I. Introduction to Idris

TABLE OF CONTENTS

NAME

Manuals - a guide to Idris documentation

SYNOPSIS

Idris Users´ Manual
- I. Introduction to Idris
- II. Standard Utilities
- III. File Formats
- IV. System Administration Guide

Idris Programmers´ Manual
- I. Whitesmithing
- II. Idris System Interface
- III. Programming File Formats
- IV. Idris Support Library

Idris Machine Interface Manuals
- I. System Generation Guide
- II. Idris System Interface
- III. Device Handlers
- IV. Device Driver Interface

FUNCTION

Idris is a comprehensive operating system that provides a uniform operating environment across a broad spectrum of computers. Modelled closely after the highly successful UNIX operating system developed at Bell Labs, it incorporates numerous facilities of proven worth for both naive and experienced users. It is sufficiently elaborate, as a result, that it is not easily described in just a few brief essays.

Hence the description of Idris is partitioned into three separate manuals: one for everybody who uses the system, one for those who must add programs to the system, and one for those who must be aware of the particulars of a given underlying machine. A different version of the Idris Machine Interface Manual exists for each family of computers sharing a given instruction set; those who wish to maximize portability will do well to ignore all instances of this particular manual and confine their study to the first two, which describe Idris facilities common to all machines.

The Idris Users´ Manual contains several essays, such as this one, covering the gross aspects of the operating system. Here one can find "manual pages", i.e. terse descriptions of the scores of standard utilities which are often sufficient in themselves to solve data processing chores. There is also a section describing any file formats presumed by one or more of the standard utilities, so that other utilities can be used to create or dissect them. Finally, there is a section summarizing the tools provided to the system administrator, with some guides to running a happy Idris environment.

The Idris Programmers´ Manual focuses on the facilities typically used only in the process of developing new utilities or applications programs, a process known to some as "whitesmithing". Programs such as the linker, librarian, and C compiler driver are documented here, as is the portable interface to all system calls and file formats of interest principally to programmers. A library of functions used extensively by the Idris standard utilities is also documented in this manual, to simplify the production of new utilities by users and to encourage more uniform practices.

Each Idris Machine Interface Manual begins by describing any peculiarities of the operating system implementation on the given machine. Particular attention is given to the "resident", which is the program that resides at all times in computer main memory to schedule resources and to carry out requests from other programs. Here is given information on how to configure the resident for a specific piece of hardware. The machine level interface to the system is described, and all device handlers supplied with the system are documented, to reveal their foibles and to serve as models for user written handlers. And finally, there is a section describing how to write new device handlers, which includes a set of manual pages describing the functions that may be called from a device handler or other user addition to the Idris resident.

BUGS

Part of the UNIX heritage is a tendency to provide documents that are exquisitely precise, but terse. The Idris project has fully embraced this approach to documentation, and has often achieved new heights of brevity, due to the constraints of an ambitious commercial enterprise. Everything you need to know about Idris is in these manuals, but it is often as hard to digest as grandma's brandied fruitcake.

Chew thoroughly.

NAME
Login - getting started with Idris

SYNOPSIS
```
login: <loginid>
password:
```

FUNCTION
Idris is an operating system that tries to be friendly to its users by maintaining a fairly uniform appearance across its many utilities and by trying to do something sensible for nearly any request made of it. As a consequence, it is not nearly as chatty as most "human engineered" systems, nor does it produce many or elaborate error messages. It does seem to strike a balance, however, that makes many people happy, be they bystanders or warriors in the data processing revolution.

It is an interactive system, which means it typically spends most of its time waiting for people to type things at it (as opposed to reading batches of card decks or monitoring the pressure gages in a refinery). It is also a multi-user system, which means it can talk to several people at once, and hence must have some way of telling them apart. So to get anything done with Idris, you have to first get its attention and then learn how to talk to it.

All this documentation presumes the existence of a "system administrator", who may be just another persona of the one and only user, but who will be treated as a separate entity, to keep discussions factored. The system administrator will be saddled with the chores that conventional interactive users should normally be relieved of (or denied). Candidates for system administrator are referred to the last section of this manual, which is devoted to that job.

At any rate, to start using Idris you have to: a) get a terminal logically connected to the computer, b) get Idris to issue a login process for that terminal, and c) get installed in the system a loginid and password for your personal use. See your system administrator for all of the above.

A "login process" is just a computer program, whose mission in life is to wait for a terminal to get logically connected and then write the invitation "login: " at the terminal and wait for input. The user wishing to login to Idris must correctly type a loginid, which is one to eight characters (usually letters, or at least printable stuff), followed by a carriage return (or linefeed on some terminals). If the loginid is known to the system, and if there is a password required for that loginid (there usually is), then the login program proceeds to write "password: " at the terminal and wait for another line of input; only this time the computer is careful not to show the characters of the password being typed at it.

If these hurdles are not both surmounted, the login program doggedly repeats its "login: " invitation; but upon success, the login program turns control over to whatever program the system administrator has assigned for use with that loginid. This is usually a very helpful program called, of all things, the "shell", or "sh" for short. The shell spends most of its time waiting for you to type a line at it, which it then endeavors to interpret as the name of yet another program to run on your behalf, possibly with some information passed to it about what files to use or what options to exercise. Unlike login, the shell does not just pass on the torch and quit; instead, it hangs around to print out any post mortem, if the other program dies an unnatural death, and to read still another request line from you. But more on that in a later essay.

The important thing to note is that each piece of information requested of you takes the form of "a line of text". This is much like the line one might type on a conventional typewriter, except that the computer looks on each keystroke as a separate character, even if that character calls for tabbing, backspacing, or returning the carriage to the left margin. Thus a line of text consists of zero or more printable characters, or spaces, or tabs, etc. up to a line terminator such as a carriage return or line feed. Since we all make mistakes, Idris permits you to do a modicum of editing of the stuff you type; it promises not to look at the line until the line terminator is typed. Up until then, you can take back the last character typed by typing a backspace (usually), which neatly erases the character, on a video terminal at least, and leaves you in position to retype the character or to erase even more. You can also scrub the entire line and start over by typing the at-sign ´@´ (usually), which causes an immediate return to the left margin as a signal that the previous information has been discarded. Those parenthetic "usually" hedges are a hint that you can even change which keys cause character and line delete, but at the risk of some confusion until you get used to the changes.

There are other things you can do via the terminal keyboard. If output is coming too fast for you to digest, you can suspend printing by typing a ctl-S (hold down the "control" key and type the letter S). You can resume output by typing anything else, but a ctl-Q will do the job innocuously, since neither ctl-S nor ctl-Q are passed on to the programs reading the terminal. Neither of these keys affect any program running on your behalf, except perhaps to delay it if it is writing to your terminal.

Other keys do affect running programs, however. You can simulate an "end of file" condition, needed to bring to a tidy conclusion a program reading what you type at the keyboard, by typing a ctl-D. You can bring most programs to an abrupt and untidy conclusion with an "interrupt", obtained by typing a DEL (sometimes represented by the RUBOUT or DELETE keys) or by striking the BREAK key. And to terminate a program with extreme prejudice, complete with abrupt cessation and a "core" file for post mortem examination, type ctl-\.

All of these magic keys, and a few more, are summarized in Section III of this manual, on the essay titled tty (usually written tty(III)). This essay describes all of the peculiar properties of a terminal when treated as a file.

While any program running under Idris has the right to bypass all of this keyboard interpretation, it is a rare one that does. Thus, no matter what program you are talking to, you have a uniform set of rules for framing your requests and for throttling its responses. You can even type ahead quite a number of characters, assuming you know what input is appropriate in the near future, and not worry about its being mislaid between programs.

Just what to say to each program is a strong function of the program, but there are a number of system-wide conventions. Among other things, there are standard conventions: for specifying the options to be exercised by a program, for selecting whole groups of files for processing by a program, and for selecting input and output files to be used by a program. The preoccupation with files stems from the fact that all long term memory in the Idris system takes the form of files of information, stored on disks or tapes as a rule. Programs are files, the data you care about are stored in files, even the information needed to track down files is stored in still other files. The stuff you type at the keyboard looks to most programs like just another file, albeit more transient than most, since files are almost always passed sequentially once through a program. Idris can best be viewed as an arena for the convenient creation and manipulation of files.

The login program gets you into that arena.

BUGS

Subtleties that are tempting to gloss over, but which may confuse the novice user are:

If the terminal displays garbage when you are expecting the "login: " invitation, there is probably a speed mismatch between terminal and computer. The login program is usually instructed to try a different speed each time an interrupt is received from the terminal; this can be made to happen even from a wrong speed terminal by striking the break key. If three or more tries fail to produce a clean invitation, see your trusty system administrator.

Typing a ctl-D to signal end of file from a terminal works by a bit of trickery that only comes off if the ctl-D is the first character on a line. In other positions, two ctl-D characters must be typed; this produces a partial last line which many programs find unpalatable anyway.

One non-printing character that frequently occurs in messages from the computer is the bell key (an ASCII BEL) or ctl-G. This causes a ding, beep, toot, or silence, depending upon the noisemaker available in the terminal. The login program rings BEL alot, as does Idris when it gets upset over something. Don´t be scared.

NAME

Commands - the shell game

FUNCTION

So, you've logged in successfully to Idris. Some sort of message has probably been typed at you, and the last thing displayed is a percent-sign and space "% " on a line by itself. What now?

Now, the system is waiting for instructions, usually called commands, or command lines. A command (slightly misnamed) is simply a request for Idris to do something. Almost always, commands are funneled through a program called the shell, or sh. The percent-sign (called a prompt) is output by the shell to indicate that it is ready to accept a command, which generally names a program that you want to run. The programs discussed in this essay are called standard utilities. From now on, when we mention a "utility", we simply mean one of these programs, whose name you give to the shell as a command. Whenever a command finishes execution, the shell will output another prompt, meaning that it is once again waiting for input.

You should be familiar already with the basics of talking to Idris. If not, re-read the preceding essay; the conventions described there apply to the shell just like to any other program. Here, only one reminder: Idris will not act on the line you are currently entering until you terminate it, which means you can edit the boots off of it, but which also means that you _must_ terminate it to get any response. Usually, you type a carriage return in order to terminate a line; depending on your terminal, however, line feed may serve the same purpose. We will use the single term _newline_ to refer to whichever terminator you must use. A newline ends each line of input you type; likewise, a newline ends each line of text output or stored by the computer.

NAMING COMMANDS

In the following examples, lines preceded by a prompt are what you should type to try out the example; other lines are the system's response to the input given. The newline at the end of each input line can't be seen, but it must be there, nonetheless.

Just to test the water, try typing:

```
    % date
```

Idris should respond with something like:

```
    Mon Sep 07 12:18:49 1981 EDT
```

Like it? The system will provide other interesting information, too. As a further example, try:

```
    % who
```

That should net you a listing of who is currently using the system, like this one:

```
tana     Sep 07 12:20:29 console
pjp      Sep 07 08:01:46 tty1
cnh      Sep 07 09:35:22 tty2
djb      Sep 07 09:54:53 tty4
```

The first field on each line is the name each user specified to login. Next is listed the time at which the user logged in, followed by the name of the terminal location being used.

COMMAND ARGUMENTS

The uses of date and who that we´ve just seen are examples of the simplest kind of command line, one giving a command name and nothing else. To be really useful, however, commands generally need to supply other information, as well. This is done with arguments, which are simply additional words on the command line, separated by spaces from the command name and from each other. (More precisely, an argument is a series, or string, of characters separated from its surroundings by whitespace.)

One simple utility that takes arguments is called echo. Try it out:

```
% echo boo!
```

The response explains its name:

```
boo!
```

More generally, echo will display all of its arguments, exactly as they appeared on its command line. That´s all it does:

```
% echo what use is this?
what use is this?
```

Though perhaps unexciting right now, echo is quite useful in other situations, as you´ll discover later.

If you make a mistake in entering a command name, the shell will tell you of it. For example, typing:

```
% data
```

yields the response:

```
data: not found
```

After displaying the warning message, the shell forgives and forgets; when the prompt reappears, it´s ready to accept another command.

FLAGS

The utilities so far discussed are obviously all quite different from each other, and Idris provides dozens more, of even greater diversity. Often, however, it´s desirable to alter the operation of a utility in some relatively small way, without the expense of creating another, separate program. Most utilities accept a special kind of arguments, called flags, that change some aspect of what the utility does, or select one option among many possible ones. Flags are almost always given before any other arguments, and invariably begin with a ´-´ or a ´+´, so that everyone involved can recognize them. They are usually optional.

Instead of the command line for echo shown earlier, try entering:

```
    % echo -m what use is this?
```

The response may surprise you:

```
    what
    use
    is
    this?
```

The flag you gave accounts for it: -m causes echo to separate its arguments with newlines instead of spaces when it outputs them, thus making each word print on a separate line. (A newline, you may recall, is the character that ends a line of text.) This still may not appear very useful to you, but it is.

If you specify a flag that a utility doesn´t recognize, the program will display a hint to remind you of what the legal flags are (assuming you have read the manual page for the program and merely need reminding). Entering:

```
    % echo -q
```

results in the message:

```
    usage: echo -[m n] <args>
```

which says that the program echo accepts the flags -m and -n (we haven´t explored -n yet), followed by <args>, a series of arguments, as explained above. If you forget what flags a utility accepts, you can force it to output a usage message by typing the flag -help. Thus

```
    % echo -help
```

will also result in the message shown above.

CREATING FILES

Doing anything significant with Idris pretty much depends on learning about files. A file can be viewed as a named storage place for information. Most of the time, a file contains text, so creating a file under Idris amounts to getting some text into the computer and saving it under the name you want. An easy way to do that is with the text editor e. A later essay explains e in detail; for now, we´ll go over the bare minimum you need in order to enter text.

