An Introductory User's Guide to IRAF Scripts

Ed Anderson
Revised by Rob Seaman

Central Computer Services
National Optical Astronomy Observatories††

IRAF scripts are used to repeatedly perform a set of manipulations on any number of images or, in general, to control the execution of any related set of tasks. Scripts can be either simple or complex. This document will aid you in constructing a variety of IRAF scripts, along the way, many "gotchas" and other subtleties will be discussed.

You can get help from the following people:

Rob Seaman - (seaman@noao.edu)
Jeannette Barnes - rm 86, x381

or from the IRAF hotline; see the latest IRAF Newsletter for details.

††Operated by the Association of Universities for Research in Astronomy, Inc. under cooperative agreement with the National Science Foundation.

September 22, 1989

Table of Contents

1. Introduction

An IRAF *script*† is a tool that allows you to put together any number of IRAF Command Language (CL) statements to be executed as a group, in much the same way as a UNIX shell-script or a VMS command procedure. Like the shell-script or command procedure, simple programming commands are provided so that you may make the script (if desired) into a complex program. Indeed, many frequently used IRAF tasks are scripts, for example: MKSCRIPT, AVERAGE, DARKSUB, ROTATE, REGISTER, IMLINTRAN, and ECBPLOT. This document will aid you in developing ways of *stringing together* IRAF commands.

This document is NOT for the beginning IRAF user; it assumes that you know how to use an editor, know how to use **eparam**, know the difference between *hidden* and *queried* parameters, and know the ins and outs of I/O redirection. *A User's Introduction to the IRAF Command Language* is a good place to start if you need to reacquaint yourself with some of these concepts.

While trying the examples, you can read about individual CL components such as **if**, **else**, and **while** in the IRAF help pages under the LANGUAGE package. To see an overview of the language elements and builtin functions, with a one-line description for each, you can use the command `help language`. The help page for each language element or function is available as a detailed description. There are also help pages for more general topics, such as **parameters**, **cursors**, **mathfcns** and **strings**; to see the help page for the CL itself (including a description of its parameters), type `help language.cl`.‡

Much of the material in this document is also covered (briefly) in *A User's Introduction to the IRAF Command Language*. The *CL Programmer's Manual* and the *Detailed Specifications of the IRAF Command Language Version 1.2* provide an out of date, but still interesting overview of the CL as it was being developed.

For ease of understanding, we have broken script writing into six levels of complexity. Each section assumes knowledge of the concepts and tools used in the previous sections but is independent of following sections. There are tested examples at each level of script writing, although there may be other ways to achieve each result. The examples provide a starting point for you to experiment with variants and alternative approaches. As with any guide to computer usage, the best way to learn is by trying the examples and experimenting with the ideas introduced.

Before starting, you must be familiar with the two different modes of the CL. **Command mode** is the default mode, and is used for most terminal interactions. It is convenient for interactive use because it minimizes the need for command delimiters, quoted character strings, and the special handling of file names and meta-characters. **Program mode** (also called **compute mode**), on the other hand, requires the *full syntax* of the CL as a programming language, but allows the use of variables and control constructs. **Program mode** is:

- Entered within the body of a procedure (see §7).
- Entered within parenthesized expressions.
- Entered on the right-hand side of an equal sign (=).

These two modes can be mixed within simple scripts; this will become apparent in the next few pages.

†While the Command Language works well for writing scripts, it has not yet been fully developed for this purpose, and has some quirks and bugs that you may have to work around. You should be aware that the current CL is a *functional prototype*, and is scheduled to be completely rewritten in the future.
‡See the footnote on page 17.

2. MKSCRIPT

MKSCRIPT is the simplest way to produce an IRAF script. MKSCRIPT is a task in the SYSTEM package that allows you to create a script, without using an editor and without knowing about syntax, containing any number of commands to be executed sequentially.

To use MKSCRIPT, type `mkscript`. You will be asked for the name of the new script file (*e.g., reid.cl*) and then for the name of a task to be included. MKSCRIPT enters **eparam** for each task, where you should edit all the parameters for the task, including the *query* parameters. After you exit the parameter editor, MKSCRIPT formats the command and places it in the script file (you are given the chance to verify that the command is correct). You can then add more tasks. When you have placed all the commands in the script file, MKSCRIPT will show you the script (using the PAGE command) and will ask whether it is correct. If the script **is not** correct, MKSCRIPT will erase it and start again; this can be painful if you have entered many commands. Sometimes it is better to tell MKSCRIPT that the script is okay and edit it yourself to make minor corrections. If the script **is** correct, MKSCRIPT will ask whether the file is to be submitted as a background job. If you answer no, you can submit the script for execution yourself (after making any corrections) with the following command:†

```
cl> cl < scriptname.cl &
```

The `&` indicates that the job is to run in the background, or `&queue_name` that the job is to run in a VMS batch queue. By convention, script filenames end in `.cl`.

MKSCRIPT is useful when you are doing *production mode* data reduction. In this type of work, you are principally interested in performing the same task on several different images, with perhaps only minor changes to the task parameters; this is why MKSCRIPT is mentioned in some of the data reduction *cookbooks*. The following was created by MKSCRIPT to REIDENTIFY arc spectra in a longslit data set.‡ The full syntax shown in this example is not necessary for all levels of script writing.

```
reidentify ("nite1003", "@compfiles", section="middle column",
shift=0., step=10, nsum=10, cradius=5., threshold=10., nlost=2,
refit=yes, database="nite1db", plotfile="", logfiles="reid1log",
verbose=no)

reidentify ("nite1004", "nite1004", section="middle line",
shift=0., step=20, nsum=10, cradius=5., threshold=10., nlost=0,
refit=yes, database="nite1db", plotfile="", logfiles="reid1log",
verbose=no)

reidentify ("nite1007[*,396]", "nite1007", section="",
shift=0., step=20, nsum=10, cradius=5., threshold=10., nlost=0,
refit=yes, database="nite1db", plotfile="", logfiles="reid1log",
verbose=no)
```

There are some limitations when using MKSCRIPT, and you are encouraged to read the help pages for this task.

†Note that the initial `cl>` is the prompt printed by the IRAF Command Language and should not be entered as part of the command.

‡These commands have been reformatted for this document.

3. Terminal Scripts

You will recall that several tasks may be called in sequence on a single command line, using a semicolon (`;`) to separate each command. For example:

```
cl> clear; cd database; dir
```

If the command sequence is too long to fit on a single line, one could construct a **compound statement** by using the curly braces, '`{}`':

```
cl> {
>>> clear
>>> cd database
>>> dir
>>> }
```

The '`>>>`' prompt indicates that the CL requires more input (in this case, the CL is waiting for a '`}`') before executing the task or sequence of tasks. A *Terminal Script*† is essentially a compound statement, but uses some of the simple programming tools provided by the CL.

Before moving on to examples of terminal scripts, we will describe some of the simpler command level programming tools. There are twelve predeclared variables in the CL: three character strings (*s1*, *s2*, *s3*), three booleans (*b1*, *b2*, *b3*), three integers (*i*, *j*, *k*) and three reals (*x*, *y*, *z*). There is also one text file (*list*) which can be read using the **fscan** command. One may define additional variables as described later. For now we will restrict ourselves to the predefined variables. Since the predefined variables are all *hidden parameters* of the CL, their current values can be seen by entering the command, `lparam cl`. One can also type `eparam cl` to set these parameters. Perhaps the most useful commands (since the CL has so many parameters) are `=x` to view the value or `x=value` to set the value of the variable, `x`.

