUSYS. There are also procedures, whose names begin with `%`, that can only be invoked from within CLUSYS, for example the `%trace_to` procedure discussed below. There are also a set of predefined procedures, whose names

begin with _ (underscore), that may be invoked from within a CLU program, as well as from CLUSYS. These procedures control various low-level functions that are of interest only to people working on CLUSYS itself. However, to avoid name conflicts, it is a good idea not to use identifiers in your CLU programs that begin with an underscore.

3.2.3 Executing CLUSYS Commands

Using CLUSYS is much like using a LISP interpreter. Each command you type is evaluated. As a result of this evaluation, procedures may be invoked, and identifiers may be bound to objects or types. For example, to bind the object `3` to the identifier `count`, type

```
count = 3
```

CLUSYS responds by evaluating this expression and printing the result. In the following, lines beginning with => are printed by CLUSYS.

```
=> 3
:
```

To see what object `count` is bound to, simply type `count`

```
:count
=> 3
:
```

To print the sum of `count` and four

```
:int$add(count, 4)
=> 7
:
```

If you need to use `int$add` very often, it might be reasonable to assign it a shorter name.

```
:p = int$add
=> INT$ADD
:
```

The result of this assignment has been to bind the identifier `p` to the *procedure object* `int$add`. Until `p` is reassigned, it may be used in place of `int$add`. Note that, in a CLUSYS expression, if the character immediately following a procedure name is a left parenthesis, evaluation causes that procedure to be invoked, and the value of the expression is whatever object is returned by the procedure. If the left parenthesis is not

present, the result of evaluation is the procedure object itself.

There are quite a few procedures and operations defined by CLUSYS, and a complete list will not be given here. However, the `pmatch` procedure is useful if you have some idea of the name of the procedure you want, but do not know the exact name. If s is a string, then `pmatch(s)` prints a list of all procedures whose names contain s as a substring. For example, `pmatch("string$")` prints the list of all string operations.

So far we have seen how to bind identifiers and invoke operations on built-in types. Let us see now how to invoke procedures in our own CLU programs. The first thing to do is to *load* the binary object files produced by the CLU compiler into CLUSYS. This is done with the built-in CLUSYS procedure `load`, which takes a single string argument which is the name of the file to load. If the string contains no ".", the default suffix `.bin` is used. For example, to load the binary file `sorted_list.bin`, type `load("sorted_list")`. If no error messages are printed, the file was successfully loaded. The procedures and cluster operations defined in the file `sorted_list.clu` are now available for use.

Debugging in CLUSYS is accomplished by simply invoking the procedures you wish to debug, and seeing what the results are. Note that if the procedures you are trying to debug require complicated argument objects, it may be necessary to load additional procedures solely for the purpose of creating these objects. The *tracing* feature of CLUSYS is useful for finding out what happened when something goes wrong. How to use tracing is described below.

When a bug is discovered, it is necessary to go back to TED to edit the file containing the offending module, use CLU to recompile the file, and reload the object file into CLUSYS. If things are arranged so that the editing and compiling does not destroy the CLUSYS (how this can be done is described below) then it is only necessary to reload the file that was recompiled; it is not necessary to reload other files that did not change. When a file is loaded, the new versions of modules supersede any old versions that may already be loaded into CLUSYS.

3.2.4 Loading Modules From Multiple Files

When you want to run a program that has been split up into several files, it is necessary to load each file into CLUSYS. The "xfile" command is useful if you don't want to have to type a separate line to CLUSYS to load each of the files. The command

```
xfile(file1, file2)
```

where `file1` and `file2` are strings, will cause the lines in `file1` to be executed as CLUSYS commands, and will place the output from those commands in `file2`. Typing

```
_xfile(file)
```

is equivalent to

```
xfile(file, "tty:")
```

and will execute the lines in "file" and send the output to your terminal. Thus, to load a program contained in the files f1, f2, and f3; make a file called, for example, "all.xfile", which contains the lines

```
load("f1")
load("f2")
load("f3")
```

Typing `_xfile("all")` to CLUSYS will then cause files f1, f2, and f3 to be loaded.

3.3 From TED to CLU to CLUSYS to TED to CLU ...

TED has special commands that enable you to start up and keep CLU compiler and CLUSYS processes. In fact, it is probably more convenient to use CLU and CLUSYS from TED rather than from EXEC. When you log in for a session of debugging, it is suggested that you do the following:

(a) Start up a TED

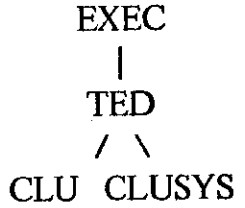
(b) Use the ↑\K command to TED to keep an inferior CLU compiler process as described above.