First, start the editor running,, by typing:

```
    % e -np
```

Don´t forget the space between the ´e´ and the ´-´. (The -np tells e to output a ´>´ prompt when ready for a command, and a line number prompt when ready to append text.) e will respond with its command prompt, a ´>´. Once the prompt has appeared, type an ´a´, followed by a newline; this is the editor command to begin "appending" text. Henceforth e will prompt for text lines with a line number, and save any text you enter in an internal area called a "buffer", until you type a period on a line by itself. Then, e will prompt you for another editing command:

```
   >a
    1   any text you like --
    2   for as many
    3   lines
    4   and lines
    5   and lines
    6   as you like.
    7   .
   >
```

Try entering some text (any text) as described. After you´ve ended it with a period, you can perform a variety of editing operations on it. The next step, though, is just to write what you´ve entered to a file. To write it to a file called trial, for instance, you would type:

```
    >w trial
```

The editor will then reassure you that it has written the file by telling you the number of characters it wrote. (Of course, you can give any filename you like after the w editor command; a later essay will discuss file naming rules and conventions.) As a further exercise, try adding some more text to what you´ve already entered. Just type another append command, and end the text with a second ´.´. Then write out the buffer again, this time to a different file, say new:

```
   >a
   7   then some more
   8   lines on the end
   9   .
   >w new
```

About now you may be seized with an understandable urge to look at what you've typed in. To get e to oblige you, type:

```
    >1,$p
```

The 'p' editor command prints some part of the editor's buffer. In this case, you asked to see all of it (from first line '1' to last line '$'), so e should have displayed the text from both a commands. Finally, you can quit the editor, and get your shell prompt "% " back, by typing a 'q' editor command:

```
    >q
```

MANIPULATING FILES

You should now have created two files, the first of which contains the text you entered with the first append command, and the second of which contains the text from both append commands. First of all, to find out what files you currently have, list their names with the ls utility:

```
    % ls
```

If you named your files as shown, the response will be:

```
    new
    trial
```

which are indeed the two files we asked to be created, listed alphabetically. Now that we know they're out there, what else can we do with them? Perhaps the most natural thing is to look at their contents again, just to make sure. The simplest program for doing so is called cat (for concatenate files, legend has it). To use it, just name the file (or files) you want to examine, and cat will output exactly what they contain, no more, no less:

```
    % cat trial
    any text you like --
    for as many
    lines
    and lines
    and lines
    as you like.
```

Or, if you want better-looking output (like for a printer), use the utility pr, which divides its output into pages, and outputs at the top of each page a heading containing the name of the file being output, and other information. It, too, takes one or more files as arguments:

```
    % pr trial new
```

(pr also can do many things that this essay won't explore, as indeed can all of the utilities we will discuss from here on. If you're feeling adventuresome, sample the second section of this manual, which contains separate manual pages for all of the standard utilities that Idris provides. First read the page at the front called Conventions, then turn to the manual page for the program you're interested in. Like escargot or squid, the typical manual page is hard to get a taste for, but exquisite once you've got it.)

Once you can get text into files and back out to a display or printer, the next item of interest is moving or copying files internally, which you can manage with similar aplomb by using three basic utilities: mv (for move), cp (for copy), and rm (for remove). Both mv and cp take two arguments, the name of the existing file you want to manipulate, and the name of the file you want to create. mv moves the original file to the new one, which amounts to renaming it. The original name is deleted. Hence the following would move new to perm:

```
% mv new perm
```

cp creates a second copy of the original file, and leaves the original intact. To make a copy of trial called transcript, you might type:

```
% cp trial transcript
```

The ls command would then reveal three files:

```
% ls
perm
transcript
trial
```

If a file already exists with the new name you give to cp or mv, it will be overwritten with the contents of the file being moved or copied; the already existing contents will be lost forever. To delete files on purpose, use rm, which will remove the files you give it as arguments. To rid yourself of transcript, for instance, type:

```
% rm transcript
```

Use this utility with care, as once a file has been removed, there's no recovering it.

The utilities we've explored so far fall mostly into two categories. A few are informational, like ls or who. The rest of them move files around in one way or another. Each of these last utilities has a source of input and a destination for its output. Some, like cp or mv, use files for both. Others, like cat and pr, take a file as input, and output to your terminal. (We'll see shortly that this distinction really isn't very important.) All of them access files without altering their contents.

FILTERING TEXT

Clearly, though, instead of merely transferring a file from one place to another, a program could make some presumably useful change to the input read before outputting it. In fact, Idris provides a number of utilities, called "filters", that do just that. Used separately, each filter reads text from a source of input -- usually the files named on its command line -- performs some operation on the input text (like sorting it), and outputs the result. Used together, filters permit suprisingly powerful text processing, without any need for additional programming.

Filters come in two kinds: some examine their input character by character, and change it according to what they find. A second group look at their input line by line, and alter it in the same way. The line-oriented filters called last and grep (honest) may be used to output only part of each of their input files, suppressing the remainder. To look at just the last three lines of trial, for instance (skipping all the preceding lines), you might type:

```
    % last -3 trial
```

and get

```
    and lines
    and lines
    as you like.
```

Be sure you understand that the contents of trial are unchanged. Filters never alter their input files; rather, they read text from the source of input named, change the text internally, then pass the altered text on to their output. The -3 is a flag consisting solely of a number that tells last how many lines it should output. last always outputs some group of lines at the end of each input file.

grep is more flexible: it searches for lines containing a specific string, and outputs only the lines that do. The string to be looked for is given as the first argument; the files to be searched are then listed afterwards. The following would output all the lines in trial containing the word you:

```
    % grep you trial
    any text you like --
    as you like.
```

The string searched for need not be a single word; it may be any string of characters. For example:

```
    % grep "and lines" trial
```

searches trial for the entire string enclosed in quotes, and outputs only the two lines that contain all of it (and not the one containing "lines" alone):

```
    and lines
    and lines
```

The quotation marks cause the shell to treat everything inside as a single argument, so that the space shown, which would normally separate two arguments, is simply included as part of the single argument inside the quotes. And of course, since it´s the shell that interprets quotes, you can use them with any command you please, whenever you want to turn off the special meaning the shell would usually give a character, and cause it to be counted as part of an ordinary argument.

Other filters change their input in subtler ways. An especially useful one is a utility called sort, which reads its source of input line by line, sorts the lines read, and then writes them to its output. The following would sort the lines in trial and output the result to your terminal:

```
% sort trial
and lines
and lines
any text you like --
as you like.
for as many
lines
```

As you can see, by default (that is, if you don´t specify any flags), sort outputs the lines it reads in ascending order. The basis for the ordering is the internal value (ASCII value) of successive characters on each line; leaving details aside, this ordering works as you would expect. Letters sort in alphabetical order: ´a´ sorts before ´b´, and so on. Uppercase letters sort before lowercase, and digits sort before either.

With a few flags and more effort, you could as readily sort files based on only parts of each line (instead of the whole thing), or sort them by numerical value, or sort them into a different order. To sort together the lines in trial and perm, and output them in descending order, this suffices:

```
% sort -r trial perm
```

The -r flag causes sort to reverse its default ordering.

A second filter often used in combination with sort is uniq. uniq searches its input for repetitions of the same line, and deletes the duplicates, outputting only one copy of each input line. To output trial with its duplicate line deleted:

```
% uniq trial
any text you like --
for as many
lines
and lines
as you like.
```

Nice as it is, uniq has a limitation that might seem fairly serious: it recognizes only adjacent repetitions of a line. If the second occurrence of a line doesn´t immediately follow the first one, it won´t be seen as a duplicate (or be deleted). In the example just given, the duplicate lines happened to be adjacent; obviously, though, they wouldn´t always be. For uniq to be more generally useful, its input should be sorted beforehand.

Since identical lines will sort to the same place, duplicate lines will be adjacent in the sorted output, whatever their position had been in the original, unsorted file.

But how can we get the output of sort to be the input to uniq? One way is to tell sort to put its output into a file, instead of sending it to your terminal. Then the sorted file can be input to uniq:

```
    % sort -o sorted trial
    % uniq sorted
```

When sort sees the -o flag, it uses its next argument as the name of the file where it will place its output. (The -o flag is therefore said to include a value: it is always followed by a string that gets interpreted as an output filename.) Many of the filters we will examine accept a -o flag (and accompanying filename), so using the flag is a fairly adequate way of making one filter talk to another.

STDIN and STDOUT

The shell, however, provides a far more powerful one. We've seen already that a filter is just a command that copies text from its source of input to an output destination, performing some operation on the text as it's transferred. The shell generalizes this notion by assigning all filters (and all programs, for that matter) a standard source of input called STDIN, and a standard destination for output called STDOUT. By default, these are assigned to your terminal: input comes from your keyboard, and output goes to the terminal screen.

Some utilities, like cp or mv, do almost nothing with STDIN and STDOUT; instead, they deal with files explicitly named on their command line. Filters, however, can do a great deal indeed. Whenever you run a filter without naming any files on its command line, the filter will read its input from STDIN. What's more, unless you specify a -o flag, a filter writes its output to STDOUT. Look back at the first examples for sort and uniq -- you'll note that both command lines shown name an input file (and so don't read from STDIN), but don't contain a -o flag (and so <u>do</u> write to STDOUT).

If STDIN and STDOUT always designated your terminal, they wouldn't be worth much, except for the odd occasion when you felt like entering the input to a filter directly from your keyboard. Their power lies in their ability to be redirected (redefined) to designated files. By using a special argument on a command line, you can cause the standard input for a program to come from a file; similarly, you can direct its standard output to go to a file. For instance, these lines have the same effect as the last example shown:

```
    % sort trial > sorted
    % uniq < sorted
```

The '>' on the first line redirects the STDOUT of sort to be a file named sorted, which is then created as sort runs. The redirection happens before sort is started; it neither knows nor cares that it is no longer sending output to your terminal. The '<' on the second line redirects the STDIN of uniq to be the file sorted that the previous line just created. Once again, uniq reads its STDIN without knowing whether it desginates a terminal or a file; either is equally acceptable. A third special symbol, ">>", causes the STDOUT of a program to be appended to a file. The previous contents of the file are preserved. To sort the lines in perm, and add them to the end of the file sorted, you could type:

```
    % sort perm >> sorted
```

PIPELINES

Nifty, huh? Well, praise in departing. The shell provides an additional facility called a "pipeline" that makes even more effective use of STDIN and STDOUT. A pipeline causes two programs to be run simultaneously, with the STDOUT of one program redirected to be the STDIN of a second one. If both programs are filters, then this means that, while the first program is writing text to its standard output, the second is reading the same text from its standard input. Intermediate files like sorted become unnecessary, and the last example can be rewritten again:

```
    % sort trial | uniq
```

The '|' specifies a pipeline. Specifically, it causes the STDOUT of the program to its left (here, sort) to be redirected as the STDIN of the program to its right (which is uniq). Data flows through the pipeline, from left to right, with sort providing the source of the data, and uniq merely altering it as lines of text pass through the pipe. (Pipelines, incidentally, may be used to link together more than two programs. Each stage in a longer pipeline works exactly like the first one: the STDOUT of the lefthand program is redirected to be the STDIN of the next one.)

TOOLMAKING

Using pipelines, utilities can be combined to perform what are normally complex tasks in processing text. Here's an extended example of how such an impromptu "tool" might be constructed. This one is built around a character-oriented filter called tr, which looks for a specified set of "target" characters in its input and either deletes them or translates them to characters in a second set. Suppose, for instance, we wanted to eliminate all punctuation from the file trial we made earlier, and leave ourselves with a uniform list of the words trial contains. A first shot at doing so might be:

```
    % tr !a-z trial
```

The first argument to tr specifies the target characters it is to look for in its input; here, the string "a-z" names the characters from lowercase a through lowercase z, and the preceding '!' causes tr to look for any characters except the ones named. tr will then delete all occurrences of the target characters from its input, and output what's left:

Since identical lines will sort to the same place, duplicate lines will be adjacent in the sorted output, whatever their position had been in the original, unsorted file.

But how can we get the output of sort to be the input to uniq? One way is to tell sort to put its output into a file, instead of sending it to your terminal. Then the sorted file can be input to uniq:

```
    % sort -o sorted trial
    % uniq sorted
```

When sort sees the -o flag, it uses its next argument as the name of the file where it will place its output. (The -o flag is therefore said to include a value: it is always followed by a string that gets interpreted as an output filename.) Many of the filters we will examine accept a -o flag (and accompanying filename), so using the flag is a fairly adequate way of making one filter talk to another.

STDIN and STDOUT

The shell, however, provides a far more powerful one. We've seen already that a filter is just a command that copies text from its source of input to an output destination, performing some operation on the text as it's transferred. The shell generalizes this notion by assigning all filters (and all programs, for that matter) a standard source of input called STDIN, and a standard destination for output called STDOUT. By default, these are assigned to your terminal: input comes from your keyboard, and output goes to the terminal screen.

Some utilities, like cp or mv, do almost nothing with STDIN and STDOUT; instead, they deal with files explicitly named on their command line. Filters, however, can do a great deal indeed. Whenever you run a filter without naming any files on its command line, the filter will read its input from STDIN. What's more, unless you specify a -o flag, a filter writes its output to STDOUT. Look back at the first examples for sort and uniq -- you'll note that both command lines shown name an input file (and so don't read from STDIN), but don't contain a -o flag (and so <u>do</u> write to STDOUT).

If STDIN and STDOUT always designated your terminal, they wouldn't be worth much, except for the odd occasion when you felt like entering the input to a filter directly from your keyboard. Their power lies in their ability to be redirected (redefined) to designated files. By using a special argument on a command line, you can cause the standard input for a program to come from a file; similarly, you can direct its standard output to go to a file. For instance, these lines have the same effect as the last example shown:

```
    % sort trial > sorted
    % uniq < sorted
```

The '>' on the first line redirects the STDOUT of sort to be a file named sorted, which is then created as sort runs. The redirection happens before sort is started; it neither knows nor cares that it is no longer sending output to your terminal. The '<' on the second line redirects the STDIN of uniq to be the file sorted that the previous line just created. Once again, uniq reads its STDIN without knowing whether it desginates a terminal or a file; either is equally acceptable. A third special symbol, ">>", causes the STDOUT of a program to be appended to a file. The previous contents of the file are preserved. To sort the lines in perm, and add them to the end of the file sorted, you could type:

```
    % sort perm >> sorted
```

PIPELINES

Nifty, huh? Well, praise in departing. The shell provides an additional facility called a "pipeline" that makes even more effective use of STDIN and STDOUT. A pipeline causes two programs to be run simultaneously, with the STDOUT of one program redirected to be the STDIN of a second one. If both programs are filters, then this means that, while the first program is writing text to its standard output, the second is reading the same text from its standard input. Intermediate files like sorted become unnecessary, and the last example can be rewritten again:

```
    % sort trial | uniq
```

The '|' specifies a pipeline. Specifically, it causes the STDOUT of the program to its left (here, sort) to be redirected as the STDIN of the program to its right (which is uniq). Data flows through the pipeline, from left to right, with sort providing the source of the data, and uniq merely altering it as lines of text pass through the pipe. (Pipelines, incidentally, may be used to link together more than two programs. Each stage in a longer pipeline works exactly like the first one: the STDOUT of the lefthand program is redirected to be the STDIN of the next one.)