We discussed the difference between the command and program modes of the CL in §1. The distinction is most important when using character strings, and hence the CL variables *s1*, *s2*, and *s3*. Unquoted identifiers are treated as character strings in command mode, but as variable names in program mode. For example, if the CL variable *s1 = "Hi there"*, four different print commands that use the identifier '`s1`' are:

```
cl> print s1              - command mode
s1

cl> print "s1"            - command mode
s1

cl> print (s1)            - program mode
Hi there

cl> print ("s1")          - program mode
s1
```

To avoid confusion when writing scripts with string variables and constants, always quote character strings and always parenthesize expressions.

†This term coined by the author.

A common use of terminal scripts is to execute tasks that do not accept image templates (*e.g.*, @-file lists of images). This applies (for various reasons) to some tasks in the PLOT, PROTO and LOCAL packages. For example, suppose you want to make contour plots of several CCD images to locate objects. CONTOUR works on only one image at a time, so the thing to do is to put the image names into a file, and then type in the terminal script:†

```
pl> sections data*.imh > contour.lis      (1)
pl> list = "contour.lis"                  (2)
pl> while (fscan (list, s1) != EOF) {     (3)
>>>     contour (s1, dev="stdplot")       (4)
>>> }                                     (5)
pl> gflush                                (6)
```

where

(1) Produces the text file, *contour.lis*, of image names to be contoured. The **files** and **sections** commands are useful for making file or image lists, and one can always edit the file to remove names that do not belong.

(2) Assigns the text file, *contour.lis*, to the file variable *list*. Please note that the quotes (either `""` or ´´)‡ are necessary.

(3) Uses the **fscan** statement to read *list*. This is an unformatted read, so the first non-blank entity encountered on a given line of *list* is assigned to the string variable, *s1*. Each execution of **fscan** reads a new line without regard for any unread data from the previous line.

Fscan is a function that returns the number of items read. A special code (`EOF`, upper case is important) is returned when the end-of-file is encountered.

The '`!=`' means *not equal to.*

Thus, the statement says *do what is enclosed in the braces, reading the string s1 from the list, until the end of the list is reached.*

(4) Does the contour plot, sending it to whatever hardcopy unit is associated with the system variable *stdplot*. You could also reference a specific device (*e.g.*, `dev=versatec`). The name of the image to be contoured is in the variable *s1*.

(5) Marks the end of the *while* loop. IRAF will only execute the script after you enter the brace.

Note: If you omit the opening brace (`{`), you may still be presented with the '`>>>`' prompt until a semicolon (`;`) is entered.

(6) Flushes the plot buffer to the appropriate device (in this case, *stdplot*). Remember, plots accumulate in a buffer before being sent to the hardcopy device at some system defined interval.

Statements *2*, *3*, and *4* are executed in **program mode**, while statements *1* and *6* are executed in command mode; thus, the file name *contour.lis* is quoted in statement *2* and the variable *s1* is parenthesized in statement *4*.

†The `pl>` prompt was chosen to indicate that this is primarily a **plot** package (CONTOUR task) example. Since the SECTIONS task is also used, the **images** package must also be loaded before the script is executed. These packages may be loaded in either order. They may be loaded from the command line, from the user's *loginuser.cl* file or from another package (*e.g.*, **noao**); the resulting prompt will vary depending on the order and the manner loaded. Note that the prompts for other examples were also chosen to reflect the package of primary interest.

‡Hint: to embed one type of quotes in a string, delimit with the other.

Another common use for terminal scripts occurs when you need to operate on the same image a few times, changing the task parameters each time. For example, the IMCNTR task in the PROTO package will only calculate the center of one object at a time. Assuming that you have determined gross pixel coordinates for several stars from one of the contour plots above, you would first edit a file (say *coord.lis*) which contains the coordinate pairs. Then you would type in a terminal script of the following form:

```
pr> list = "coord.lis"                          (1)
pr> while (fscan (list, x, y) != EOF) {         (2)
>>>     imcntr data001 x y >> data001.cen       (3)
>>> }                                           (4)
```

where

(1) Assigns the file *coord.lis* to the file variable *list*.

(2) Sets up a *while* loop based on whether **fscan** hits the end-of-file while reading two real numbers into the CL variables *x* and *y*. The script **does not check** to see if two numbers were actually read, or if instead only one number or no number (if that line of the file was blank) was read from the list. The return value of **fscan** can be checked if that level of error detection is required.

(3) Calls IMCNTR to operate on the image *data001*. The initial x and y coordinates are set to the current values of the variables *x* and *y*. The output (which is normally sent to the terminal, or more precisely to the STDOUT) is being redirected to the text file, *data001.cen*.

It is important to note that the *append* redirection is used in this case, as shown by the double arrow ('>>'). If one were to use only the single arrow ('>') type of redirection, one of two things would happen the second time through the loop:

a) If the system variable *clobber = no*, then the script would encounter an error and stop, since the file *data001.cen* would already exist.

b) If *clobber = yes*, then *data001.cen* would be deleted with each execution of IMCNTR. Only the last centered coordinates would be recorded.

Note that the append redirection will create the file if it does not already exist.

The *clobber* parameter is useful for managing temporary files within CL scripts. This is usually done in conjunction with the MKTEMP task, for creating the files, and the DELETE task, for finally disposing of the files, as later examples show.

This statement is interpreted in command mode. The string *data001* is exactly what we want (*i.e.*, **not** a variable); recall that command mode interprets identifiers as character strings. The CL is smart enough, however, to recognize that *x* and *y* are not numbers and so must be interpreted as variables.

(4) Marks the end of the *while* loop.

The preceding terminal script will only work on one image, *data001*. What if you need to operate on many images with the same input coordinate list? One solution is to use the *history* feature of IRAF to edit the terminal script. Unfortunately, the history editor, **ehistory** does not work well on compound statements;† all you will see if you edit this command is the closing curly brace. The following example shows another method of editing the *history*:‡

```
pr> list = "coord.lis"                                    (1)
pr> while (fscan (list, x, y) != EOF) {                   (2)
>>>     imcntr data001 x y >> data001.cen
>>> }
pr> type data001.cen                                      (3)
[data001] x: 230.46   y:  12.17
[data001] x: 187.50   y:  33.85
[data001] x: 161.59   y:  18.91
[data001] x: 115.55   y:  67.56
[data001] x:  63.88   y:  53.11
pr> ^list                                                 (4)
list = "coord.lis"
pr> ^while:p                                              (5)
while (fscan (list, x, y) != EOF) {
    imcntr data001 x y >> data001.cen
}
pr> ^001^002^g                                            (6)
while (fscan (list, x, y) != EOF) {
    imcntr data002 x y >> data002.cen
}
```

where

(1) Assigns the file *coord.lis* to the file variable *list*.

(2) Executes the terminal script on image *data001*.

(3) Examines the output file.

(4) Uses the caret (^) to recall the most recent **list** definition. Before you issue this command, the *list* file is open and sitting at the end-of-file. Re-issuing the **list** command closes the present list file and opens a new one at the beginning. In this case, the same file is opened (*i.e.*, rewound).