(c) Use the ↑J command to TED to start up an inferior CLUSYS process. When ↑J is typed, the message

```
---- INFERIOR FORK NAME (DEFAULT 'CLUSYS')
```

will appear at the bottom of your screen. Since you want to run CLUSYS, type a RETURN in response to this question. CLUSYS will then be started, and will be given control of the terminal. If you already have a CLUSYS process from a previous use of the ↑J command, you will be asked whether you wish to continue, restart, or kill that CLUSYS. Respond with a C, R, or K, as appropriate.

If you have followed parts (a), (b), and (c) above, the process structure you have created will now appear as follows:



To return to TED from the inferior CLUSYS type `valret("")` as a CLUSYS command. It is not a good idea to type ↑C to this CLUSYS, since this will send you all the way back to the top-level EXEC. However, if you do happen to type ↑C, typing `continue` will return you to CLUSYS, and `start` will restart TED, without losing your buffer.

Once a CLUSYS process has been started from TED, the ↑J command in TED gives you three options. The C option will continue the CLUSYS from where it stopped, the R option will restart the CLUSYS, and the K option will kill off the CLUSYS. If you somehow get a runaway CLUSYS, try ↑G which should return you to CLUSYS command level. If that doesn't work, type ↑C twice followed by `start` to return you to TED.

4. Debugging CLU Programs

This section describes how to use the debugging features of CLUSYS. These features will first be briefly described, and then illustrated with a sample debugging session. For a complete list of debugging commands and options, consult the files `<clu>clusys.intro` and `<clu>trace.info`.

4.1 Stack Manipulation

Effective debugging in CLUSYS requires some understanding of the run-time environment provided by CLUSYS. The most important part of this environment for debugging purposes is the *stack*. When CLUSYS is started, the stack is empty, except for a "dummy" frame, called `%base%`, which indicates the bottom of the stack. The first thing that happens is that CLUSYS invokes a procedure called `listen`, which prints a : as a prompt, and then reads and executes commands typed at the terminal.

Recall that commands to CLUSYS are either assignments or procedure invocations. Each time a procedure is invoked, a *frame* is placed on top of the stack. The frame contains information about the name of the procedure, the arguments passed, and the local variables to that procedure. When a procedure returns, the corresponding frame is removed, or "popped" from the stack.

Whenever CLUSYS is at command level, the current state of the stack can be printed by typing `frames()`. Note that being at command level in CLUSYS means that the `listen` procedure is running; therefore when `frames()` is used to print the stack, the top of the stack will always be a frame for `listen`. A problem with the `frames()` command is that it always prints the whole stack, and you may only be interested in the top few frames. In this case, typing `lframes(N)`, where `N` is an integer argument, will print the top `N` frames on the stack in a somewhat less compact format than `frames()`. If you are interested in printing a particular frame on the stack in great detail, complete with local variables, use `frame(N)`, where `N` = 0 prints the top frame, `N` = 1 prints the second frame from the top, etc.

Typing `↑G` to CLUSYS causes the procedure currently being executed to be interrupted, and a `listen` loop to be started. Since the frame for this new `listen` loop is placed at the *top* of the stack, the frames for the procedure that were interrupted may be viewed with the stack-listing commands described above. It is possible to return to the interrupted procedure by typing `erret()`. To throw away the entire stack, type `restart()`.

4.2 Accessing Arguments and Local Variables

The CLUSYS command interpreter provides a way for objects referenced by arguments and local variables on the stack to be named and used. For example, if one of your procedures has a local variable `foo`, and the current stack contains an activation of that procedure, with frame number 4, then you can type `@ 4 foo` in a CLUSYS command as a name for the object reference by the local variable `foo` in that particular procedure activation. Arguments to procedures can be named in a similar way, for example, `? 4 bar` names the object passed as argument `bar` to the procedure activation in frame 4. It is also possible to omit the frame number (e.g. `? bar`), in which case the stack is searched starting from the top for the specified argument or local variable.

Examples of the use of these features are shown in the sample debugging session below.

4.3 How Objects Are Printed

How should an object of abstract type be printed? Ideally, each cluster would include the definition of a print operation, which would print a suitable representation of an argument object. However this is not done in CLUSYS for various reasons, not the least of which is the fact that since CLUSYS retains no type information with an object, it is impossible to select the appropriate print operation. It has therefore been necessary in CLUSYS to settle for less than the ideal.