TOOLMAKING

Using pipelines, utilities can be combined to perform what are normally complex tasks in processing text. Here's an extended example of how such an impromptu "tool" might be constructed. This one is built around a character-oriented filter called tr, which looks for a specified set of "target" characters in its input and either deletes them or translates them to characters in a second set. Suppose, for instance, we wanted to eliminate all punctuation from the file trial we made earlier, and leave ourselves with a uniform list of the words trial contains. A first shot at doing so might be:

```
    % tr !a-z trial
```

The first argument to tr specifies the target characters it is to look for in its input; here, the string "a-z" names the characters from lowercase a through lowercase z, and the preceding '!' causes tr to look for any characters <u>except</u> the ones named. tr will then delete all occurrences of the target characters from its input, and output what's left:

```
    anytextyoulikeforasmanylinesandlinesandlinesasyoulike%
```

Not quite what was wanted. Even the shell prompt ended up in the wrong place (any idea why?). A better approach would be to translate the punctuation characters to something more innocuous, like spaces, that would then neatly separate the words in the list. You can do so by giving tr a -t flag, naming the set of output characters to which the target characters are to be translated. After the -t, tr expects the next string on the command line to name the desired output characters. Since we want the output character to be a space, we have to enclose it in quotes; otherwise, tr would never see it, because spaces normally separate arguments on a command line, and are not passed as arguments themselves. All of which adds up to the following revised command line, and a better result:

```
    % tr !a-z -t " " trial
    any text you like for as many lines and lines and lines as you like %
```

As you can see, every string of characters in trial not containing a letter has been translated to a space. (Remember that each line of trial was ended by a newline, each of which was translated.)

tr fits in very well with sort and uniq. A common application is using tr to translate a text file into a list of words like the one just created, then using sort and uniq to rework the list into useful form. Creating a glossary -- a list of all the words occurring in a file -- is easy. Believe it or not, the following pipeline will output a sorted list of the words used in trial:

```
    % tr !a-z -t "\n" trial | sort | uniq
    and
    any
    as
    for
    like
    lines
    many
    text
    you
```

Let's examine it piece by piece. The tr command is almost the same as the last one we used, except that now we want to translate every string that is not part of a word into a newline, instead of a space. (That way, each word is passed to sort on a separate line.) Newlines, like spaces, can't be directly specified as arguments, since typing one will terminate the line you're trying to enter. So we use the characters "\n" to stand for a newline; tr will recognize what we mean.

It writes the list of words to STDOUT, which is redirected to be the STDIN of sort. sort reads the list from STDIN (because no input files are named on its command line), sorts it, and outputs it to its own STDOUT, which is redirected to be the STDIN of uniq. uniq also reads its input from STDIN, eliminates duplicate lines (i.e., multiple occurrences of the same word), and writes unique lines to STDOUT. Because the STDOUT of uniq hasn't been redirected, the final list of words is output to your terminal.

A problem with the glossary-maker as it stands is that it deals only with lowercase letters. We could make it handle uppercase, and ignore case distinctions, simply by putting another tr command at the start of the pipeline, this one to translate uppercase letters to lowercase:

```
    % tr A-Z -t a-z trial | tr !a-z -t "\n" | sort | uniq
```

Note that this new tr command works slightly differently from the previous one. Because the output set of characters is now the same length as the target set, tr translates each occurrence of a target character into the corresponding character in the output set, instead of compressing strings of target characters into a single output character. And notice, too, that the second tr command in the pipe, since it no longer has any input files, reads its STDIN just as sort and uniq do. With this final change, you've produced a surprisingly useful tool, assembled on the fly from pre-existent parts. Making tools like this one, quickly and easily, is what Idris is all about.

You may have wondered by now whether programs other than the filters we've looked at can be used in a pipeline. The answer is yes -- nearly all utilities that normally output something to your terminal are really writing it to STDOUT, and most utilities that usually process text files will read STDIN if no files are given on their command lines. pr, for instance, can function as a filter, and might be used to gussy up the list of words produced in the last example. It will paginate text read from STDIN just as it would the text read from a file:

```
    % tr A-Z -t a-z trial | tr !a-z -t "\n" | sort | uniq | pr
```

This pipeline is the longest you'll be seeing in this essay. But don't be overawed -- adding pr at the end of the pipe changes nothing at all in the rest of the command line.

Our introductory tour of the shell and the most basic standard utilities ends here; the capabilities of the programs you've seen, though, do not. Only a small minority of Idris utilities have been presented, and even these (as well as the shell) can do many things that this essay did not cover. Read on in this series of tutorials, and look at the detailed manual pages on the utilities to be found in Section II. You should by now be acquainted with the power and ease of use that Idris provides; getting to know it better will only increase your awareness of them.

NAME

Editing - the text editor e

SYNOPSIS

e -np <file>

FUNCTION

A text editor is a program used to create and edit a file of "text", that is, a file containing information entered by a user at a keyboard. Some common examples of text are documents, letters of correspondence, computer programs, lists of data, and so on.

e is used whenever you want to create a text file from scratch, or modify text that already exists. In either case, it will allow you to manipulate your material in a variety of ways, to transform time-consuming editing tasks into quick and efficient ones. For example, e allows you to:

1) add new lines to an old file of text,

2) delete lines you no longer want,

3) search for typing errors and correct them,

4) shift the position of various portions of text,

5) display what the file looks like at a given point in time, allowing you to verify changes,

6) make a permanent record of the file.

All of the text in this manual, and all of the programs that go with it, were typed in to this editor (or to some earlier version). It is a very powerful and important tool.

How does e handle all of these jobs? By nature, e is "interactive", meaning that it depends upon you, the user, to interact with it if anything is to be accomplished. This means that you must learn how to talk to e in order to get a reasonable response. You will get e´s attention and cooperation if you use "editor commands" to tell it what you would like it to do. For each task there is a specific command entered by the user (these commands are given in detail in this essay); if the command is entered correctly, e will do its best to execute it for you. Once you know its language, you will find that dealing with e is not only practical, in terms of saving valuable work time, but also easy and enjoyable.

e is really nothing more than a "tool", so the more you use it, the more skill you acquire in handling it. This implies that the way to exploit e´s potential is simply to make frequent and pragmatic use of it. This tutorial will introduce you to the commands, provide examples to illustrate their functions, and suggest exercises that will develop your skill in using them. We will begin with the basic utilities, and then move on to fancier usages of e. Feel free to experiment with the files you create in the test exercises -- since they are just practice files, you can´t do any harm by playing with them. After completing the tutorial, you will have a good understanding of how e is used; for further details and for the complete set of commands e offers, refer to the manual page e in Section II of this manual.

Since the editor is a program like any other program you might want to run, you´ll <u>first</u> need to know how to get started on the computer and then, when you´re ready to edit some text, how to enter into e. In response to the shell prompt "% ", you would type:

```
    % e -np
```

followed, as always, by a newline (carriage return). The flags "-np" tell e to precede all lines of text with a line number, and to prompt for all input, much like the shell. More experienced users often prefer more silent operation, and so leave off the flags.

If you´re starting from scratch, that is, if your file doesn´t exist yet and you need to create it, then you may want to go ahead and give your file a name. Let´s suppose you give your file the name new. Type:

```
    % e -np new
```

Be sure to follow the e and the -np with spaces before typing in the name of the file you want to edit. The editor´s response should be simply:

```
    ?
```

followed by its prompt ´>´ on the next line. This is the editor´s way of saying that the file new doesn´t yet exist and you need to add material to it. You can enter e with either of these commands; if you don´t specify the filename now, you can do it at the end of the session.

Before going any further, it will be helpful to understand just how e handles text. e makes use of a temporary storage area, or workspace, known as a "buffer". The buffer is just an empty region made available when you run e. When you enter text it is written into the buffer; any further changes to the text are also made there. Then, when you´re done, the final product can be copied from the buffer to a permanent place. From then on, whenever you use e to modify the file, e makes a copy of the file into its buffer. You´ll see that working with a buffer is a comfortable way to experiment with your text without harming any previous work you may have done.

To get the editor´s attention once you know what you want to do, you´ll enter a specific command. e obeys commands that you give it <u>if</u> <u>and</u> <u>only</u> <u>if</u> you enter the command in a form it can understand. An editor command generally consists of a single character, typed in lowercase, followed (of course) by a newline. (From now on, we´ll assume that you won´t forget to complete every command or text line with a newline.)

APPENDING

Your first job, then, when creating text from scratch, will be to use the a command, which allows you to "append" or add lines of text to the buffer. The procedure looks like this:

```
>a
1   (lines you want to enter)
2   .
```

Type the ´a´ on a line by itself, and the editor will commence prompting by "line number". The prompt "1\ \ \ " indicates that the next line you enter will be taken as text to be added as the first line of the buffer. The next line will have a "2\ \ \ " prompt, and so on. When you´ve finished entering lines, type a ´.´ (period or dot) in response to the prompt. The ´.´ signals the editor that you have finished appending. The editor responds by reverting to the ´>´ command prompt; without the ´.´ the editor just keeps adding whatever lines you type to the buffer. If you want the file new to contain the beginning stanza of a poem, for instance, you might type:

```
>a
1   Tweedledum and Tweedledee
2   Agreed to have a battle;
3   For Tweedledum said Tweedledee
4   Had spooled his rice new rattle.
5   .
```

Now the buffer contains the four lines:

```
Tweedledum and Tweedledee
Agreed to have a battle;
For Tweedledum said Tweedledee
Had spooled his rice new rattle.
```

The line containing the ´a´ and the one with ´.´ are not recorded, because e recognized that they were simply instructions and not part of the text. Similarly, the prompting line numbers are also not recorded with the text.

CHANGING

Notice that you made a mistake when typing line 4, and instead of "spoiled his nice" you actually typed "spooled his rice". You can correct these spelling errors with the c or "change" command. Type:

```
>4c
4   Had spoiled his nice new rattle.
5   .
```

That says: "Take line 4, and change whatever is currently written there to what I have just written here". Or, in other words,

```
    Had spooled his rice new rattle.
```

is replaced by the new line you typed in with the c command. Again, the c and the '.' are not recorded. If you accidentally made a mistake in typing the command, the editor will let you know. If instead of typing "4c" you typed "c4", for example, e quickly throws a '?' at you. It's confused (can you blame it?) and is saying: "Sorry, please try entering that command again!" It will not go into long explanations of just how you botched things, but that's part of e's nature. A quick signal is all you get, but you'll probably see right away what you did wrong. Try again. Once you've changed line 4, you may be ready to add more material to your file; type another a command and e will place any new lines in the buffer.

WRITING

At this rate, you could keep appending and changing lines all day long, but chances are that you'll be interrupted and have to move onto some other project eventually. Or, perhaps you've added everything that you intended to add. In either case, you may need to save your text, so that it's available when you want to either look at it later on or make further changes to it. The w command "write" takes a picture of the buffer and copies its contents into the file new (the name you originally gave to e as your new filename, remember?). Type:

```
    >w
```

or

```
    >w new
```

to specify that you want your file to be named new. You'll need to supply the name if you didn't specify it earlier, when you first entered e. (Notice that the w command doesn't require a '.' at the end.)

The write command will destroy any previous contents of a file that you write to, so be careful in selecting filenames, and double check the filename you typed before terminating the line.

After you type your command, the editor prints out a number, such as:

```
    115
```

that tells you how many characters e copied from the buffer into your file. If you were now to look at the contents of new, they would be identical to what's currently in the buffer. Since you're still in the editor, the buffer does not get erased; if you want to add more lines, keep on adding. The point to note is that the file itself changes only when you execute a w command; the buffer is simply a copy of what's in the file, and not the file itself. Whenever the buffer looks exactly like what you'd like to see in the file, finish with a w. As a matter of fact, it's a good idea to write out the contents onto a file frequently, in case you're distracted or the computer loses power or some other unforeseen event occurs. That way, whatever work you've done up to that point will be saved (for later retrieval). As soon as you save your text with the w command, you're ready to end your editing session. You previously entered the editor with the e command, if you recall, and to leave the editor you must also issue a command -- the q command "quit". Type:

```
>q
```

and the prompt "% " appears on the screen. This is the official indication that you're back in the shell, no longer in the editor.

An interesting thing happens when you attempt to leave the editor before you've saved your text with w. Let's say you've typed in a business letter and you forgot all about w and typed q instead. Well, the editor knows that you've inserted text in the buffer and it doesn't want it to be lost unless you're <u>sure</u> you don't want it. So the q command, in this case, doesn't cause the editor to vanish but instead prints out:

```
are you sure?
```

meaning: "Are you sure you want to leave the editor without saving your text?" In other words, the prompt question reminds you that you haven't issued a w command. Just type anything other than 'Y' or 'y' in response then, on the next line type the w command. Now if you try to quit, the editor won't raise a ruckus but will quietly disappear.

If you respond with a 'Y' or 'y' to "are you sure? ", e assumes that you know what you are doing; the q command causes the editor to vanish in spite of the fact that your material isn't saved.

EXERCISE 1:

Try creating a file called abc and then writing it out. After entering e, add some text with a, then write it out with w. Now quit the editor. After the "%\ " appears, you can use the command

```
% cat abc
```

to see if the file really contains what you think it should. If it doesn't, try the sequence again.

ENTERING

Let's return to new, the file containing stanza 1 of the nursery rhyme. You're ready to enter the remaining stanza now, so if you're still in e, just type the e or "enter" command:

```
>e new
```

new is copied into the buffer and the system prints the number of characters it contains. This command (which is deliberately identical to the name of the editor) clears the buffer before copying new into it, so if anything was stored there it is promptly deleted. Now you can append the last stanza.

Or, perhaps you discover that a co-worker has already typed the remainder of the poem into the file poem. Instead of retyping it yourself, you can read the file poem into the buffer, right after stanza 1, with the r or "read" command.

```
    >r poem
```

A character count appears to indicate the total number of characters in this file; now the buffer contains:

```
    Tweedledum and Tweedledee
    Agreed to have a battle;
    For Tweedledum said Tweedledee
    Had spoiled his nice new rattle.

    Just then flew down a monstrous crow,
    As black as a tar-barrel;
    Which frightened both the heroes so,
    They quite forgot their quarrel!
```

To save the text, type a w and now the complete poem is contained in new. As you will see shortly, you can read in a file at any point in the buffer, not just at the end.