(5) Uses the caret-colon-p to recall the *while* loop to the most recent spot in the *history*. The colon-p indicates that the command is NOT to be executed, but is to be printed on the terminal instead.

(6) Globally replaces all instances of `001` with `002` in the most recent command and executes the command (in this case the most recent command is the terminal script). Note that the `^g` is extremely important. Without it, only the first `001` in the command would be replaced causing the centering output from *data002* to go into the file for *data001*.

†Editing of compound statements can be made to work with a little effort. First, clear the screen (`clear`) so the command block will not be superimposed on top of other text. Next, enter the history editor, for instance: `e while`. To see each line, you must *repaint* it: point to the line using the ↑ and ↓ keys and repaint it with `^R` (if your **editor** is *vi* or *edt*, in general this is the same command as used in **eparam**).

‡The caret (^) will only behave as described if the CL parameter *ehinit* includes `noverify` as an option. If `verify` is used, a caret command will execute **ehistory**.

In our final example of a terminal script, we consider the common problem faced by many IRAF users who are trying to reduce data that were **not** taken with KPNO or CTIO instrumentation; namely, changing the image keywords so that the IRAF reduction tasks can function properly (*e.g.*, the flux calibration procedures expect the exposure time in the header keyword *EXPTIME*, *ITIME*, or *EXPOSURE*). Assume, for the sake of example, that your data have the exposure time under the keyword *INT* (for integration time). To reduce the data, one must use the task **hedit** to add the keyword *EXPTIME* and set its value to that of *INT*. This can be done with a terminal script of the following form:

```
im> sections data*.imh > datalist                          (1)
im> list = "datalist"                                      (2)
im> while (fscan (list, s1) != EOF) {                      (3)
>>>     imgets (s1, "int")                                 (4)
>>>     hedit (s1, "exptime", imgets.value, add+, ver-)    (5)
>>> }                                                      (6)
```

where

(1) Makes a file with the names of the images whose headers need modification.

(2) Assigns the text file to the variable *list*.

(3) Sets up a *while* loop based on whether **fscan** hits the end-of-file while reading the image name.

(4) Uses the IMGETS task to get the value of the keyword *INT* for the image *s1*. The value is stored in the **imgets** parameter *value*.

(5) Uses HEDIT to create the new keyword, *EXPTIME*, and give it the value of the keyword, *INT*. Note how the IMGETS parameter, *value*, is directly referenced and that *verify = no* (*i.e.*, `ver-`) is used so that the user will not have to verify each HEDIT operation.

(6) Marks the end of the *while* loop.

NOTE: HEDIT is a powerful tool that can be used for complicated editing chores. The results from above could be achieved with the simple command:

```
hedit data*.imh exptime "(int)" add+ ver-
```

The value field is quoted to avoid interpretation of the parentheses as an argument list; instead, the '`()`' force the interpretation of '`int`' as the name of the header keyword, *INT*, that is as a header "variable". The value of *INT* will be read from each image header and will be assigned to a new keyword, *EXPTIME*.

4. CL Scripts

You will probably create some scripts that you will want to use repeatedly; you may even write a single large script. A *CL script* is a file containing the commands you wish to execute.† As with the terminal script, both **command** and **compute** mode may be used in this type of script. The file will be submitted to the CL for execution; if there is an error, one edits the file to correct it and then resubmits it to the CL for execution.

Consider the IMCNTR example of the previous section. If you do not like the idea of editing your *history* to get a large number of images done, you can do two other things (both of which could be terminal scripts if you trust your typing). Consider the following CL script (these commands have been put into a file called *center.cl* where the *.cl* extension indicates that the file is a CL script; the *.cl* is not necessary but is a highly recommended convention):

```
list = "coord.lis"                          (1)
while (fscan (list, x, y) != EOF) {         (2)
    imcntr (s1, x, y,>> s1//".cen")         (3)
}                                           (4)
```

- This is the same thing as the first IMCNTR example of the previous section except that statement (3) is different: it now uses the full syntax of the CL with commas, quotes, and parentheses. This is because it now uses a CL string variable (*s1*). If the command were written in **command mode** (*i.e.*, without full syntax) the CL would assume that *s1* is an image name instead of a variable, and would respond with an error saying that it could not find image *s1*.
- Note how the output text file name is constructed. The string variable *s1* is concatenated (`//`) with the string literal `".cen"`. The variable **is not** quoted, but the literal **is** quoted.
- Also note that the **I/O redirection occurs inside the parentheses**.

To use this script on any given image, you first assign the image name to the variable *s1* and then execute the script. For example:

```
pr> s1 = "data001"                  (1)
pr> cl < center.cl                  (2)
pr> s1 = "data002"                  (3)
pr> ^cl                             (4)
pr> s1 = "data101"                  (5)
pr> ^cl                             (6)
```

where

(1) The image name *data001* is assigned to the string variable *s1*.

(2) The CL script *center.cl* is executed. The command says *start up a new CL and take as its input the commands contained in the file center.cl*. When all the commands have been executed (or an error occurs), the new CL will close.

(3) Redefines the variable *s1* to be the image *data002*.

(4) Executes the CL script again using the caret (^) to recall the command from the user's *history*.

(5) Redefines *s1* yet again.

(6) Executes the script again.

This type of thing can then be repeated at will.

†The term *CL script* is used in this document to distinguish between *terminal scripts* and the scripts being discussed in this section. By definition, all scripts discussed in this document are *CL scripts*.

The second method is useful if there are many images to be operated on (say *data001* through *data550*). In this case, even the method used above could take a long time. The problem is that there is only one text file variable (**list**) at the CL level and the script is already using it (*i.e.*, there can be only one list open). One solution is to make a loop in which the image names are constructed. The CL script, *center.cl*, could be modified to the following:

```
for (i=1; i <= 550; i+=1) {                          (1)

    if (i < 10)                                      (2)
        s1 = "data00"//i
    else if (i < 100)
        s1 = "data0"//i
    else
        s1 = "data"//i

    if (access (s1 // ".imh")) {                     (3)
        list = "coord.lis"                           (4)
        while (fscan (list, x, y) != EOF)            (5)
            imcntr (s1, x, y, >> s1//".cen")         (6)
    }
    ;                                                (7)

}
```

where

(1) Declares a *for* loop that executes the commands enclosed by the curly braces while the integer *i* is incremented from 1 to 550. The term `i+=1` is equivalent to `i=i+1`.

(2) Begins an *if-else* block that constructs the image names *data???*. Note how the integer has been concatenated to a character string.†

(3) An *if* statement checks that the image (hence the `.imh` extension) exists before attempting to use it, preventing the task from aborting if it is unable to open the image.

(4-6) The main event (same as before).

(7) A bug in the CL (see footnote on page 1) requires the outer most *if* block to end with an *else* block, semicolon (`;`), or extra closing brace (`}`). This is not true of more deeply embedded *if* statements and **does not** apply to *procedure scripts* (§7).

The script is now even easier to invoke, since the image names are constructed and used within the script. The command sequence reduces to:

```
pr> cl < center.cl
```

†A more concise way to construct these names is described in Appendix V, under **Numerical Formats**.