An object in CLUSYS is printed as its ultimate representation in terms of primitive CLU types, such as integers and characters, and type constructors, such as arrays and records. In other words, all abstract type is thrown away, and the object is printed as one big hairy structure made up of primitive types and constructors. The easiest way to see how this works is to make some objects and print them. When your objects get bigger than a certain size, you may notice that their printing is truncated. You can extend the point at which an object will be truncated with the routines `set_print_width` and `set_print_depth`, which take an integer argument. When CLUSYS starts up, the print width and print depth are set to 4.

4.4 The Trace Feature

The major feature of CLUSYS intended specifically for debugging is the *trace* feature. This package of routines enables the user to selectively set breakpoints in his program. Breakpoints are always associated with the invocation and return of a procedure or iterator. The trace package consists of the following routines.

`%trace_to(P)` - break on each call to and return from procedure `P`.

`%untrace_to(P)`	- remove breakpoints set by `%trace_to(P)`.
`%trace_from(P)`	- inside `P`, break on each invocation and return of a procedure.
`%untrace_from(P)`	- remove breakpoints set by `%trace_from(P)`.
`%trace(P)`	- combine effect of `%trace_to(P)` and `%trace_from(P)`.
`%untrace(P)`	- remove breakpoints set by `%trace(P)`.
`%print_slots(P)`	- print a numbered list of all invocations of procedures inside `P`.
`%trace_range(P, low, high)`	- selectively set breakpoints on specific invocations inside `P`. The integers low and high are indices in the list printed by `%print_slots(P)`.
`%untrace_range(P, low, high)`	- remove breakpoints set by `%trace_range(P, low, high)`.
`%untrace_all()`	- remove all breakpoints.

Note that the argument `P` to these routines must include all parameters; that is, if `mergesort` is a procedure with a single type parameter, then `%trace(mergesort[int])` must be used, and not `%trace(mergesort)`.

When a breakpoint is encountered during execution, some information about the stack frame will be printed, followed by `-- next --`. At this point, a single character should be typed. Some of the more useful characters and their effects are:

space - Continue until the next breakpoint is encountered.

`x` - (eXchange) Toggle the frame printing mode between the short form, which is that used by `frames()`, and the long form, which is that used by `lframes()`. The frame is redisplayed, and the `-- next --` prompt is printed, requesting another command.

`^L` - Clear the screen, redisplay the stack frame, and request another command.

RETURN - Start up a new listen loop on top of the stack. This listen loop may be exited with `erret()`.

`Q` - (Quit) Stop all tracing until the listen loop is reentered. This is not the same as doing `%untrace_all()`, because the permanent breakpoints are not removed; just temporarily ignored.

`?` - Print a brief listing of the command characters and their meanings.

`@` - Escape to the superior process.

The above commands are usually sufficient for most purposes. For real debugging problems though, there are several other characters which allow you to single-step through a procedure, turning on and off breakpoints as you go. For a list of these commands, refer to the file `<clu>trace.info`.

4.5 Scripting

Scripting is a way of sending CLUSYS output to a stream other than the primary output. Typing `stream$add_script(from_stream, to_stream)` will cause all subsequent input from or output to `from_stream` to be output to the `to_stream`. It is possible to set multiple scripts by repeated executions of `stream$add_script` with various arguments. Scripting can be turned off with `stream$rem_script(from_stream, to_stream)`, which removes scripting for just this particular pair of streams, or `stream$unscript(from_stream)`, which removes all scripting on `from_stream`. Thus to cause input and output to your terminal to be sent to a file "script.out", type the following:

```
to_stream = open_write("script.out")
```

which opens "script.out" for writing, and then

```
stream$add_script(po, to_stream)
```

which sets up the scripting. Note that when CLUSYS is started, the identifier `po` is initially bound to the primary output stream. To turn off this scripting, use

```
stream$unscript(po)
```

It is then a good idea to close the file "script.out" by typing"

```
stream$close(to_stream)
```