The e command can be used in two instances; first to enter the editor, then as a command within the editor when you want to enter another file into the buffer. For example, let´s suppose that the two-stanza poem is still in the buffer. If we type:

```
    >e inventory
```

then the file inventory is entered into the buffer, causing the current buffer contents to be deleted. Again, if you haven´t saved the changes made to new, you would be warned at this point with "are you sure?\ " before the buffer is cleared.

Since you may enter or read more than one file into the buffer in a given editing session, you may need to know which file you´re actually working on. You can easily find out by typing:

```
    >f
```

The f command displays the currently "remembered" filename, in this case "inventory". (The "remembered" filename is the last one you entered with an e or r command.) If you want to modify inventory and then call it "inventory2", you could cause "inventory2" to become the remembered filename instead of "inventory" by typing:

```
    >f inventory2
```

Now when you go to write out the file, e will remember that "inventory2" is the correct filename. Thus, the f command is used in two ways: to let you know which file is the currently remembered one, or to let you set the remembered filename to whatever name you choose.

PRINTING

No doubt, you will frequently want to see what lines are contained in your text. The p command "print" will show you what you've got, by outputting the specified lines to your terminal. This is useful, especially when you're making changes frequently and can't remember exactly how a line reads. If new is the file you're editing, type p preceded by a line number to display a given line:

```
    >1p
```

should display the contents of line 1. As a matter of fact, you can step through the file line by line by typing a newline; each newline will display the following line in the buffer. To display the entire poem, type:

```
    >1,9p
```

This prints out the range of lines 1 through 9. Any range could be specified with the formula "m,np", where 'm' and 'n' are decimal numbers referring to specific lines in your text. The same results will be had with:

```
    >1,$p
```

The '$' is a special character representing the last line in the buffer. So this command, like the previous one, displays all lines, first to last. This is the most convenient way to output the buffer's contents: by using the '$' you don't have to remember the actual number of lines present (a silly and often impossible task).

EXERCISE 2:

Create a second file following the procedure given in EXERCISE 1. Be sure to write it out. While this file is in the buffer, enter the original file created in EXERCISE 1, using the e command. Check with the f command to see which file is currently remembered. Next, use the r command to read the second file back into the buffer.

Display the entire buffer with the "1,$p" command to see if your r succeeded; you should see the contents of your original file immediately followed by the contents of the second.

ADDRESSING

You´ve noticed by now that e keeps track of "lines" of text. When you access or display portions of the buffer, then, you are really addressing selected lines. e numbers lines, starting with line 1, as soon as you append material, and keeps adding line numbers as you enter lines of text. But these numbers are not recorded with the text; they´re just a convenient way to talk about lines in the file in order to make changes.

In fact, most editor commands contain line addresses at the beginning to let e know which lines you want to alter in some way. Fortunately, the text editor is very proficient at helping you get around. First of all, it keeps track of the "current line" and "last line" of the buffer. The "current line" refers to the most recent line affected by a command that you entered; the symbol for "current line" is the character ´.´ (period or dot, hereinafter known as "dot"). And you´ll remember that ´$´ represents the last line. You can type:

```
>.p
```

in order to display the current line; \&´.´ is not always easy to predict, so you have to realize that different commands set ´.´ to different locations. As an example, when you enter a file, dot is set to the last line in the buffer. If line 7 needs correcting:

```
>7c
7   (line as it should read)
8   .
```

causes ´.´ to be set at line 7, the last line you altered in some way. Now type:

```
>.p
```

to print the current line, which indeed is the newly changed line 7. When you enter many of the commands, the editor will assume that you want the command to apply to the current line ´.´ if you don´t specify any other line number. So simply typing:

```
>p
```

will do the same thing; it assumes you are referring to ´.´ since you didn´t provide any other number. Whenever you lose track of ´.´ and want to know what line it´s referring to, type:

```
>=
```

or:

```
>.=
```

and the line number will be displayed. Of course, you can also just print the current line and see what number precedes it. To output only the last line of the file, type:

```
>$p
```

\&´$´ and ´.´, then, are methods of addressing specific lines which could also be addressed by specifying decimal numbers. As you get more familiar with e, however, you´ll find yourself preferring them to numbers whenever possible.

There are other ways to move around, too. It has been mentioned that "m,n" will address a range of lines, as in:

```
>2,11p
```

which displays all lines beginning with line 2 and ending with line 11, inclusive. An address may also consist of more than one part, or "term". If two terms of an address are separated by a plus ´+´ or minus ´-´, then the addressed line is whatever line happens to be the sum or difference of the two terms. For instance,

```
>7+3c
```

means that line 10 is to be changed. Or:

```
>$-3p
```

displays the third line from the end of the buffer. If ´+´ or ´-´ precedes the first term, as in:

```
>+3p
```

then e assumes that ´.´ is meant to be another term, to which you want to add 3. This command is identical to:

```
>.+3p
```

Displayed is the third line beyond the current line. If you were to type a ´+´ or ´-´ by itself, you are actually referring to "+1" or "-1". So:

```
>++++p
```

means "move forward four lines from the current line and print it". Either ´+´ or ´-´ can also follow an address term, with the same effect:

```
>8---p
```

means "back up three lines from line 8 and print it"; you´re now at line 5. All of these methods can be combined at will; choose whichever are most palatable to your editing tastes. Sequences frequently used, in addition to "1,$p" mentioned earlier, are

```
>.,$p
```

or:

```
>$-10,$p
```

The first outputs all lines from the current to the last line, while the second displays the last 11 lines of the buffer.

EXERCISE 3:

Experiment freely with line addressing, using the alternate methods described above. Note what happens if you try:

```
>5,7,12p
```

then try:

```
>7,4p
```

and see what you get. You may use 0 (zero) as a line address with some of the commands but not all -- it´s a fictitious place you can put things after, but can´t do things to. Typing:

```
>0r header
```

allows you to read the file called header at the start of the buffer; what was previously in the buffer now follows the text you have just read in, and is thus renumbered. Notice how the line numbers change.

Observe that

```
>0a
```

has a different effect than

```
>0c
```

e won´t let you do the last one, because it calls for a change to the fictitious line 0.

INSERTING

Now that you´ve mastered moving around in the buffer, we can introduce more editor commands. The i command is used to insert text before a specified line. If the file contains the three lines:

```
Day 1:
Day 3:
Day 4:
```

and you wish to enter "Day 2:" into the list in its appropriate place, type:

```
>2i
2   Day 2:
    3   .
```

The editor interprets the command to mean: "Go to line 2, and insert in front of line 2 the text given". Now the file contains 4 lines, with line 2 containing "Day 2". "Day 3:" has moved down to line 3. You can insert more than one line at a time, and when you've finished, you'll find that '.' is set to the last line inserted. Typing:

```
    >1i
    1   (lines to be inserted)
    2   .
```

will insert a line at the start of the buffer. Note that i and a are very similar: each is typed as a line by itself, followed by text to be added, followed by a '.' alone on a line. There is, however, a fundamental difference -- you insert text before a given line, while you append after a given line.

DELETING

We've spoken of changing lines and inserting new ones, but what if you need to get rid of some altogether? This situation could arise when portions of your file are no longer needed and should be erased; or perhaps when you've got redundant material as a result of inserting new lines. You could use c to change the lines, then type nothing in their place (i.e., just a dot); but there is a more convenient way. The d command will "delete" or remove specified lines from the buffer. For example, if a file read:

```
    Four score and seven years ago
    Our fathers brought forth upon this continent
    A new nation
    A new nation, conceived in liberty,
```

Somehow in your typing, you duplicated the phrase "A new nation" (certainly Mr. Lincoln would not approve). You'll need to delete line 3, and

```
    >3d
```

will do it. The current line is now

```
    A new nation, conceived in liberty,
```

In other words, '.' refers to the line originally after the last line deleted. Line 4 moves up to become line 3. If you delete '$', however, the current line is set to the new '$':

```
    >7,$d
```

will delete all lines from line 7 to the end of the buffer, and will put '.' at line 6. Be careful when using d, as deleted text is forever gone from the buffer.

You may recognize by now that the editor commands provide a range of options to the user in the creation or modification of text. For example, deleting and inserting text when done sequentially will produce nearly the same effect as changing the text with the c command. Or, to enter text from scratch you can use either the a or i command. The choice is yours, and no doubt you will develop preferences as you make use of e. But each command has its own modus operandi and you must learn its behavior. Note, for instance, what happens when you try

```
    >2a
    2   .
```

and do then the same procedure replacing a with i and then with c. List your file after each (with "1,$p") to see the results; they may not be what you´d expect!

EXERCISE 4:

With your practice files, insert new material and then delete portions of the text. Keep track of ´.´ to see how it moves around with different commands. Notice that you only use one line address with i, but with d you can specify a range.

MODIFYING

We´ve talked quite a bit about changing files by modifying whole lines at a time -- either appending, deleting, changing, inserting, or reading them. Quite often, however, you´ll simply need to change one or two letters or words within a line. If you´ve misspelled a certain word or perhaps forgotten to insert a comma or period, you shouldn´t have to retype the entire line -- correcting the single letter (or word) is much more efficient. The tool e uses for this task is the s or "substitute" command. The command works by substituting one string of characters on a line for another; its command line names both strings. For instance, if our line reads:

```
    O, for a draught of vinage. . .
```

you´ll need to correct "vinage" to read "vintage". The command to do so would be:

```
    >s/vinage/vintage/p
```

and is interpreted like this: "Substitute for the letters "vinage" the letters "vintage". Another more general way of expressing it would be:

```
    substitute for / this string / that string /
```

"that string" is also known as a "replacement string" as it replaces characters as they <u>are</u> on the line with the characters that <u>should</u> be on the line. If you don´t specify line addresses, s assumes you want it to work on the current line. You may also specify a range of lines, as in:

```
>1,10s/this/that/p
```

This command examines each of the lines 1 through 10 and changes the phrase "this" to "that" whenever it occurs. (The p at the end is our familiar command which displays the last line changed -- this is one of the places when two commands can be written together.) However, the substitution only changes the first (leftmost) occurrence of "this" in a given line; if there is more than one occurrence, you need to add a g to execute the command "globally" on every occurrence of "this". Thus:

```
>1,$s/usa/USA/gp
```

will examine the entire file, changing usa from lower to upper case everywhere it finds it, and print the last line changed. (If no changes are made, you will get the old faithful "?" instead.) It´s also possible to replace something with nothing:

```
>s/,//p
```

will delete the first comma from a line. The "//" in the example, a slash followed immediately by another slash, refers to a string containing nothing at all, not even a space. Spaces, in fact, are typical inhabitants of an s command line:

```
>s/thedog/the dog/p
```

Or, a quicker way to do the same thing, once you get cockier:

```
>s/ed/e d/p
```

It is a good idea to print the result of each substitution, particularly such a cocky one, since the wrong part of the line may match the pattern. Remember that it is the leftmost matching string that wins; and if there is more than one with the same starting position, then the longest string is chosen.

CONTEXT

Consider the following situation. You find out after typing in a rather lengthy text that you need to change the word "unicorn" to "dragon". The problem is remembering which lines contain "unicorn". No small task, in fact impossible. The solution lies in performing what´s known as a "context search" on your file. This mysterious-sounding feat is a way of locating a particular line by searching for a specified string of characters appearing on that line. You execute the search by placing slashes around the string you want to nab:

```
>/unicorn/
```

Given this "line number", e will search forward in your file and arrive at the next line that contains "unicorn"; it will even print it out for you, since the default command is p. Now perform the s command:

```
>s/unicorn/dragon/p
```

A context search always starts at the line just after the current line and moves forward; if it arrives at the end of the buffer and still hasn´t found the desired string, it "wraps around" to line 1 and keeps on looking. If it arrives at the line where you started and still hasn´t found it, the standard complaint "?" indicates an unsuccessful search -- the string was not to be found.

Significant here is the understanding that /string/ determines a line number; it points to a particular line where that string occurs. Thus it can precede a command just like any other address:

```
>/unicorn/s//dragon/p
```

accomplishes in one command what the previous two steps did: look for the next line containing "unicorn", substitute in that line "dragon" for "unicorn", and print out the corrected line. The "//" refers to the string specified previously, so you needn´t type it out again. e remembers the last string specified. It also remembers the last replacement string you entered. so if you want identical changes on more than one line, just type s to get your results (or "sgp" if you desire all occurrences changed, and the line printed out).

This remembered string comes in handy on another occasion, too. If there is more than one line containing "unicorn" but you don´t want to change that word to "dragon" in every instance, you´ll need to look at each line and then decide if you want to change it. Once you´ve specified your string and e finds the first occurrence of it, just type:

```
>//
```

to arrive at the second line containing an occurrence of unicorn and to print the line. Repeat this to locate the third line containing an occurrence, and so on. If you use them all up, by changing each "unicorn" to "dragon", for instance, the context search eventually finishes with a "?".

Finally, you may also specify a range of lines using /string/. If you want to display a paragraph whose first line begins with "START" and whose last line contains "END",

```
>/START/,/END/p
```

will work fine.

You can even search backwards, if you need to. Instead of "/string/" you´ll use "?string?" or "%string%" and the search will start at the line preceding the current line, move back to the next nearest occurrence of "string", and display the line. If no match has been found before reaching the start of the buffer, the search wraps around to the last line, $, and keeps searching. Again, if no match is found, the error message "?" appears.

Note that "??" serves the same function in a backwards search as "//" in the forward search -- it searches backwards for the last expression specified, saving you from typing it again. Search strings are highly useful items.

GLOBAL

Another command making use of search strings is the g command -- g searches the buffer "globally" for all lines matching the string you specify, and then executes another command for each of those lines:

```
>g/INTERIM/d
```

will search for every line containing "INTERIM" and then delete it. If you then list the buffer, all of those lines will be gone.

If you want to locate every line in your file that contains an occurrence of the word "company", and then change all occurrences of "company" to "corporation",

```
>g/company/s//corporation/gp
```

will make the change and print all the corrected lines. You can follow a g command with any number of commands, separated by semicolons ´;´. (The rule about one command per line was a white lie.)

```
>g/micro/s//macro/;.+3s/bits/bytes/;.-6sg;.,$p
```

The v command acts as the converse of g; it will search for any lines <u>not</u> matching the specified string, and perform commands on those lines. Thus:

```
>v/./p
```

prints out every line not containing a period.