As previously mentioned, it is possible to define and use variables other than the predefined CL variables. This can be done at any stage of script writing (*i.e.*, terminal scripts, CL-scripts). It is our experience, however, that when additional variables are called for one is usually in need of a *Procedure Script* (§7), and hence the discussion of user defined variables takes place there. On the other hand, the above example is a prime candidate for a rewrite involving a user defined variable as follows:

```
string *list1                                   (1)
list1 = "data.lis"                              (2)
while (fscan (list1, s1) != EOF) {              (3)
    list = "coord.lis"                          (4)
    while (fscan (list, x, y) != EOF)           (5)
        imcntr (s1, x, y, >> s1//".cen")        (6)
}                                               (7)
list = ""; list1 = ""                           (8)
```

where

(1) Defines a user variable *list1* as a special type of text file, exactly the same as the CL variable *list*. Discussion of the variable declaration will be deferred to §7.

(2) Assigns the text file *data.lis*, containing the names of the files, to *list1*.

(3) A *while* loop that reads the image name into the CL variable *s1*.

(4-6) Same as before.

(7) Closes the outer while loop.

(8) Closes the list files as a precaution. See the sixth paragraph on page 31 for details.

The script can still be run the same way: `cl < center.cl`. Note that to run the script with a different set of images or a different set of coordinates, the script itself might have to be edited, not just the listfiles. Even this (usually) petty annoyance will be eliminated when we discuss *procedure scripts* in §7.

With the introduction of multiple list variables, more complicated loops are possible. It is sometimes useful to read from two lists simultaneously, rather than within nested loops as above. For example, to center a different source in each of several images:

```
string *list1 = "data.lis"
list = "coord.lis"
while (fscan (list1, s1) != EOF && fscan (list, x, y) != EOF)
    imcntr (s1, x, y, >> s1//".cen")
```

- Note how *list1* is assigned an initial value while being declared.
- Note how a compound boolean expression can be used with the *while* statement. The `&&` is the logical **and** operator; the loop will continue as long as **neither** list reaches an EOF. There are also logical **or** (`||`) and **not** (`!`) operators available, and a full suite of comparison operators: `==`, `!=`, `>`, `<`, `>=` and `<=`.
- Of course, the intent of the *while* loop is that both lists should be the same length.

Alternately, if the coordinates are listed in the same (three column) file as the image names:

```
list = "data.lis"
while (fscan (list, s1, x, y) != EOF)
    imcntr (s1, x, y, >> s1//".cen")
```

5. Using Graphics Tasks

At some point, you will probably want to use a task that involves interactive plotting or curve fitting (*e.g.*, SPLOT). Most (if not all) of the tasks that use interactive curve fitting can also be executed non-interactively by setting the parameter *interactive = no*. The tasks will then use the values of the hidden parameters for fitting and will not produce plots. However, there may be times when one wishes to run the task non-interactively and be able to save certain plots for later examination. Similarly, you might want to use SPLOT to measure equivalent widths of a series of spectral features in a large number of images in an automatic (or batch) fashion and save only the log output and certain plots.

To solve these problems, you can submit the normal interactive cursor commands to a task in a *cursor file*, a text file where every line is a cursor command of the following form:

x y world_coord_sys command† *[colon_command_string]*

where

x, y	-	The cursor position in world coordinates.
world_coord_sys	-	The number of the world coordinate system. This is almost always **1**.
command	-	The cursor command to be executed.
colon_command_string	-	The string to follow a ':' in the *command* field.

Consider as an example, that one wishes to measure the equivalent width of Hα, Hβ, and Hγ using the **k** keystroke in SPLOT‡ (an operation requiring that the continuum be normalized to unity). The cursor file (say *equiv.cur*) would be the following:

```
6562 1 1 t     -  normalize the continuum
6562 .98 1 k   -  equivalent width of Hα
4861 .98 1 k   -  equivalent width of Hβ
4340 .98 1 k   -  equivalent width of Hγ
```

The first line does the normalization, and the next three lines measure the equivalent width of the three hydrogen lines at a flux level of .98. Note that the normalization could be done before entering SPLOT by using the CONTINUUM or ECCONTINUUM tasks. To execute the task on a spectrum, one could type a command such as:

```
splot data001 cursor="equiv.cur"
```

SPLOT would draw the plots on the standard graphics terminal (*i.e.*, the STDGRAPH data stream) but accept its cursor input from the file *equiv.cur*. The status lines would flash by at the bottom of the screen and the logfile (*splot.log*, by default) would be modified accordingly.

What if one does not want to see the plots? Graphics output may be redirected by using the standard redirection symbol (>) followed immediately by the appropriate token:

```
G  -  Redirects STDGRAPH
I  -  Redirects STDIMAGE      (not implemented)
P  -  Redirects STDPLOT
```

Most of the tasks send their graphics data over the STDGRAPH stream (regardless of the device used) and hence the `G` token is used most frequently. For example, to perform the above operation while throwing out the graphics (*i.e.*, no graphics, even on the terminal), the

†A limitation is that *cursor mode* keystrokes (capital letters) cannot be used, since the cursor file **replaces** the cursor mode interaction.

‡Note that in this example, SPLOT expects the data to be wavelength calibrated with the starting wavelength and the wavelength per channel in the (FITS standard) header keywords CRVAL1 and CDELT1, respectively (or alternatively, W0 and WPC).

command would be:

```
splot data001 cursor="equiv.cur" >G dev$null
```

which sends the plot to the null device.

Alternatively, one could save the plots in a metacode file for later examination:

```
splot data001 cursor="equiv.cur" >G plotfile
```

Now lets expand the above example. This time we'll make things a bit more complex by saying that we wish to save expanded plots of each of the three equivalent width determinations. The cursor file therefore is the following:

```
6562 1 1 t     -   normalize the continuum
6500 1 1 a     -   expand plot about Hα
6600 1 1 a
6562 .98 1 k   -   equivalent width of Hα
4500 1 1 a     -   expand plot about Hβ
5500 1 1 a
4861 .98 1 k   -   equivalent width of Hβ
4000 1 1 a     -   expand plot about Hγ
5000 1 1 a
4340 .98 1 k   -   equivalent width of Hγ
```

The CL script (say *equiv.cl*) would be:

```
list = "equiv.lis"                                          (1)
while (fscan (list, s1) != EOF) {                           (2)
    splot (s1, cursor="equiv.cur", > "dev$null",
        >G "tmp$eqp")                                       (3)
    gkiextract ("tmp$eqp", "3-5", >> "equiv.plt")           (4)
    delete ("tmp$eqp", ver-, >& "dev$null")                 (5)
}                                                           (6)
gkimosaic ("equiv.plt", nx=3, ny=1, fill+, rotate+,
    device="versatec")                                      (7)
gflush                                                      (8)
```

where

(1) Assigns the text file *equiv.lis* (containing the image names) to the CL variable *list*.

(2) A *while* loop based on reading the string variable *s1*.

(3) Executes SPLOT on the image *s1* using the file *equiv.cur* for cursor command input.

The first redirection (`> "dev$null"`) sends the status line (the line that shows up at the bottom of the terminal) to the NULL device.

The second redirection (`>G "tmp$eqp"`) uses the `G` token to redirect STDGRAPH output to a metacode file called *eqp* located in the system defined *tmp* disk area.