4.6 A Sample Debugging Session

The use of the debugging features described above will now be illustrated with a sample debugging session. The program to be debugged is an implementation of the mergesort algorithm. There are three procedures, (1) mergesort[T], which accepts an array of objects of type T, turns it into an array[array[T]], and repeatedly calls merge[T] to do the merging; (2) merge[T], which takes two array[T] objects and merges them; and (3) append[T], which takes two objects of type array[T] and appends one to the end of the other.

```
%
% Mergesort program - contains bugs in marked lines
%

mergesort = proc[T: type](a: at) returns(at)
   at = array[T]
   aat = array[at]
   aa: aat := aat$new()
   for e: T in at$elements(a) do
         aat$addh(aa, at$[1: e])
         end
   while aat$size(aa) > 1 do
         naa: aat := aat$new()
         for i: int in int$from_to_by(1, aat$size(aa), 2) do
            aat$addh(aa, merge[T](aa[i], aa[i+1])) % buggy
              except when bounds:
                              aat$addh(aa, merge[T](aa[i], at$new())) %buggy
                        end
            end
         aa := naa
         end
   return(aa[1]) except when bounds: return(at$new()) end
   end mergesort

merge = proc[T: type](a, b: at) returns(at)
            where T has lt: proctype(T, T) returns(bool)
   at = array[T]
   c: at := at$new()
   ia: int := 1
   ib: int := 1
   while true do
         if at$size(a) < ia then
            append[T](b, ib, c)
            return(c)
         elseif at$size(b) < ib then
            append[T](a, ia, c)
            return(c)
         end
         if a[ia] < b[ib] then
            at$addh(c, a[ia]) % buggy
            else
```

```
                    at$addh(c, b[ib]) % buggy
                end
            end
        end merge

append = proc[T: type](src: at, start: int, dst: at)
    at = array[T]
    for i: int in int$from_to(start, at$size(src)) do
            at$addh(dst, src[i])
            end
    end append
```

```
: load("mergesort")                                    % load in the object file

: a = array[int]$[1: 1, 6, 3, 4, 7]                    % set up some test data

=> [1..5: 1 6 3 4...]
: mergesort[int](a)

% Here the program went into a loop; I typed ↑G

Quitting to new command level.

: frames()                                             % What does the stack look like?

 0: listen (tyi: stream#[R, tty:], tyo: stream#[W, tty:])
 1: %quit ()
 2: _ctrlg_handler ()
 3: merge (a: [1: 1], b: [1: 6])
 4: int$from_to_by (from: 1, to: 5, by: 2)
 5: mergesort (a: [1..5: 1 6 3 4...])
 6: value$apply (item: mergesort, av: [1: [1..5: 1 6 3 4...]])
 7: scan$mexpr (ac: [18:])
 8: listen (tyi: stream#[R, tty:], tyo: stream#[W, tty:])
 9: %base% ()
: set_print_width(10)                                  % Lets me see more of the arrays.

=> 10
: a

=> [1: 1 6 3 4 7]
: lframes(5)                                           % How about the top five frames?

 0: listen 27535
        tyi:    stream#[R, tty:]
        tyo:    stream#[W, tty:]
 1: %quit 27527
 2: _ctrlg_handler 27525
 3: merge 27464
        a:      [1: 1]
        b:      [1: 6]
 4: int$from_to_by 27455
        from:   1
        to:     5
```

```
        by:     2
 5: mergesort 27443
        a:      [1: 1 6 3 4 7]

...

: frame(5)      % Let's see the local variables inside "mergesort".

 5: mergesort 27443
        a:      [1: 1 6 3 4 7]

        aa:     [1: [1: 1] [1: 6] [1: 3] [1: 4] [1: 7]]
        e:      7
        naa:    [1:]
        i:      1
: %trace_to(merge[int]) % Now get breakpoints on calls to merge[int]

: mergesort(a)

Error - could not snap: call#(desc#(array)$new, 0)
Quitting to new command level.

: mergesort[int](a)     % I forgot to type the [int] parameter,
                        % so it couldn't find the mergesort procedure.

Calling merge                   % First breakpoint, call to merge

merge (a: [1: 1], b: [1: 6])    % Stack frame info
mergesort (a: [1: 1 6 3 4 7])
-- next --                      % Looks OK, I typed space to continue,
                                % but program looped, so I typed ↑G

Quitting to new command level.

: frames()                      % Note that stack has grown, since I