EXERCISE 5:

Practice the s and g commands using simple examples to start. Try substituting ´?´ for ´.´ in each line of your file:

```
>1,$s/./?/
```

Did you forget to get every occurrence of ´.´ within every line? Do the same command, only add "gp" at the end. This same task can be done with g, and it would look like this:

```
>g/./s//?/gp
```

The first g assures that the command searches every line in the buffer; the last g sees that it makes the substitution on every occurrence of the string in each line that fits the match. The difference is that the second version prints <u>every</u> line changed, while the first one prints only the <u>last</u> line changed.

REGULAR

It's now a convenient time to introduce a new concept -- the notion of a "regular expression". Simply speaking, a regular expression is a shorthand notation for a sequence of characters contained in a line. In other words, the regular expression is a (possibly) abbreviated representation of certain characters you want to pinpoint. The characters are said to "match" the regular expression. Regular expressions are used most frequently in commands calling for searches or substitutions, because they provide a method for specifying character strings.

The most obvious regular expression is one you're already familiar with: a character string which is an identical match to a string present in a file line. Remember the context searches?

```
    >/xyz/
```

implies a search for a line containing the same string "xyz". So, any ordinary character can be considered a "regular expression" which matches that very character when it occurs on a line.

There are other regular expressions which are not ordinary at all; in fact, they are somewhat magical and are regarded as "special" by e. One of them is '?'. Instead of specifying particular characters, you can point to "any" character at all with '?'. For example:

```
    >s/?/M/
```

will search until it arrives at any character on the line (including a space) and will replace it with 'M'. In essence, it simply replaces the first character. Or:

```
    >/t?e/
```

will match either "the", "tie", "toe", "tee", "t5e", "t!e", or any other three-character string that begins with 't' and ends with 'e'. It simply uses the first match it finds.

A '^' (caret) following a character matches zero or more occurrences of that character; that is, if your line reads:

```
    baaaa!
```

then

```
    >/aa^/
```

would match the entire string of 'a's. The logic goes like this: "Find an 'a', and then see if it's followed by another 'a', and another 'a', etc." So /aa^/ would have matched either one 'a' or thirty 'a's, if that many were present in the string.

A '*' (asterisk) is a bit more ambitious: it matches anything at all! That means, if you type:

```
    >/m*n/
```

the match will be any string that starts with ´m´ and ends in ´n´. Try replacing "jolly" with "jovial" in the following line:

```
    Jupiter greeted the jolly, joyous Juno.

    >s/j*y/jovial/p
```

will produce:

```
    Jupiter greeted the jovialous Juno.
```

It didn´t work as desired, because the ´*´ will match the longest possible string. The first ´j´ it encountered was in "jolly" but the last ´y´ was in "joyous", so it chose the longer match. The correct version would have read:

```
    >s/j*ly/jovial/p
```

To delete everything on a line:

```
    >s/*//
```

that is, substitute for everything, nothing. To replace all contents up to (and including) a ´:´, type:

```
    >s/*:/replacement/
```

The special character ´^´ mentioned earlier has another important usage. When it is the leftmost character of the regular expression, it points to the beginning of the line. If you specify:

```
    >/^TUT/
```

then only if "TUT" are the first characters in the line will the match occur. If our line reads:

```
    KING TUT
```

then the search attempt won´t find the match. Similarly, ´$´ signifies the end of a line. For example:

```
    >/TUT$/
```

will match the string "TUT" only if it occurs at the end of the line. Thus:

```
    >/^Hurrah$/
```

locates a line whose entire contents are "Hurrah".

If we use brackets ´[´ and ´]´ to enclose characters, that too has a special effect. For instance:

```
    >/[abcdef]/
```

will match <u>any</u> <u>one</u> of the characters in the bracketed list, that is, an 'a' or a 'b', etc. through 'f'. Another way to express this is:

```
    >/[a-f]/
```

On the line "7 days remain to teach 12 women 5 ways to dance.", the command:

```
    >s/[0-9]/N/p
```

will change '7' to 'N'. \&'7' was the first character in the list that was found. (If you do the same substitution again, which character changes?)

A tricky inversion of this is:

```
    >s/[!a-zA-Z]/?/p
```

means locate the leftmost character that is not a letter, change it to '?', then print the line.

The '!' before the list of characters means: "Match any character <u>except</u> one in the bracketed list." Thus the match could find a numeric, a punctuation character, or any other character <u>not</u> in the class [a-zA-Z].

These regular expressions are all valuable editing tools, but they can cause problems in your substitute commands. Let's suppose the line you need to change has some of these magical creatures on it:

```
    ZOUNDS!*?^$*!
```

You want to replace the '*' with a ':', and your first attempt seems reasonable:

```
    >s/*/:/p
```

Reasonable, yes, but magic characters don't respond to reason. As you can see, the contents of the entire line have been replaced by a ':'. e has given '*' special powers -- to match a string of any length -- and now you're asking it to regard '*' as an ordinary character. The solution is to insert <u>another</u> magical character, the backslash '\', in front of the special character. This takes away the magical meaning and forces e to treat \&'*' like any other character. Thus:

```
    >s/\*/:/p
```

will work. Since '\' is a special character too, you must apply the same rule when deleting a '\' from a line; precede the '\' with yet another '\':

```
    >s/\\//p
```

The backslash can also be used to turn on magical properties of otherwise tame characters:

\b is a visible way of typing a backspace,

\t becomes a horizontal tab,

\r becomes a carriage return,

\v becomes a vertical tab,

\f becomes a formfeed, and

\n is the only way to talk about newlines inside character strings, since a literal newline is invariably taken as the end of the command line.

Finally, you can represent even more exotic unprintable characters by writing up to three digits after a backslash -- a practice you rarely need to engage in, but it´s provided for. If you´ve typed a tab key to skip four spaces between columns, your line resembles:

```
    1981    COST    ASSETS
```

To remove the spaces and replace them with ´+´, type

```
    >s/\t/+/gp
```

and you get:

```
    1981+COST+ASSETS
```

Warning: when making substitutions on lines that contain lots of special characters, the procedure can become quite complicated. If things get too messy, (if for instance you find yourself trying unsuccessfully to edit a string of backslashes!) just give up and start the command over. In extreme cases, you may want to abandon the s command and retype the line from scratch, using c instead.

FANCY

We´ve now illustrated most of e´s capabilities, but a few more fancy tricks remain which deserve brief mention. One is the ´&´ (ampersand) character. This becomes special only when it´s used on the right-hand side of a substitute command, where it is transformed into the character string which the left-hand side matched. If we want to put the word "perhaps" in parentheses, for instance, we could type it out twice, surrounding it by parentheses in the replacement string:

```
    >s/perhaps/(perhaps)/p
```

But the easier solution is:

```
>s/perhaps/(&)/p
```

which substitutes for the string "perhaps" the same string, surrounded by parentheses. You can even repeat the match:

```
>s/very /&&/p
```

which turns, for example, "very\ big" into "very\ very\ big". (Note the careful inclusion of a trailing space.)

If you have an unfinished phrase you want to complete, the ampersand again saves typing:

```
>s/grand,/& I thought to myself,/
```

which might result in:

```
She looks grand, I thought to myself, due to her recent success.
```

If you want to break lines or perhaps join them, use the regular expression "\n" mentioned above, which represents a newline. To break the line:

```
The computer will be unavailable; my apologies!
```

try:

```
>s/; /;\n/p
```

(Again note the space.) You'll now have the two lines:

```
The computer will be unavailable;
my apologies!
```

Or, to join a line with the next, just delete the newline:

```
s/\n//p
```

i.e., substitute for the newline, nothing; the second line is moved up to the current line. (Note: you will not actually see the lines joined as one, unless you print them both at once, but when you write out your file, they will be merged.)

e provides a fancy method for keeping track of certain lines of text in the buffer; you can give some lines a "tag" (any lowercase letter) which is used to address the line. The command is called k, or "know" the addressed line to be 'x' (the lowercase letter you choose). For example, if we want the line

```
WARNING: HIGH VOLTAGE
```

to be tagged with the letter 'v', type:

```
>/WARNING/kv
```

This looks for the line we want, giving it the tag ´v´. To address the line from now on, type "´v" (no need to worry about its location in the buffer; e remembers it as "´v"). This can be useful when you´re doing lots of modifying, (i.e., ´.´ will be changing frequently) but also want to keep a certain line readily accessible.

Another tagging mechanism can be used with regular expressions in the s command by surrounding the regular expression with "\(" and "\)". The characters matched by the expression can then be included in the replacement string under the tag name "\(#" where # is any digit from 1 to 9; the numbers correspond to the order of the expressions in the first part of the command. An example will elucidate matters. If we need to reverse the order of the words in the line "boy=girl",

```
    >s/\(*\)=\(*\)/\(2=\(1/p
```

prints out:

```
    girl=boy
```

It divides the line into two regular expressions, tags them with "\(" and "\)", then reverses the second one with the first one. Such "tagged" regular expressions are hard to type correctly, and seldom used, but very helpful when you need to do some messy edit repeatedly.

The m or "move" command is used to shuffle portions of text around from one location to another. Basically, you move a set of lines from where they currently reside to immediately after another line. If we want to move the last ten lines of the file to the beginning, say:

```
    >$-10,$m0
```

In general terms, the command goes like this:

```
    (range of lines) move (to after this line)

    >10,´vm3
```

will move all lines from 10 to ´v (our WARNING: HIGH VOLTAGE line) to <u>after</u> line 3. \&´.´ is set to the last line moved. This command also takes a little practice to master, but is definitely worth the investment.

The t command is similar to m; it makes a "twin" of a set of lines and places the duplicate lines immediately after a specified line. If lines 1 to 3 contain:

```
    This software is
    a product of
    Whitesmiths, Ltd.
```

and we want to repeat that message at the end of our document, type:

```
    >1,3t$
```

Note that both t and m change the positions of lines, but t makes a copy of them, while m moves the original lines themselves.

We spoke earlier of e as a significant tool which will do editing jobs (of all sorts) quickly and thoroughly. A further extension of its capability lies in the creation of "scripts", i.e. canned sequences of editing commands stored in files. Suppose you want to perform the same editing task on a number of files. Here, for instance, are the commands needed to discard trailing spaces and to fold long lines at the end of a word:

```
    >g/  ^$/s///
    >g/\(????????????????????????????????????????[! ]^\)  ^/s//\(1\n/g
    >w
```

The first command looks for one or more spaces at the end of a line, then deletes them. The second one looks for a long line and inserts newlines in place of spaces at convenient intervals (honest). The third command writes the file back out. Now this is a hard sequence to type correctly even once, let alone repeatedly, so the best thing to do is use the editor to prepare its own input! If you type this in as ordinary text and write it to a file called, say, fixup, you can then launder a file called, say, needsit, by running the editor:

```
    % e needsit <fixup
```

and the job is done. Not only that, it can be done over and over again with a minimum of typing -- you have made a useful tool.

There are many more things that the editor can do, and much more that you can do with just the features introduced here. Read the manual page for e to see what´s there, then practice what you have learned. Soon, you will develop a style of editing which best suits you; the more advanced features you can refer back to only when you need them.

Good luck!

NAME

Files - reading and writing data

FUNCTION

So far, little has been said directly about the nature of files under Idris. This was intentional, in part because it is important to understand how to interact with the shell and the editor to facilitate exploring the properties of files; but it also illustrates that you can do quite a lot of work without knowing much about the underlying machinery of reading and writing.

Most of the examples have, in fact, been in terms of keyboard input and display output to a terminal, hardly the sort of thing one associates with the term "file". And yet the shell makes it easy for a program to read a file instead of keyboard input, or to divert output to a file instead of to the display, so there must be some important way in which terminal I/O (input and output) can be made to look identical to file I/O.

The common ground is the "text stream".

If a program expects to read its input just once, sequentially from beginning to end, then it hardly matters whether the data is typed in on the fly or copied out of a file; if the program has to browse around in a data base, that´s a different story. And if a program produces its output in one sequential pass, from beginning to end, then it hardly matters whether the data is directed straight to the display or copied into a file for later display; if the program must back and fill to generate the output, that too is a different story. As it turns out, a large number of useful programs do perform such stream or sequential I/O, and hence are "filters" equally usable with files or with people sitting at terminals.

Another consideration is whether a program deals with the kind of characters that can be easily typed at a keyboard, and meaningfully written to a display, or whether it produces all the possible binary codes that can be represented in a single "byte" or character position. Sometimes a program must communicate with other programs with such binary codes, that are hard for terminals to represent or for people to understand, but once again this is remarkably rare. (Perhaps it is more honest to say that the Idris system makes a point of avoiding binary I/O.) At any rate, human consumable data is called "text", and the text stream is the lingua franca of the Idris world.

To further standardize the representation of text streams, Idris always deals with the American Standard Code for Information Interchange, known as ASCII, so that a lowercase ´a´ is recognized as such by any program that cares. This means that character codes may be mapped on the way in, and unmapped on the way out, for non-ASCII I/O devices, or "peripherals". Even within the ASCII community, some folks end a line with a carriage return followed by a linefeed, some with a linefeed followed by a carriage return, some with just a carriage return, and some with just a linefeed. In the last case, the linefeed is usually called by its alias "newline"; and this is the case that Idris has standardized on. All text streams, and all text files, consist of zero or more "lines", each terminated by a single newline.

There are a few additional rules about how long lines can be and what can be in them, but these are less important. See the manual pages called ASCII and text in Section III of this manual for more detailed information.

The important thing to remember is that I/O to terminals, other peripherals, and files can usually be made to look the same if it is in terms of text streams. The shell underscores the importance of this by providing each program a standard input stream STDIN, a standard output stream STDOUT, and a standard error reporting stream STDERR; and it makes STDIN and STDOUT dead easy to redirect from the terminal to files or other peripherals. Often, these streams may be arbitrary binary codes and need not be text, but text is always acceptable. This is why pipelines and file redirection are so powerful.

FILESYSTEMS

It is not enough to be able to read and write terminals, however, nor is it enough to be able to read and write all the peripherals on a given computer as if they were interchangeable data streams. The Idris system proper needs at least a hundred different files to hold all its programs and working data; assigning one file to each peripheral would be unthinkable. A disk drive can usually hold the contents of many different files, if only there be some way to administer the available storage space. The way Idris does it is to impose a structure on disk storage (or on other high speed, random access storage) called a "filesystem"; if the disk is large enough, it may even be chopped up into more than one "logical device", each of which can have a filesystem structure imposed on it.