Note how a line is automatically continued onto the next line when it ends with a comma, an operator, or a redirection symbol. An explicit continuation is indicated by ending a line with a backslash (\), immediately before the newline. Continuing a line in the middle of a string or a variable name is not recommended.

(4) GKIEXTRACT is used to extract the three expanded plots from the metacode file, and to put them into a new file (*equiv.plt*) in the users current directory. Note that the *append* type of redirection is used.

(5) Deletes the temporary plotfile. The `ver-` turns off the verify option of the DELETE task, in case you have that set as your default. As a precaution, the redirection, `>& "dev$null"`, sends any error messages to the null file, *i.e.*, throws them away; such messages should only be produced if something goes wrong with your script. In general, `>&` indicates that both the STDOUT and STDERR streams should be redirected to a file, while `>>&` indicates that they should both be appended.

(6) Closes the *while* loop.

(7) Executes GKIMOSAIC to get a hardcopy of the saved plots. Note how *nx*, *ny*, *fill*, and *rotate* are used to control the format of the output. The plots will have a **portrait** aspect ratio since *rotate* is enabled. The parameters *nx* and *ny* have been swapped because each plot is rotated **individually**; for this reason, the plots will also be stacked in reverse order with Hα at the bottom of the page. The *fill* parameter is enabled to make the best use of space on the page.

(8) Flushes the graphics buffer so that the plots will be processed immediately. If this line is omitted the plots will only be generated at intervals (site defined - usually every eight plots) or when the user logs out of IRAF.

By now you should realize that there are different ways to get input to a task: CL parameter input, cursor input, standard input. There is yet another form of input called *ukey* for *unbuffered* (or *raw*) input. *Ukey* is a CL parameter (try `lpar cl`) that is normally NULL (`""`) but can be set to a file name. It is used for keystroke input to tasks like PAGE. SPLOT uses this kind of input when performing its *deblend* operation. Recall that when using the **d** keystroke, you are able to scroll the status line with the **+**, **-**, **r**, or **q** keys. In general, to tell if a plot task is using *ukey* input, note whether the cursor is displayed while the task is waiting for input. If the cursor is there, the task is waiting for cursor input - if not, the task is waiting for ukey input.

For example, if we wanted to run a script using the SPLOT deblend routine, the cursor file (*e.g., db3.cur*) would be the following (for an unconstrained three component deblend):

```
0 0 1 t      -   normalize the continuum
7400 1 1 a   -   expand the plot
7800 1 1 a
7565 1 1 d   -   set the deblend limits
7670 1 1 d
7604 1 1 m   -   mark the three line centers
7628 1 1 m
7641 1 1 m
0 0 1 q      -   exit marking mode
0 0 1 f      -   fit the lines
0 0 1 q      -   exit fitting mode
```

A file (*e.g., ukey.dat*) containing the single line

```
q
```

is also necessary. If you call deblend more that once in the cursor file, then you would need the appropriate number of `q`'s in the ukey file.

That part of the script containing the call to SPLOT would look like:

```
cl.ukey = "ukey.dat"
splot (s1, cursor="db3.cur", > "dev$null", >G "db3.plt")
cl.ukey = ""
```

where *ukey* is redefined back to its normal state so that it does not foul up some other task.

Now consider using a task that involves interactive curve fitting. For example, you might want to run the longslit task RESPONSE non-interactively on several quartz exposures, while saving plots of the fit, residuals, and ratio. First you should set up the fitting parameters using **epar**. Next, you can create a script (say *resp.cl*) of the following form:

```
list = "quartz.lis"                                              (1)
while (fscan (list, s1) != EOF)                                  (2)
    print ("yes") | response (s1, s1, "resp."//s1, int+,
        curs="resp.cur", > "dev$null", >>G "resp.plt")           (3)
gkimosaic ("resp.plt", nx=1, ny=1, dev="versatec")               (4)
gflush                                                           (5)
```

where

(1) Assigns the file *quartz.lis*, containing the names of the images to be operated on, to the CL file variable *list*.

(2) Sets up a *while* loop that reads the string variable *s1*. No braces '`{}`' are needed since there is only a single (albeit compound) statement within the loop.

(3) Executes RESPONSE interactively (so that plots are produced), taking cursor input from the file *resp.cur* and redirecting STDGRAPH output to the file *resp.plt*. Note that the graphics output is **appended** (`>>G`) to the metacode file.

The cursor file, *resp.cur* is the following:

```
1 1 1 j   -   plot the residuals of the fit
1 1 1 k   -   plot the ratio of the data to the fit
```

This file should behave as if it ended with the line, `1 1 1 q`, since the cursor loop will exit when the end of the file is reached. This can be verified interactively by typing an EOF character, usually `^Z` or `^D`, while in an interactive cursor loop. In this case, it actually requires two EOF's because of the logic used within this particular cursor loop. Which is to say that **cursor files should be verified** before running large batch plotting jobs!

The `int+` is needed so that plots will be created, although the routine is not being used interactively, from a strict point of view.

The *print* statement pipes the answer *yes* to the task when the query (which has been redirected to *dev$null*) asks if you want to do this *interactively*.

(4) Makes hardcopies of the plots. Setting *nx* and *ny* to one results in one plot per page. There will be three plots per image: the initial plot of the raw data, a plot of the residuals, and a plot of the ratio.

(5) Flushes the graphics buffer.

6. Defined Tasks Without Parameters

The next step in script writing is to define your scripts as IRAF tasks. This can be done from the CL (*i.e.*, any package) by simply typing a command of the form:

```
cl> task $taskname = path/scriptname.cl
```

where *taskname* is usually the same as *scriptname* (without the .cl) and *path* is the directory path to the script. The *path* is necessary to execute your script from directories other than the one where the script resides. The dollar sign ($) indicates that the task does not have parameters. Parameterized tasks will be discussed in the next section.

There are a few subtleties that need to be made clear.

- If you define a task while in a package (*e.g.*, onedspec) and then release that package with the bye command, the task definition also disappears. This is sometimes a useful and desirable effect. If you want to force your interactively entered task and parameter definitions to vanish (for instance if you are actively developing some task and want to start over with a clean slate), use the cl command to mark the state of the CL before making the definitions, and the bye command when you are finished or wish to start anew. This avoids unnecessary loading and unloading of packages. Type: help language.cl for more on this.†
- All packages containing tasks that your script uses **must** be loaded before execution of the script. **Packages should not be loaded within scripts**, although an earlier version of this manuscript suggested that they could be. For example, the script *center.cl* **should not** be modified to load PROTO as follows:

```
proto                                          # wrong!
list = "coord.lis"
while (fscan (list, x, y) != EOF) {
    imcntr (s1, x, y, >> s1//".cen")
}
```

There are known bugs with loading packages inside of scripts (these bugs seem to be restricted to procedure scripts, see §7), but more fundamentally, package loading and other functions that define the context in which a script executes are more properly performed before running the script. This context can be defined within your *loginuser.cl* file (see below), or you can declare a separate package in which your tasks and their context is defined. Package declaration is discussed more fully in §8.