                                % never returned from the previous
                                % listen loops (frames 8 and 16)

 0: listen (tyi: stream#[R, tty:], tyo: stream#[W, tty:])
 1: %quit ()
 2: _ctrlg_handler ()
 3: merge (a: [1: 1], b: [1: 6])
 4: int$from_to_by (from: 1, to: 5, by: 2)
```

```
 5: mergesort (a: [1: 1 6 3 4 7])
 6: value$apply (item: mergesort, av: [1: [1: 1 6 3 4 7]])
 7: scan$mexpr (ac: [18:])
 8: listen (tyi: stream#[R, tty:], tyo: stream#[W, tty:])
 9: %quit ()
10: %dfail (thing: call#(desc#(array)$new, 0))
11: %snap (plnk: call#(desc#(array)$new, 0), ent: 777777#612371)
12: %linker ()
13: mergesort (a: [1: 1 6 3 4 7])
14: value$apply (item: mergesort, av: [1: [1: 1 6 3 4 7]])
15: scan$mexpr (ac: [13:])
16: listen (tyi: stream#[R, tty:], tyo: stream#[W, tty:])
17: %quit ()
18: _ctrlg_handler ()
19: merge (a: [1: 1], b: [1: 6])
20: int$from_to_by (from: 1, to: 5, by: 2)
21: mergesort (a: [1: 1 6 3 4 7])
22: value$apply (item: mergesort, av: [1: [1: 1 6 3 4 7]])
23: scan$mexpr (ac: [18:])
24: listen (tyi: stream#[R, tty:], tyo: stream#[W, tty:])
25: %base% ()

: frame(3)                          % This is the interesting frame

 3: merge 30055
        a:      [1: 1]
        b:      [1: 6]

        c:      [1..11265: 1 1 1 1 1 1 1 1 1 1...]   % Aha! I forgot
        ia:     1                                    % to increment
        ib:     1                                    % ia and ib.

: foobar = @ 3 c                        % Note that I can print
                                        % the object referred to
                                        % by the local variable c

=> [1..11265: 1 1 1 1 1 1 1 1 1 1...]

: foobar                                % and bind it to a new
                                        % identifier.

=> [1..11265: 1 1 1 1 1 1 1 1 1 1...]
```

```
% Here I escaped to TED, edited and recompiled.

: load("mergesort")

: a                                          % See if my test array is
                                             % still OK.

=> [1: 1 6 3 4 7]                            % Yes

: mergesort[int](a)

=> [1:]                                      % Good, no loop, but still
                                                % the wrong answer.

: %trace_to(merge[int])                      % Breakpoints go away when
                                             % you reload the procedure.

: restart()                                     % Get a clean stack

: mergesort[int](a)

Calling merge

merge (a: [1: 1], b: [1: 6])
mergesort (a: [1: 1 6 3 4 7])
-- next --                                   % I typed space
merge returns [1: 1 6]

merge (a: [1: 1], b: [1: 6])
mergesort (a: [1: 1 6 3 4 7])
-- next --                                   % Here too
Calling merge

merge (a: [1: 3], b: [1: 4])
mergesort (a: [1: 1 6 3 4 7])
-- next --                                   % And here
merge returns [1: 3 4]

merge (a: [1: 3], b: [1: 4])
mergesort (a: [1: 1 6 3 4 7])
-- next --                                   % And here
Calling merge
```

```
merge (a: [1: 7], b: [1: 1 6])                % Hmm, what's this nonsense?
mergesort (a: [1: 1 6 3 4 7])
-- next --                                     % I typed RETURN
                                               % to start a listen loop
: frames()

 0: listen (tyi: stream#[R, tty:], tyo: stream#[W, tty:])
 1: %enter_pause (first: true)
 2: merge (a: [1: 7], b: [1: 1 6])
 3: int$from_to_by (from: 5, to: 5, by: 2)
 4: mergesort (a: [1: 1 6 3 4 7])
 5: value$apply (item: mergesort, av: [1: [1: 1 6 3 4 7]])
 6: scan$mexpr (ac: [18:])
 7: listen (tyi: stream#[R, tty:], tyo: stream#[W, tty:])
 8: %base% ()

: frame(4)                                     % This should tell me what
                                               % I want to know

 4: mergesort 27443
        a:      [1: 1 6 3 4 7]

        aa:     [1: [1: 1] [1: 6] [1: 3] [1: 4] [1: 7] [1: 1 6] [1: 3 4]]
        e:      7
        naa:    [1:]
        i:      5

% Eureka! I'm appending elements to aa instead of to naa.
% I escaped to TED, edited and recompiled.

: load("mergesort")

: restart()                                    % Clean stack again

: a

=> [1: 1 6 3 4 7]
: mergesort[int](a)

=> [1: 1 3 4 6 7]                              % Success
```