The System Administrator is responsible for laying out filesystems. What matters for now is that an Idris system can have one or more filesystems available or "on line" at any given time, that each filesystem can support one or more "directories" or collections of files, and that each file within a filesystem is recorded with a number of useful attributes. Among the more important attributes of a file are its contents (of course), its size in bytes, and the name (or names) by which it can be accessed.

And the way to find out about the various attributes of files is with the standard listing utility ls. You have already used ls without flags to determine the names of files you have created:

```
% ls
abc
new
perm
poem
sorted
trial
```

Now try the long form listing with -l:

```
% ls -l
total    6
-rw-rw-rw-  1 pjp        36 Oct 16 16:27 abc
-rw-rw-rw-  1 pjp       250 Oct 16 16:25 new
-rw-rw-rw-  1 pjp       104 Oct 16 16:25 perm
-rw-rw-rw-  1 msk       135 Oct 15 09:27 poem
-rw-rw-rw-  1 pjp       176 Oct 16 16:27 sorted
-rw-rw-rw-  1 pjp        72 Oct 16 16:27 trial
```

What a surfeit of information! Over on the right are the filenames, as before; but now to the left is the date and time the file was last modified, and to the left of that is the number of bytes in the file. To the left of that is information concerning who owns the file, how many different names it has, and what access permissions are recorded for the file -- these attributes will be covered later. There is lots of whitespace in the middle to leave room for owner loginids of up to eight characters and file lengths of up to 16 million bytes.

DIRECTORIES

But where are all the other files? Somewhere must be stored files with names like cat, echo, ls, and sh, to hold the various standard utilities, not to mention the rest of the hundred-odd Idris files mentioned above. The answer is, they're in other directories. Unless told to do otherwise, ls confines itself to listing files in the "current directory", which is typically your personal collection of files. At login time, you were placed in this personal directory (i.e. it was made the current directory) before your shell was given control. So far, that has been the only part of the filesystem you've known how to access directly.

Often it is enough. The usual convention under Idris is to give each user a personal directory, to which other people might even be denied access if security warrants it, where each is free to think up filenames without fear of collision with others. Only when you must cooperate with others in a common directory, or when your collection of files becomes large enough to require some internal structure of its own, do you have to worry about the directory structure of a filesystem.

To see where you live in the directory structure of your system, use the pwd utility (to "print working directory"):

```
% pwd
/usr/pjp
```

The response will be some funny sequence of slashes and terse names like the one shown. This says that the current, or working, directory is called pjp (which is often the same as your loginid, but need not be), and that pjp is a subdirectory within the parent directory called usr, and that usr is a subdirectory within the fundamental or "root" directory of the entire system. The list of slashes and names /usr/pjp is called a "pathname", because it traces a path from a known place (the root in this case) to a given filesystem entity (the working directory pjp in this case).

Thus, all files within the system are organized into a "tree", or branching hierarchy, because any one can be reached by tracing a path from the root (of the tree) to the appropriate entity; directories are nodes, or branching points along the way. So the files you have been working with have pathnames like /usr/pjp/poem and /usr/pjp/abc, even though you have known them by their family names, shorn of slashes.

By keeping track of your current working directory, Idris permits you this convenient shorthand. The rule is: a pathname that begins with a slash is called an "absolute pathname" and always traces a path from the root; whereas any other pathname traces a path from the current directory. So poem and abc are acceptable ways of naming files if you are working in the directory /usr/pjp, and the absolute pathnames /usr/pjp/poem and /usr/pjp/abc are acceptable and have the same meaning anywhere.

Any "link" in the pathname can be up to fourteen characters long (additional characters are simply ignored), and may consist of any characters except a slash (of course) or an ASCII NUL (which has the value zero and is fortunately hard to type). Convention imposes much stricter rules, however. Anything other than printable characters within pathnames can be difficult if not impossible to get past the shell to various utilities. It is rarely possible to abbreviate directory names, so long ones quickly become tedious to type. And it is foolish to use names obscure in meaning or hard to type, such as "0001I10". A later essay in this section describes some shorthand that the shell supports for filenames; this often suggests a useful discipline for naming groups of files within a directory.

You can make a "subdirectory" of your very own by using the mkdir utility:

```
    % mkdir poems
```

This creates an initially empty directory called poems (or /usr/pjp/poems). To see what you now have:

```
    % ls -l
    total    7
    -rw-rw-rw-  1 pjp          36 Oct 16 16:27 abc
    -rw-rw-rw-  1 pjp         250 Oct 16 16:25 new
    -rw-rw-rw-  1 pjp         104 Oct 16 16:25 perm
    -rw-rw-rw-  1 msk         135 Oct 15 09:27 poem
    drwxrwxrwx  2 pjp          32 Oct 16 16:30 poems
    -rw-rw-rw-  1 pjp         176 Oct 16 16:27 sorted
    -rw-rw-rw-  1 pjp          72 Oct 16 16:27 trial
```

Note that poems has somewhat different permissions than the other files, including a leading ´d´ to indicate a directory, but it is in many ways a file just like all the others.

What can you do with the new subdirectory? Well, you can copy files into it:

```
    % cp poem poems/poem
    % cp new poems/new
```

and so on. Now try

```
    % ls -l poems
```

and see what you get. You can also make poems the current working directory by using cd:

```
    % cd poems
    % pwd
    /usr/pjp/poems
    % ls
    new
    poem
```

LINKS

You can get back to your original directory one of three ways: 1) cd with no argument usually returns you to your "home" directory, the one you were put in after login; 2) cd with the absolute pathname /usr/pjp will get you there from anywhere; or 3) cd with the relative pathname .. will climb up one level of the directory hierarchy, because .. is always a local alias or "link" pointing at the parent directory. The .. was put there quietly by mkdir, when it made the directory poems, along with another link called, simply, . (dot or period) which is an alias for the directory poems itself. Neither of these names is listed, nor any other links that begin with a dot, unless you give ls the -a flag:

```
    % ls -al
    drwxrwxrwx  2 pjp           64 Oct 16 19:50 .

    total    2
    drwxrwxrwx  2 pjp           64 Oct 16 19:50 .
    drwxrwxrwx  3 pjp          144 Oct 16 17:25 ..
    -rw-rw-rw-  1 pjp          250 Oct 16 19:50 new
    -rw-rw-rw-  1 pjp          135 Oct 16 19:50 poem
```

The links . and .. are placed in every directory as a matter of convenience and for uniformity. The alias . is useful whenever you want to talk about the current directory and don't want to have to specify its absolute path name. It is the default argument to ls, for instance, as seen more clearly in the last example. And .. is a useful back link for relative navigation, as with the cd above. You could also refer to /usr/pjp/trial, from /usr/pjp/poems, as ../trial (got that?). These are, in fact, the only loops permitted in the pure tree structure of the Idris filesystem, so that utilities which climb over directory subtrees know how to stay out of trouble.

To be more explicit about the underlying machinery: All of the information about a file is stored in a central place (called an "inode" for bizarre reasons). Each directory entry merely provides a named link to this central place, so there can be any number of pathnames, or aliases, or links leading to the same file. The number just to the left of the owner loginid counts the number of links to a file. Note that /usr/pjp/poems has two links, the second one being /usr/pjp/poems/., of course.

You already know how to erase a link; the remove utility rm does the job. The file actually disappears only when the last link to it is erased. And there is a utility for explicitly creating aliases, called ln for link. Suppose, for instance, you didn't want two copies of poem, but merely two ways of referring to the same file:

```
    % rm poem
    % ln ../poem poem
```

Try this, then use ls to inspect the results. Perhaps by now you have guessed that the move utility mv merely performs a link, a la ln, to establish the new name, then removes or "unlinks" the old name.

PERMISSIONS

The only remaining unexplored field in the long format ls listing is the cryptic "-rw-rw-rw-" and its ilk. These are highly abbreviated notes on what various people are permitted to do with and to a given file. There are three kinds of access permission: 1) read permission 'r' means that the file or directory may be read, 2) write permission 'w' means that the file may be written, or that the directory may have links added or deleted within it, and 3) execute permission 'x' means that the file may be executed (as a utility or other program), or that the directory may be scanned in the process of tracing a pathname. There are also three classes of people: 1) the owner is the user whose loginid matches that displayed by ls with the -l flag, 2) the group is the user or users whose groupid matches that displayed by ls with the -lg flag, or 3) everyone else. The System Administrator can tell you what group you are in.

Any program that attempts to access a file first is put in one of the three classes above, then has its permissions checked against the appropriate rwx group displayed (reading left to right for owner, group, and everyone else). No permissions, no access. As you can see, most files are created with read/write access for all concerned, and directories are fully accessible. In a tighter environment, your home directory might deny scan permission to anyone but yourself, so that anyone outside your directory subtree can never get inside it to access your files, regardless of their individual permissions.

You can add or delete permissions, for files that you own, by using the chmod (for "change mode") utility. To deny write permission to anyone but yourself for the file poem:

```
    % chmod -wgo poem
```

Try this, followed by ls to see the result once again. And read the manual page in Section II for chmod, to see what all those flags mean. It is also possible to change the owner of a file, but this is a function reserved to the System Administrator, for various obvious and subtle reasons.

There are two or three more permissions not described here, called set userid, set groupid, and save text, which are generally of interest only to System Administrators setting up special utilities. They are mentioned in passing on the manual page for chmod in Section II; at least one of them is clever enough to have been patented; but they are beyond the scope of this essay.

There are no explicit limits on how large any one file can get, short of the 16 million byte maximum size, nor are there any per user limits imposed on how much disk space you can use. A file must, however, fit entirely within one filesystem, which could be quite small if written on a diskette, or it could be as large as 32 million bytes. What matters, when space is at a premium anyway, is that disk space within a filesystem is handed out in 512-byte chunks, called "blocks". If you write 500 one-byte files, you are going to consume 500 blocks (more actually, because you need space to represent the large directory as well).

The "total" line at the head of each long format ls listing is a guide to the space conscious user. It sums the size, in blocks, of all the files listed (overestimating if there are multiple links, by the way). To find out how much space is left in the filesystem you are using, try the "disk free" utility df:

```
% df
/dev/rm3:
free blocks: 6318 / 16320
```

This says that there 6318 blocks available, in a filesystem that occupies a total of 16320 disk blocks. It also, by the way, tells you the physical device name of the disk that the filesystem is written on -- a piece of information the innocent user seldom needs to know, nor care about. As mentioned before, there can be many filesystems made available within the same hierarchy, all with different sizes and different amounts of free space. Aside from a few firewalls (such as files and links not bridging filesystems), you are seldom reminded of these physical boundaries.

PHYSICAL

When you do have to be aware of the underlying peripherals, however, Idris provides a nice mechanism for accessing them. Certain files are called "block special" or "character special", because they cause actual I/O devices to twitch when read or written. They bridge between the world of named files and the world of physical I/O.

A block special device is one that can be treated as an ordered sequence of 512-byte blocks, whose blocks can be read and written in any order. This is also known as "random access" or "direct access". Disks usually can be made to look like this, even if their natural block size is smaller or larger. Tapes can be also, but usually to a lesser extent; it may be possible to read a tape as a block special device, but write it only sequentially. Filesystems must be written on block special devices before they can be used directly by Idris.

A character special device is typically a "tty" or terminal (terminal port, actually). If so, it has various magical properties such as the ability to control process execution (DEL key and ctl-\, for instance), and facilities for editing input and throttling output. Other character special devices are things like line printers, serial tape drives, or bizarre hardware too idiosyncratic to be made to look like a block special device or a tty. Often the same device will have both block special and character special interfaces, for a variety of esoteric control reasons.

There are also a handful of character special files supported directly by the Idris resident. These do things like: make available to ps, the "process status" utility, information on the programs that are running; provide a place for discarding arbitrary amounts of output (/dev/null); and permit the System Administrator to peer inside the resident while it is running, on the rash assumption that this is a good idea.

You can spot a special file in a long form ls listing, because the first character of the access permissions is neither a ´-´ for a plain file, nor a ´d´ for a directory; instead it is a ´b´ for block special, or a ´c´ for character special. Only the System Administrator has the power to make special files; and generally they are confined to the directory /dev. Try

```
    % ls -l /dev
```

to see what sort of devices are on your system. Compare this with the listing you get from who to see what is done with ownership and permissions of the active tty files.

The best thing about special devices, as was stated at the outset of this essay, is that you really don´t have to know much about them to make meaningful use of Idris. Nor, for that matter, do you have to worry much about links, inodes, or filesystems.

You do have to mess with access permissions, from time to time; they offer considerable flexibility in setting up protected systems. And you should be familiar with directory hierarchies, pathnames, and relative navigation among nearby directories. The best way to learn is to look around, as this essay has encouraged you to do. Read the manual page for ls in Section II; it is a powerful probe.

You can look on the Idris filesystem as a sophisticated way to organize large quantities of information. Or you can take advantage of its ability to shield you from most of the complexity needed to support a multi-user system. Take your pick.

NAME

Advanced - processing and programming with the shell

FUNCTION

By now, you should know your way around the editor, and around an Idris filesystem. And you´ve seen enough of the shell to use it more or less like the "command interpreter" or "executive" of many an operating system. Here, you´ll discover how much more (and how much less) the shell really is. It is, first of all, a mechanism for precisely manipulating instances of other programs (called processes); but given a few interface conventions and several builtin facilities of its own, the shell can also serve as a programming language of surprising power and utility.

PROCESSES

A "process" is simply a running instance of a program. Whenever you type the name of a utility to the shell, it starts a process to execute the utility. The process runs until terminated, usually by the exit of the utility. Typing a pipeline to the shell results in the startup of one process for each program in the pipe; the processes are run simultaneously, with their input and output redirected as needed.

A given process may start and control an arbitrary number of other processes, called its "children", which may in turn have children of their own, to an extent limited only by the internal table space of the operating system. Each user in an Idris environment thus has his own tree of processes, at the top of which is usually a process running the shell (though any program at all could be run there).

In all the command lines examined so far, the shell started up a process for a single utility, or a set of processes for a pipeline, waited for the processes to terminate, then prompted you for another command line. You can also make the shell execute a sequence of commands before prompting again, by separating the commands with a ´;´, and giving them on the same line of input. For instance, to change to a directory and immediately examine its contents, you might type:

```
    % cd /final/doc; ls -l
```

In response, the shell would change to the directory, then execute the ls utility, and would prompt again only when ls terminated.