Your tasks may be defined at login time (*i.e.*, when starting up the CL) by placing the task definitions in a file called *loginuser.cl*, which should be in your IRAF home directory. The *loginuser.cl* file will be executed by your *login.cl* file.‡

For example (in VMS):

```
set  scripts  = "usr0:[anderson.scripts]"              (1)
task $center  = "scripts$center.cl"                    (2)
task $equiv   = "scripts$equiv.cl"                     (3)
task $resp    = "scripts$resp.cl"                      (4)
task $mosaic  = "usr0:[anderson.irimage]mosaic.cl"     (5)
noao                                                   (6)
keep                                                   (7)
```

†See the footnote on page 17.

‡You could put the definitions directly in the *login.cl* file, but whenever you ran MKIRAF (for instance, for a new release) the definitions would be wiped out. The *loginuser.cl* is not clobbered by MKIRAF.

where

(1) Sets a *path* variable (*scripts*) to the path of a subdirectory that contains the scripts - it is not necessary to have all your scripts in the same place, but it is good practice. Note that the left bracket '`[`' may need to be escaped by a backslash '\' so that it won't be interpreted as a metacharacter: `"usr0:\[anderson.scripts]"`. This may also apply to dollar signs (`$`) and right brackets (`]`) and in general to any metacharacters that you want to include in a path or filename. It is usually better to avoid such characters but sometimes that is impossible.

(2-4) Define tasks for the script examples in the previous sections of this document.

(5) Defines a task where the CL script resides in another directory.

(6) Loads the NOAO package. You could load any packages you will need for your scripts within the *loginuser.cl* file.†

(7) The *keep* command tells the CL that the definitions of the *loginuser.cl* file (which is itself just a CL script) should be kept after completion of the script.

Now, for example, instead of typing the command

```
cl < center.cl
```

to run the script *center.cl*, one enters

```
center
```

NOTE: When using a path (environment) variable, a dollar sign ($) separates the path variable from the remainder of the path or file name. For example:

`scripts$taskname.cl`	Refers to the cl script `taskname`
`scripts$doc/taskname.hlp`	Refers to the help file for `taskname` in the subdirectory `doc`

Only one path variable can be used in a given path expression; a path of the form `path1$path2$path3` is **not** allowed (but see **Environment** in Appendix V).

In general, IRAF allows pathnames that are either: *relative* - '`../`' is the parent of the current directory, *virtual* - '`scripts`' as defined here, or *host system* - '`/tmp5/seaman/pixels/`' might be my *image directory* (or **imdir**, an IRAF virtual directory). A *relative* pathname is not very useful for a task statement, since the task will only be usable from this (remote) directory. *Host system* pathnames can be (and are) used, but there are some difficulties in specification, especially under VMS with all the dollar signs and brackets. One problem that people encounter when using UNIX pathnames is that the slash (`/`) is a directory *separator* in IRAF, so to specify the *root* directory use '`/.`' or a variant.

It is usually safest to define a set of IRAF path variables for your data and script directories and make all directory references with respect to these.

†There is a problem loading packages that have package parameters (*i.e.*, CCDRED) in your *loginuser.cl*. Changes made to the package parameters, even with **eparam**, will not be recognized unless you log out of the CL completely and then log back in again. This is a good reason to **always log out of IRAF correctly**, since any changes to such package parameters might otherwise be lost.

7. Defined Tasks With Parameters

The step to creating scripts with parameters is a large one since one can no longer be unaware of what might be termed the *full programming structure and syntax* of script writing. Do not let this last statement scare you ... you are still just stringing IRAF commands together in a file, but now all commands are executed in **compute mode** and hence you must be aware of all the commas, quotes and parentheses that are needed. In addition, all variables, including what used to be predefined CL variables, must be declared.

The best way to learn to write *procedure scripts* (a name used for scripts with parameters)† is by doing. Start with simple things and build up to the more complex. You will find the help pages of the **language** package useful at this stage of script writing; there are descriptions of each of the language elements (*e.g.*, **if**, **for** and **while**) as well as of general subjects (*e.g.*, **parameters**, **declarations** and **commands**).‡ The basic form of a procedure script is:

```
procedure taskname (arg1, arg2, ...)   arguments are query parameters
declaration of parameters
begin
        declaration of variables
        script commands
end
```

The hardest part of procedure script writing is getting the parameter and variable declarations correct. Be sure to declare **ALL** your variables in a procedure script.

The CL supports a variety of parameter datatypes, from the conventional *string*, *integer*, and *floating point* types, to the exotic *struct* and *cursor* types. There is no complex type in the CL. Arrays can be defined for the simpler datatypes, as described in Appendix V.

int Integer parameters and variables are used to store integer information. They are stored internally as a long integer, permitting at least 32 bits of precision.

real Real parameters and variables are stored internally with double precision. In general, they may be entered with or without a decimal point, and with or without an exponent. Note that the exponent should be entered using an E not a D.

bool Boolean parameters and variables may only have the values **yes** or **no**.

string String parameters and variables are used to store strings of ASCII characters.

struct Structs are character strings that are treated specially by the *scan* and *fscan* functions. *Scan* and *fscan* set structs to the remainder of the line being scanned without further parsing. **There is a 62 character limit for a struct.**

file File parameters are string parameters that are required to be valid file names. All operations legal on strings are legal on file parameters. Various checks on the accessibility or existence of a file may be automatically performed when a *file* parameter is used at runtime. (The checks are not implemented.)

†Not strictly true. Any CL script task (*i.e.*, not a procedure script) may have an **external** parameter file associated with it, in just the same way as a compiled task.

‡The CL will be confused if a language keyword is used as an argument to a task; for example, the command `cl> help procedure` will cause the CL to complain of a syntax error. To get around this enclose the keyword in quotes: `cl> help "procedure"`. Keywords are indicated with an asterisk (*) when you type: `help language`. You may also need to specify the package if there is a name collision: `help language.cl`, rather than `help cl`.

gcur, imcur

The cursor parameters have a character string value with a predefined cursor value format. When a cursor type parameter is read in query mode, the hardware cursor on the graphics terminal or image display is physically read. If the cursor parameter is list-directed, cursor input may also be taken from a list (text file) as in the previous section. For a more detailed discussion of cursor control in the CL, type: `help cursors`.

Frequently you may have a list of values, *e.g.*, numbers or file names, which you wish to analyze in turn. To do this you may use a list directed parameter. The parameter is defined with its value field set to the name of a file containing the list. The next time it is referenced its value will not be the string containing the file name, but the first value in the list. Subsequent calls will return later values in the list until an end-of-file is reached, at which point the parameter will appear to be undefined. The file may be rewound by reassigning the filename to the list parameter. Assigning the null string to a list parameter closes the associated list file.

For example, if you have a list of numbers (*i.e.*, a single column, or the first column in a multi-column file) then you could access that list in the following fashion:

```
procedure procedure_name ( ... )
    ...
int     *numlist
begin
        int     value, i
             ...
        numlist = filename
        for (i=1; i <= nlist; i+=1) {
                 ...
            value = numlist
                 ...
        }
end
```

where

- The asterisk (*) in the parameter declaration indicates that the parameter is *list-directed* (in this case, a list-directed integer parameter).
- *`filename`* contains the list of numbers and is assumed to be known explicitly or to be passed as a query parameter.
- *`nlist`* is the number of lines in the list (a constant or passed as a parameter).

NOTE: Only parameters of a procedure may be list-directed. Thus they must be declared in the parameter declaration part of the procedure (*i.e.*, before the *begin* statement). List directed parameters should not be query mode parameters because the CL may be confused when issuing prompts.

Note that in the example given on page 10, we have declared a list-directed variable (not a parameter). That particular example is **not** a procedure (there are no *begin* or *end* statements). More exactly, interactively declared variables are actually parameters of the CL itself. This can be verified by typing `lparam cl` after such a declaration.

A more common usage of list-directed parameters is to read files in conjunction with the *fscan* function (the CL variable *list* that we have been using in the previous sections is really a *list-directed struct*.). The next example uses a list-directed structure in this way. Consider a simple procedure task to print a file to STDOUT:

```
procedure printit (file_name)                    (1)

string file_name                                 (2)
struct *flist

begin
        struct line                              (3)

        flist = file_name                        (4)

        while (fscan (flist, line) != EOF)       (5)
            print (line)                         (6)
end
```

where

(1) Defines the procedure *printit* which has one query parameter called *file_name.*

(2) Declares the query parameter *file_name* to be a string and the hidden parameter *flist* to be a list-directed struct. This parameter is hidden because it is not included in the argument list of the procedure statement.

(3) Declaration of a struct *line* which will be the string associated with one line of text from the file.

(4) The list-directed struct *flist* is assigned the name of the file to be printed (*file_name*).

Note that it would be *incorrect* to make *flist* a query parameter as well as a list directed parameter, that is to eliminate the parameter **file_name** and have a single list directed query parameter for the task. Recall that whenever a query parameter is encountered (that was not specified on the command line), the user will be prompted for a new value of the parameter. In the case of a list directed parameter, the user would be expected to enter the value of the next member of the list, *not* the name of the list file itself.

(5) A *while* loop based on reading a line of text from *flist*.

Since there is only one statement inside the *while* loop, it is not necessary to enclose the loop with curly braces. This is also true for *if* statement blocks.

(6) Prints *line*.

NOTE: To use the task, enter the task definition (or put it in your *loginuser.cl* file):

```
task printit = path$printit.cl
```

A few important points:

- Procedure scripts **may not** be executed by CL redirection. They **must** be declared as tasks.
- The taskname, *printit*, and the filename, *printit*.cl **must** be the same; the CL locates a task's parameter file using the taskname, but a *procedure script*, in effect, has its parameter file within the script file itself (in between the `procedure` statement and the `begin` statement).
- PRINTIT is now a **fully functional** IRAF task, sharing the same facilities as any other task, including: command abbreviation, I/O redirection & pipes, learnable parameters and background job control.

Consider executing *printit* on the text file *data.txt* containing the following lines:

```
Hello world.
How are you?
Iamfine.
```

All one need do is enter the command

```
printit data.txt
```

to cause *data.txt* to be printed to STDOUT (usually the terminal) just as you see it above. You could have executed *printit*, having it query you for the *file_name*:

```
cl> printit
file_name: data.txt
Hello world.
How are you?
Iamfine.
```

The following executes *printit* again, this time printing the contents of the file *newfile.dat*. Note how the previous value has been remembered and is now presented as a default; if the <**return**> key is pressed in response to the prompt, *data.txt* will be printed again.

```
cl> printit > printit.out
file_name (data.txt): newfile.dat
```

Remember, *printit* is now a task like any other so you can access its parameters with commands such as: `lpar printit`, `epar printit` and `= printit.file_name`.

There is an important, but subtle, point in the above example. It lies in the declaration of the variable *line*. The tasks **scan** and **fscan** parse the contents of the string being read into segments delimited by spaces or tabs. These tokens are then deposited into the variables given in the command. If there are too many tokens they are discarded; if there are too few, the corresponding variables are not affected. Thus, if the variable *line* were declared as a **string** rather than a **struct**, the output resulting from the above example would be:

```
Hello
How
Iamfine.
```

An assignment to a **struct** terminates the scan of a record as the entire unscanned portion of the string is assigned to the struct (as far as **scan** and **fscan** are concerned, a *struct* is a *string* with blanks and tabs in it). Thus any struct should be the last parameter in a **scan** or **fscan** statement: the CL will report an error if this is not the case. Please note that the current CL enforces a **limit of 62 characters for the length of a struct**. Scanned lines longer than 62 characters will be silently truncated.

Now, let's make a procedure script for the centering example that we have been using throughout this document. The following example has two query parameters: the image list to be operated on, and the name of the coordinate file containing the initial guess for the object centers. There is one hidden parameter (*cboxsize*) which allows the user to control the same parameter in the call to *imcntr* itself.

- Note how the query parameters are specified merely by including them as arguments in the procedure statement. All other parameters will be hidden unless the mode is explicitly set. The parameters have been set up with appropriate prompts, initial values, and minimum values (for *cboxsize*). See the DECLARATIONS help page in the LANGUAGE package for more about parameter setup.

- Note also how the *.imh* extension of image names is stripped from the name. This is especially useful when one needs to modify or append to the name. The IRAF tasks themselves will assume the appropriate image extension, so it is not really necessary to carry it along in the variable. A limitation in coding the script this way is that it will only handle the *standard* IRAF image format. If your scripts are to be used on other formats, *e.g.*, STSDAS format images, they will have to take the different extension into account.
- Note that comment statements can be included by using a '#' as the delimiter. Everything on the line after the '#' is considered a comment.
- Note the use of the MKTEMP function to generate a name for the (as yet uncreated) temporary file in which to store the individual image names. A file with this name will not exist (immediately) after MKTEMP is invoked.

```
procedure center (images, center_file)

string  images          {prompt="Images to be used"}
string  center_file     {prompt="Text file of approximate centers"}
int     cboxsize        {5, min=5, prompt="Size of extraction box"}
struct  *imglist
struct  *ctrlist

begin
        int     cbox, i
        real    xinit, yinit
        string  img, imgfile, outfile, ctrfile

        # Make sure the noao and proto packages are loaded
        if (! defpac ("proto"))
            bye

        # Expand the image template into a text file list
        imgfile = mktemp ("tmp$ctr")
        sections (images, option="fullname", > imgfile)

        # Get the rest of the parameters
        ctrfile = center_file
        cbox    = cboxsize

        # Open the list of images and scan through it
        imglist = imgfile
        while (fscan (imglist, img) != EOF) {

            # Get rid of the ".imh" extension
            i = strlen (img)
            if (substr (img, i-3, i) == ".imh")
                img = substr (img, 1, i-4)

            # Make up the output file name.  This file will
            # appear in the same directory as the image!
            outfile = img // ".cen"

            # Do the centering
            if (access (img // ".imh")) {
                ctrlist = ctrfile
                while (fscan (ctrlist, xinit, yinit) != EOF)
                    imcntr (img, xinit, yinit, cbox=cbox, >> outfile)
            }

        }

        # Clean up
        delete (imgfile, ver-, >& "dev$null")
end
```

The script may be defined as a task in the normal way:

```
task center = path$center.cl
```

In addition, the *library packages*† that define the tasks used within the script should be loaded before the task is executed. The definition of tasks and the loading of needed packages, as well as the installation of help files for your tasks and packages, are usually performed within some exterior package; **tasks are happiest within packages**. The creation of your own package is described in the following section (§8).