The shell can also be made to prompt immediately after starting up a process, instead of waiting around for it to terminate. (Try it with a long ls.) In this way, long or tedious tasks can be run "in the background" simultaneously with whatever you´re doing at your terminal, freeing you (and the terminal) for other things. Processes running in the background still can type to your terminal, so interactive programs should usually be run this way only if their STDIN and STDOUT are redirected. (Otherwise, they will get an empty file for input, and will send their output to your terminal -- a confusing situation.) The following would run e in the background, taking commands from the file fixup, editing the file needsit, and sending its output to the file trace:

```
    % e needsit <fixup >trace &
```

When you initiate a command with ´&´, the shell outputs a number called the processid, which uniquely identifies the process just started to execute the command. If the command involved more than one process, then a separate processid is output for each process started. Also, ´&´, like ´;´, can be used as a command separator, which causes the shell to start the left-hand command, then immediately start the right-hand command.

In order to kill a background process, rather than wait for its natural death, you can give its processid to the kill utility. The following would terminate the process with processid 317:

```
    % kill 317
```

In addition, should you forget a processid, or just want to see what´s going on, you can use the utility ps (for "process status") to display all of the processes associated with your terminal:

```
    % ps
    1577 run  ps
    1576 run  sh
    1544 run  pr
    1520 wait List
     331 wait sh
```

In this case, five processes are present. Listed for each one are its processid, its execution status, and the name of the program being run.

Finally, the wait command may be used to wait for all background processes to terminate before continuing. wait returns only when all processing is complete:

```
    % wait
```

CONNECTIVES

Both ´;´ and ´&´ cause a set of commands to be executed unconditionally. Clearly, however, it would sometimes be quite useful to execute commands based on the result of some previous command. Any program under Idris returns one of two values, called success or failure, when it terminates. You can execute a command based on the return value of the last previous command by separating the two with "&&" or "||".

A "&&" causes its right-hand command to be executed only if the left-hand command returned success, while "||" executes its right-hand command only if the left-hand one returned failure. For instance, the utility cmp compares pairs of files, and returns success if each pair is identical. The following would delete a personal copy of the file manpage only if it had already been copied into a second directory:

```
    % cmp manpage /final/doc/manpage && rm manpage
```

while the following would update a personal copy from the second directory if the file had changed there:

```
    % cmp manpage /final/doc/manpage || cp /final/doc/manpage manpage
```

(There are shorter ways of writing these calls on cmp and cp -- see their manual pages in Section II.) These "logical connectives" don´t cause any data to be passed between the commands they connect; they are used exclusively by the shell to determine whether or not to execute the right-hand command.

Any series of commands may be grouped together by enclosing them in braces "{}". The series will then be treated as if it were a single, simple command. Thus to record some lines in a file before running an editor script, you could type:

```
    % {grep string file1 >before; e <script >trace file1} &
```

Note that if you entered these two commands without grouping them, then grep would be executed in the "foreground" (i.e., while you waited), then e would be run in the background, because ´&´, as a separator, affects only the command immediately preceding it.

This point brings up the more general question of how the shell connectives we´ve looked at can be used together. They group together according to the following hierarchy:

1) a simple command (i.e., a command name and a series of ordinary arguments) may contain a single STDIN redirection (signalled by ´<´ or "<<"), a single STDOUT redirection (signalled by ´>´ or ">>"), or both.

2) a simple command may also be used as part of a pipeline, separated by ´|´ from the other commands in the pipe; however, because a pipeline involves redirection, a pipeline "source" (the left-hand command) cannot have its STDOUT already redirected, while a pipeline "sink" (the right-hand command) cannot have its STDIN already redirected.

3) any two of the above groups (simple commands or pipelines) may be connected by "&&", which causes the right-hand group to be executed only if the left-hand one succeeds.

4) any two of the above groups (including groups connected by "&&") may be connected by "||", which causes the right-hand group to be executed only if the left-hand one fails.

5) any two of the above groups (including groups connected by "||") may be separated by ´;´, which causes the left-hand group, then the right-hand group, to be executed in sequence.

6) any two of the above groups (including groups separated by ´;´) may be separated by ´&´, which causes the left-hand group to be started, then the right-hand group to be started, without waiting for the first group to terminate.

7) braces "{}" may be used to enclose any other construct, and cause it to be treated as if it were a simple command (i.e., as a single unit), thus overriding the hierarchy as parentheses would in an arithmetic expression.

All of the operators listed here obviously have special meaning to the shell, and are not normally passed to a command as arguments. If you want to strip them (or any other characters) of special meaning, you can do so in two ways. Any string of characters can be enclosed in quotes, causing each character inside to be passed literally to the command. Or, any single character (outside a quoted string) may be passed literally by preceding it with a backslash `\`. The quotes (or backslash) themselves are removed by the shell, and are not passed to the command.

PREFIX COMMANDS

Several utilities exist that take a command as an argument, and execute it after changing some condition under which it will run. One of these "prefix" utilities, called error, enables you to redirect the error output for a command, either to STDOUT or to another file. Normally, such error output is written to a standard file called STDERR (a counterpart to STDIN and STDOUT which gets nearly all error messages); STDERR, however, cannot be redirected straight from the command line. So to redirect it you would type something like:

```
% error -o trace grep string file1 file2 file3
```

which would run the grep command shown, after redirecting its STDERR to the file trace, to be created before grep is run.

Two other prefix utilities, called nice and nohup, enable you to run a command at a different priority than usual (be nice to others), or immune from certain interrupts (don't die if dataphone hangs up). A third, named time, records the amount of time consumed by its argument command, and outputs the totals to STDOUT after the command terminates. Because of the way the shell parses a compound command, a prefix utility can accept only a simple command as an argument; however, any command can be treated as a simple one just by enclosing it in braces. Thus to time the execution of a pipeline, you could type:

```
% time {sort data | uniq -c >output}
```

time, in turn, would output the amount of real time consumed by the pipeline, and the time it spent executing user code and system calls. (The last two figures together represent the usual "execution time".) Note that, without the braces, time would be timing sort alone, and the entire command line would be interpreted as a pipe, with the output of time piped into uniq.

FILENAME SHORTHAND

In addition to the process control operators presented so far, the shell provides several varieties of shorthand to ease the use of complex or frequently invoked commands. First of all, the shell permits whole groups of files to be named as a single argument, by using each argument containing any of the characters "*?^[" as a regular expression to be matched against the names of the files in the current directory. (A regular expression is a shorthand for naming a set of strings; see the editor essay earlier in this Section for a full description.) Each such argument is stripped of any directory pathname (i.e., all the characters up to and including the rightmost slash), then compared against the set of filenames. If any filenames completely match the remaining pattern, they replace it in the command line. Otherwise, it is left untouched. For instance, the command

```
% echo *
```

would output the names of all the files in the current directory, and

```
% cmp file[a-g] /final/doc
```

would compare any files with the names filea, fileb, ..., fileg to a set of files in /final/doc with the same names.

Since the shell bestows this expansion on you quite unbidden, it also permits you to disable it, by quoting any argument you don´t want expanded. Thus

```
% echo "*"
```

would pass to echo a single argument, consisting of a literal asterisk.

SHELL VARIABLES

A more flexible form of shorthand is provided by "shell variables", each of which can be assigned a string as its value, then used in place of the string. Shell variables have one-character names drawn from the set [0-9a-zA-Z$]. You refer to a shell variable by giving its name, preceded by a ´$´. Wherever a reference, like "$a", occurs, the shell substitutes the current value of the variable. The text of the value is then scanned by the shell exactly as if it had been directly specified in the command line. This means that, if the value contains special characters (including further variable references) they will be fully interpreted.

Variables may be assigned values by using the command set, which takes two arguments: the name of the variable you´re assigning to, and its new value. Suppose, for example, you were frequently accessing a directory other than one you were working in. Rather than constantly naming the second directory, you could assign its name to a shell variable, and then use the variable in place of the name:

```
% set p /final/doc
```

Once this set command had been given,

```
    % cmp manpg1 $p
```

would be exactly equivalent to

```
    % cmp manpg1 /final/doc
```

The new value must be quoted if it is to contain whitespace, since set takes only its second argument, not the rest of the command line, to be the value.

Four shell variables of the possible 63 are used by the shell to specify parameters, and so have special meaning. The variable P specifies the prompt that an interactive shell issues before accepting a command; H names your "home directory" (i.e., the directory changed to by a cd with no arguments); X specifies your "execution path"; and $ contains the processid of the process running the shell. To change any of these parameters, just use the set command on the appropriate variable. The following would suffice to alter your prompt:

```
    % set P A:
```

An execution path names the set of directories that the shell will search for files with names matching the command names you give. Utilities like cmp or pr are merely executable files stored in a standard directory that is normally included in your execution path. By default, you may be given the path, say,

```
    |/bin/|/etc/bin/
```

which causes the shell to look for commands first in the current directory, then in the directory /bin, and finally in the directory /etc/bin. Each string between ´|´ is a pathname prefix that the shell prepends to the command name you give. It then tries to execute a file with the resulting name. If it can´t do so, it tries the next pathname; if no pathnames are left, you get the "not found" message you´ve probably already seen.

An execution path is most commonly modified to cause the shell to search some private command directory when it looks for commands. That way, you can create your own commands, put them in a single place, and then use them from anywhere at all. For example,

```
    % set X "|/usr/myid/bin/|/bin/|/etc/bin/"
```

would add /usr/myid/bin to your execution path, where it would be searched before the system-wide binary directories. You could then name any executable file stored there, and the shell would automatically find it. The execution path string given must always be quoted, since otherwise the shell would interpret each ´|´ as a pipeline delimiter. A shorter way of giving the same path would be:

```
    % set X ´|/usr/myid/bin/|$X´
```

Inside single quotes, but not double quotes, references to shell variables are expanded, but the expanded text is inserted into the command line with no interpretation, except for the expansion of any further variable references it contains. Here, the old contents of X would be concatenated with the new pathname to make a single string, which would then become the new value of X. Except for this limited expansion of variable references, single quotes have the same effect as double: they suppress the ordinary interpretation of whatever characters they enclose.

SCRIPTS

A previous essay noted in passing that the shell is nothing more nor less than a standard utility named sh. You haven´t seen it used that way, because sh is automatically run for you when you login to a session, in what is called interactive mode. In this mode, the shell prompts for commands from STDIN, and ignores interrupts (so that you can use DEL to stop shell commands, and not the shell itself). But since it is an ordinary program, sh can also be run directly. If you type:

```
    % sh -i
```

you´ll start up a new interactive shell underneath your current one that will act just like its parent. (So of course typing ctl-D will terminate it.)

More interestingly, the shell will also read commands from a file other than STDIN, specified on its command line. This permits you to place shell commands in an ordinary text file (called a "shell script"), then pass the filename to sh as an argument. Thus potentially complex operations that you need to perform frequently can be invoked with a single, short command line. For instance,

```
    % sh cleanup
```

would start up a shell to read the commands stored in cleanup, which might contain the actions you normally take before logging off.

And scripts can be invoked directly by name, just as if they were binary programs, by giving them execute permission. Through a bit of chicanery when it tries to run a command, the shell will interpret as a script any executable file that is not a legitimate binary program, and will start up a second shell to read the script. So, if you typed:

```
    % chmod cleanup
```

in order to make cleanup executable, you could then run it exactly like a binary program. And, if you stored it in some directory on your execution path, you could run it from anywhere just by typing its name:

```
    % cleanup
```

Though simple scripts like this one can be a valuable form of shorthand, shell variables greatly extend their potential, by making it possible to pass arguments to a script like the ones passed to a binary program. Whenever you run a script, the variable 0 is set to the name of the script, and variables 1 through 9 are set to the values of its first 9 arguments. If fewer than nine arguments are given, the extra variables are assigned an empty string as their value; and outside a script, the whole group contains empty strings.

If we wanted to simplify the usage of the glossary-maker in the first essay, we could put this line into a shell script named gloss, and then invoke the script instead of typing the command line:

```
    tr A-Z -t a-z $1 | tr !a-z -t "\n" | sort | uniq
```

When the script is run, the reference "$1" is replaced by the value of the first argument to the script. So to run gloss on a file, all you´d need to do is name the file as the first argument. Presuming gloss had been made executable, the following would produce a glossary of trial, just like the last example of the first essay:

```
    % gloss trial
```

In addition to the numeric variables, two special variables, @ and *, are provided, which have as their value <u>all</u> of the arguments passed to a given script. The reference "$@" is replaced by the literal text of the arguments, each one treated as a separate string; "$*" is replaced by the same text, but quoted so as to be treated as a single string, consisting of the arguments separated by spaces. Thus, to make the script gloss produce a combined glossary from any number of input files, you could change it to be the following line:

```
    tr A-Z -t a-z $@ | tr !a-z -t "\n" | sort | uniq
```

which would pass to tr all of the arguments given to the script. Outside a shell script, @ and * always have the empty string as their value; never can they be assigned new ones.

This ability to access command line arguments from within a script is central to still another form of shorthand, the prefix command exec. exec takes as an argument a command that the current process will start running <u>instead</u> of the shell. The shell you had been running is lost forever. From an interactive shell, exec is not generally useful, and is even somewhat dangerous, since the shell does not return when the specified command finishes execution. (Instead, you´ll usually get logged off.)

From within a script, however, exec is a fast way to invoke a single (perhaps complicated) command line. It´s fast because, since the shell running the script is destroyed when exec is run, it doesn´t hang around for the command line to finish. If, for instance, you frequently used pr to generate tabular data, you could save yourself time and potential errors by putting the special pr command you need into a script, say maketab, then using the script whenever you wanted to generate data:

```
    exec pr -t -m -s: $@
```

(See the manual page for pr in Section II for the meaning of all these neat flags.) So long as maketab were executable, you could form a table from three files just by typing:

```
    % maketab file1 file2 file3
```

since any filenames you give to maketab are automatically passed to pr.

The special variable @ makes it very easy to pass an entire argument list to a command inside a script. But suppose we wanted to have the same external interface (i.e., the ability to accept an arbitrary number of arguments), and also wanted to process each argument separately? To do both, you might make the script contain a loop, each iteration of which would process a single argument file. For instance, the following would selectively remove each file passed to it as an argument, based on whether or not it also existed in a second directory:

```
    : loop
        echo $1:
        cmp $1 /final/doc && rm $1
        shift && goto loop
```

The first line is a label that will be used by the later goto command, while the next line outputs to STDOUT (presumably a terminal) the name of each file, followed by a colon, as it is processed. The last line does two things. First, shift is used to "shift" the arguments to the script. That is, shell variable 1 is made to point to the previous value of variable 2, 2 is made to point to the value of 3, and so on. In this manner, the filenames on the command line are shifted left each time shift is run, and each one in turn is associated with variable 1. shift succeeds only if variable 1 was assigned a non-empty string, i.e., if there was another filename to be shifted into it. So long as there was, goto is run, which scans the script from the beginning for a "label command": whose first argument matches the first argument to goto; if such a label command exists, then the execution of the script resumes with the line following it. When shift returns failure, goto is not run, and the script terminates.

It should be emphasized that, once any script is made executable, it can be used in a command line exactly like a binary program. So if the last script were stored in an executable file called remove, a command line like this one could be used to remove a series of files:

```
    % remove file1 file2 file3
```

Or to redirect the output of maketab to the file table, you could type:

```
    % maketab column1 column2 column3 > table
```

while the following would suffice to pipe the output of gloss into pr:

```
    % gloss file1 file2 | pr
```

SHELL PROGRAMMING

Scripts like remove, which execute some set of commands on each of a list of files, are examples of the programming language that the shell defines, a language of which scripts can be viewed as the procedures or even the subroutines (because, of course, scripts can be named as commands inside other scripts). The shell language, however, has further capabilities. Scripts are capable of full interaction with a terminal, and can perform conditional branching based on the responses they receive. The following lines might appear at the start of a (rather simple-minded) script managing the access to a file of data records:

```
    : input
        echo -m "enter next command --" "(newline to quit)"
        set c -i
        test $c || exit
        test $c == a && goto add
        test $c == d && goto delete
        test $c == h && goto help
        echo "don´t recognize response -- type h for help"
        goto input
```

Each time it was executed, this excerpt would prompt STDIN for a single line of input, which it would try to interpret as a single-letter command. The opening echo command -- banal enough if typed in from your keyboard -- here serves to output the two strings shown to the terminal. The set command specifies a flag, -i, that causes the value for the variable c to be read as a single line of STDIN, instead of being taken from the set command line.

What follows is a series of checks of the value that c has just been assigned. The utility test compares two strings for equality (s1 == s2), or one string for being non-null, and succeeds only if the relationship tested is true. The first test command checks whether c has a non-null value. If not (i.e., if test fails), then the user typed an empty line, and exit is run -- which, as its name implies, causes the script to exit. The subsequent test commands all check whether or not c is equal to a specific command letter. (The "==" is what checks for equality; "!=" could be used instead to check for inequality.) If one of the command letters does match, then test succeeds, and the corresponding goto is executed. Otherwise, the script falls through to the following echo, which outputs an error message and goes back to get another command line.

The exit command can also be used to control the return value of a script, or more precisely, of the shell reading the script (which returns success or failure just like any other program). If an exit is executed with no arguments, the shell returns success; if the exit command contains an argument, the shell returns failure. Or, if the shell terminates by reading off the end of a script (and not with an explicit exit), it returns the return value of the last command it executed. Hence, this line could be inserted at the start of the script remove, to force it to fail when not given at least one argument:

```
    test $1 || echo "usage: $0 <files>" && exit NO
```

The argument "NO" to exit causes the shell to exit reporting failure (any non-null argument would do). Note the use of $0 to pick up the name by which the shell script was invoked, for printing out the error message.

Another versatile command is set -i, which, since it reads one line of STDIN, can easily be used as a sink in a pipeline. A frequent usage is:

```
    pwd | set d -i
```

which permits a script to record what directory it´s running in, so that the directory can be restored later.

One additional feature of scripts is that they may contain embedded instances of interactive commands. If a command contains the redirection symbol "<<", then the current source of shell command input is read, up to a line containing only the string following the "<<". All of the lines read are put into a temporary file that is then made to be the STDIN for the command. For instance, if shell variable w already contained the name of the user running a script, then the run could be recorded at the start of a log file with the commands:

```
    date | set t -i
    cat > /tmp/logent << END

    run by $w at $t:
    END
    cat /final/doc/log >> /tmp/logent
    mv /tmp/logent /final/doc/log
```

The first line, of course, gets the output of date into the variable t. The cat command following reads its STDIN (because it has no input files), and writes the lines read to the file /tmp/logent. Because of the "<<", the STDIN for cat is created from the next lines in the script, which is read until END is encountered on a line by itself. (The terminating string can be anything at all, so long as it matches the one given after the "<<".) The final two lines add the new entry to the start of the log file, by appending the old log to the new entry, then naming the result so as to replace the old log.

As shown above, shell variable references are expanded in the STDIN created by "<<", though by simpler rules than in a shell command line. In a "<<" STDIN, any sequence "$x", where x is a valid variable name, will be replaced with the contents of x, as for an unquoted command line reference. However, if x is not a legal name, then the two characters are left untouched. Any ´$´ may be treated as literal by preceding it with a backslash -- an important thing to remember when creating command input for e, to which ´$´ has special meaning.

Since all of these last features are part of the shell, they can also be used from your terminal in interactive mode, though with varying utility.

A few, like labels and the goto command, make sense only in scripts. Others, like "<<", can be quite useful when used interactively. Consider the following line, which would enable you to enter a script of editing commands from the keyboard, then execute the editor in the background, working on file1:

```
    % e << END >trace file1 &
```

When you entered this command, the shell would read lines from STDIN until you typed END on a line by itself; only then would it start up a background process to run e, which would read the commands you just entered.

All of which provides a final example of a more general fact: the shell is designed to encourage experimentation, and the innovative use of its facilities. It provides a versatile set of parts out of which "programs" (perhaps as simple as a pipeline) may be quickly and easily built to solve a broad range of data processing problems. The goal of this essay has been to introduce those parts, and give some hints of how they can be used. But this has been only an introduction -- the power of the shell and of the utilities accompanying it lies in the many, flexible, and often surprising ways in which they can be combined, which no essay could ever adequately cover. So experiment.

NAME

Glossary - a lexicon of Idris terms

FUNCTION

The terms defined here are either jargon specific to the world of data processing under Idris, or terms with a slightly more specialized meaning in this context than may at first be apparent:

argument - a whitespace delimited string on a command line, frequently a filename, conveyed to a program to modulate its behavior.

ASCII - acronym for American Standards Code for Information Interchange, the method of representing characters as small numbers, in the range [0, 128), used by Idris.

backspace - ASCII BS code, which causes the cursor (or print head) to move backward one character position.

binary - a positional numbering system with two digits [0-1], used to perform all arithmetic, at the lowest level, in a computer. Also, containing byte values other than the ASCII character codes, hence in opposition to text. Also, used to describe programs directly executable by the computer, as opposed to shell scripts.

bit - a binary digit, hence a number with just two values (0/1, false/true, off/on). Usually grouped in multiples of eight to represent positional binary numbers. All computer storage and computation is, at the lowest level, in terms of bits.

block - a contiguous group of 512 bytes, the basic unit of space allocation in filesystems.

bootstrap - the process or program that gets a computer going from a standing start.

buffer - an area of temporary storage, used to stage data before it is copied to its final destination.

byte - a group of eight binary digits, hence a number with 256 possible values. The basic unit of data storage and transmission, since it can comfortably represent all the ASCII character codes, one per byte.

character - a single keystroke, usually stored as ASCII in a single byte, typically representing a letter, digit, punctuation, or other printable text.

child - the younger member in a hierarchical relationship, as among directories or processes.

close - to break the connection between an opened file and a program.

command - a line of text interpreted by an interactive program, such as the shell or the editor, that tells it what to do next.

completion - (pathname), a method for determining the name of a file, given its parent directory and a source file name.

copy - to duplicate, as when copying a file or other data stream to a new place.

core - common name for a file used to record the running status and memory image of a program that was forced to terminate abruptly.

create - to make a new instance of, as when creating a file.

ctl- - ASCII FS code, typed at a terminal keyboard on a controlling tty to send a quit signal, which causes most programs to terminate abruptly, with a core dump.

ctl-D - ASCII EOT code, typed at a terminal keyboard to simulate end-of-file condition.

ctl-Q - ASCII DC1 code, typed at a terminal keyboard to force printout to be resumed.

ctl-S - ASCII DC3 code, typed at a terminal keyboard to force printout to be suspended.

decimal - a positional numbering system with ten digits [0-9], used to count all the fingers on your hands.

decrypt - to translate encrypted data back to a recognizable form, by using a secret password.

default - a value assumed in the absence of an explicit input value.

destination - the place where data is to end up.

device - a piece of machinery or electronics connected to a computer for the purpose of entering or extracting data.

diagnostic - an error message output by a program to inform the user of some problem.

directory - a file whose contents are the names of other files.

editor - the standard utility providing commands to create and modify text files.

encrypt - to translate data to a form that is neither recognizable nor easily translated back to recognizable form in the absence of a secret password.

executable - having access permission, and the proper format, to be run as a program.

exit - to terminate program execution.

failure - status returned on program exit, indicating something went wrong. Testable by the shell.

file - a contiguous collection of bytes stored within a filesystem, known by one or more names or links.

filename - the name by which a file is referenced, synonymous with pathname under Idris.

filesystem - a logical data structure imposed on a contiguous collection of disk blocks, used to represent files and directories under Idris.

filter - a program that consumes an input stream from STDIN and produces and output stream to STDOUT. Often the streams are text.

flag - a shell command argument beginning with a '-' or '+', used to pass parameters to programs.

footer - a line of text appearing at the bottom of a formatted page.

free - available for reuse, as a block of a filesystem.

groupid - a number in the range [0, 256) used to encode group membership for access permission checking.

hexadecimal - a positional numbering system with sixteen digits [0-9a-f], used as a shorthand for groups of four bits.

I/O - input/output, generic term for data transmission between a program and its environment.

Idris - a Persian god, the patron deity of craftsmen, credited with the invention of any number of tools and crafts, including the art of sewing things together.

inode - a filesystem entity that contains all of the attributes of a file, except its contents and name(s).

interactive - a mode of program operation in which a person can type a command, then see the result before deciding on the next command.

interrupt - a signal, generated by striking the DEL key on a controlling tty, that causes most programs to terminate abruptly.

invoke - to cause a program to begin execution.

kill - a signal, generated by the kill utility, that causes any program to terminate abruptly, with a core dump.

link - a directory entry that points to a file, one component of a pathname.

loginid - a string of one to eight characters used to identify a user to the system at login time, and to associate a password, userid, groupid, and other operating information with a given user.

match - to meet the requirements of a regular expression, as for a substring within a text line or for a filename in a directory.

mode - of a file, the access permissions associated with the file.

mount - to make a filesystem available through the directory hierarchy.

move - to change the location of, as when renaming a file.

newline - ASCII linefeed (LF) code, used in all Idris text streams and text files to terminate each line.

octal - a positional numbering system with eight digits [0-7], used as a shorthand for groups of three bits.

open - to make a file available to a program for input and/or output.

parameter - a value used to modulate the behavior of a program, often passed as an argument.

parent - the older member in a hierarchical relationship, as among directories or processes.

password - a short text string used to encrypt and decrypt confidential data, also used to authenticate a loginid by encrypting a known constant. Preferably kept imaginative and secret.

pathname - a sequence of slashes and links that specifies the name of a file, relative to either the current or root directory.

peripheral - synonym for device, often used redundantly as in "peripheral devices".

permission - for a file, the license to access it for reading, writing, or executing. For a directory, the license to access it for reading, altering, or scanning.

pipeline - a data stream connection between two simultaneously executing programs that permits one to read what the other is writing.

prefix - (command), a command which runs other commands, possibly first modifying the execution environment.

process - the execution of a program, which in Idris is controlled by a resident data structure often also called a process.

program - a file that can be directly executed by the computer, with the assistance of the Idris resident.

prompt - a string of text written to a terminal to signal that an interactive program is ready for a command line to be input.

recursive - a self-referential rule for performing an operation, as when climbing over a directory tree to visit all subdirectories.

redirection - altering the source for input to a program, or the destination for its output, to a file specified on a command line to the shell.

regular - (expression), a notation for representing text patterns, used for matching.

resident - that portion of the Idris system which resides in main computer memory at all times, to assist executing programs and to administer peripheral devices and other shared resources.

return - (carriage), to bring the cursor (or print head) back to the left margin, as with an ASCII carriage return (CR) code. For a program, to deliver up a success or failure indication upon completion.

root - the fundamental directory of any filesystem, in particular, the fundamental directory of the filesystem specified at system startup. Also the conventional loginid of the superuser.

script - a file containing a sequence of commands for a program that is normally interactive, such as the shell or the editor.

shell - the standard utility used to interpret commands to run other programs.

source - the place from which data is to originate.

space - the ASCII character generated by pressing the space bar on a keyboard, which causes a one-character wide space to be left when displayed.

special - (file), a file that causes a peripheral device to twitch when accessed. There are character special files, such as ttys, and block special files, such as disks.

startup - (system), the time right after bootstrap and before normal operation, when the Idris system initializes everything in sight.

STDERR - standard error text stream, made available to each utility run by the shell.

STDIN - standard input text stream, made available to each utility run by the shell.

STDOUT - standard output text stream, made available to each utility run by the shell.

stream - an ordered sequence of data that passes through a program once, in sequence.

success - status returned on program exit, indicating nothing went wrong. Testable by the shell.

superuser - userid number zero, usually a persona of the System Administrator, having the power to circumvent most access restrictions and to perform restricted operations.

tab - ASCII HT or VT code, which advances the display to the next tab stop. Horizontal tabs (HT) are assumed every four columns under Idris, but can be passed on to the terminal to expand. Vertical tabs (VT) are always passed on.

text - a stream of printable characters, usually intelligible to people.

title - a line of text appearing at the top of a formatted page.

tty - a character special file that supports an interactive terminal.

unmount - to disconnect a mounted filesystem from the directory hierarchy.

userid - a number in the range [0, 256) used to encode the identity of a user for access permission checking.

utility - a program of general usefulness provided by the Idris system.

whitespace - a generic term for spaces, tabs, newlines, and other ASCII characters that cause printer motion but produce no graphics.

word - in text processing, a contiguous string of letters and punctuation, delimited by whitespace. In some circles, a group of bits that is a natural size, in some sense, for a given computer.