Note that since this procedure uses the SECTIONS task internally, it is not necessary to explicitly read from a list of images, although that is still an option; a *listfile* (or *@-file*, pronounced "atfile") is a type of image template. You simply enter commands of the form:

```
cl> center data001,data005,nite2123 coord.lis
```

or

```
cl> center nite1*,nite2* coord.lis
```

or

```
cl> center @nite1.list coord.lis
```

In the last example, the file *nite1.list* is assumed to contain a list of image names.

†To define SECTIONS and INCNTR, we need **images**, **noao**, and **proto**; or more succinctly, just **noao** and **proto**, since **noao** loads one but contains the other.

8. Making Your Own Package

If you have written a script that is of general interest to the user community at your site, then you might want to install it in the LOCAL package of IRAF. LOCAL has been set up for just this purpose. Similarly, if you have several scripts that are useful only to yourself, you may want to make your own personal package so that you will not need to define the tasks at login time. In essence, when loading an IRAF package all you are really doing is executing a particular IRAF script.

Installing Scripts in LOCAL

At most IRAF sites, only the system manager will have the authority (and necessary privileges) to install a task in LOCAL. Thus, this process will not be described here. If you are the system manager, please refer to the *IRAF Site Manager's Guide* for further information. Other documentation can be found under the LOCAL package in *Volume 1B* of the *IRAF User Handbook* and in the *README* files in the *local$* and *local$src* template directories that came with your release.

Installing Your Personal Package

To build your own personal package, do the following:

1. Create a subdirectory and place your scripts in it.
2. In that subdirectory, edit a file called *mypackage*.cl, similar to the file below, where *mypackage* is the desired name of your package. The **package** statement must occur before the task definitions. The **clbye** causes the script to accept and interpret user commands from the STDIN (unless redirected) until the command **bye** is entered, causing the package task to finish. The following is the file for the package, *edsprog*:

```
# Package script task for the EDSPROG package

# load necessary packages
noao
imred
proto
imdebug
vtel

set     eraprog         = "usr0:[anderson.iraf.scripts]"†

package edsprog

set     helpdb          = "eraprog$edsprog.db"

task    center          = "eraprog$center.cl"
task    composit        = "eraprog$composit.cl"
task    cube            = "eraprog$cube.cl"
task    iprof           = "eraprog$iprof.cl"
task    mosaic          = "eraprog$mosaic.cl"
task    smimage         = "eraprog$smimage.cl"

clbye()
```

†Under UNIX this might be: `set eraprog = "/u2/anderson/iraf/scripts/"`. Note that **the trailing slash is required** but that the quotes are optional in most circumstances.

2. If you are going to have help pages for some of your tasks then:

   a) Put the **.hlp** files in a subdirectory (conventionally named **doc**).

      To create a manual page of the proper format: load the **softools** package, execute **mkmanpage**, and set the parameter *module* to your taskname.

      MKMANPAGE comes up inside your default editor with the **lroff** skeleton of an IRAF manpage. Edit the manpage file to fill in the necessary data. You may wish to print out the *.hlp* file of some other task beforehand to use as an example by simply changing to the package directory of your choice (*e.g.*, `cd images$doc`) and examining the *.hlp* files. You might find the the help page for the LROFF task in the softool package useful.

   b) In your main package subdirectory edit a file called *mypackage*.hd. The following is the *.hd* file for the package *edsprog*:

```
# Help directory for the EDSPROG package

$doc            = "usr0:[anderson.iraf.scripts.doc]"

center          hlp =doc$center.hlp,    src = center.cl
composit        hlp =doc$composit.hlp,  src = composit.cl
cube            hlp =doc$cube.hlp,      src = cube.cl
iprof           hlp =doc$iprof.hlp,     src = iprof.cl
mosaic          hlp =doc$mosaic.hlp,    src = mosaic.cl
smimage         hlp =doc$smimage.hlp,   src = smimage.cl
```

   c) In the same subdirectory, edit a file called *_mypackage*.hd. This file (*e.g.*, *_edsprog.hd*) will be used to define the help for the package itself:

```
# Help definitions for EDSPROG itself

edsprog         men = edsprog.men,
                hlp = ..,
                sys = edsprog.hlp,
                pkg = edsprog.hd,
                src = edsprog.cl
```

      Note how this file points to the previous file with the *pkg* assignment.

   d) There is one more level of complexity to this scheme. The two files above are enough to access the help pages for the tasks within the package, *edsprog* (or *mypackage*), but will not allow you to access the help for *edsprog*, itself. This is because there is nothing to point to the root of any given help tree. We need one more file to serve as the root (call this file, *root.hd*):

```
# Root help directory for EDSPROG

_edsprog        pkg = _edsprog.hd
```

   e) Load the **softools** package and execute **mkhelpdb**:

```
so> mkhelpdb root.hd helpdb.mip
```

      The '`.mip`' extension indicates that the file uses a machine independent protocol. This same help database can be used on machines with completely different architectures.

f) If you wish, you could also create a menu file (*mypackage*.men) in the main directory. Such a file, called *edsprog.men* to agree with the examples above, might be:

```
  center - Determine the center of the Seeing Monitor image
composit - Mosaic fiber optic data in proper positions
    cube - Analyze Seeing Monitor data
   iprof - Construct the instr. profile from spectral features
  mosaic - Mosaic solar magnetograms
 smimage - Make a dicomed of Seeing Monitor data
```

g) Finally, it is possible to add a help file (*edsprog.hlp*) to describe the package itself. This is not the same as the menu file in *edsprog.men* and may contain a design document for the package or a cookbook for users, for example. If you add such a package help file the command: `help edsprog`, will display this help file rather than the menu that is usually listed. To see the menu type: `help`, while the *edsprog* package is loaded.

3. In your *loginuser.cl* file, add the definition for your package and modify the current helpdb:

```
task $mypackage = path$mypackage.cl
reset helpdb = (envget ("helpdb") // ",path$helpdb.mip")
```

4. Now (after logging in) you may load your package by simply typing `mypackage`. The help pages will be available online whether your package is loaded or not.

   Note how the help database allows you to examine the scripts:

```
cl> help center option=source
```

   This will also work with many normal IRAF (SPP or CL script) tasks.

Appendix I

Running Scripts as Background Jobs

IRAF tasks cannot write into their parameter files while they are running in the background. This allows the user to execute the same task (or tasks) in the foreground while the background job is running. There are two consequences that you should be aware of, if you are going to run your scripts in the background.

1. Various tasks such as IMGETS and SECTIONS (in the images package) will not work in the background since they store output results in their own parameters, *imgets.value* and *sections.nimages*, respectively.† Previous versions of this manual declared that "any task using IMGETS [or SECTIONS] must be run in the foreground". Currently, there seems to be a partial or complete workaround available. If each of the tasks for which an output parameter is used has its parameters "cached" then the tasks should function correctly in the background. This can be done using the *cache* task in the language package. The effect of parameter caching is to create a copy in memory of the parameters from the parameter file on disk.‡ For example, the following line should be placed near the beginning of a script that uses IMGETS (after the parameter and variable declarations but before using IMGETS itself), if the script is to be run in the background:

   ```
   cache imgets
   ```

   The user should use *cache* with caution. Any script for which there is no easy alternative is probably near the desirable limit of complexity for a script. Note that the HSELECT task is a much more general approach to the problem of header keyword access from the CL, and may be more appropriate for a given application.

2. Since CL scripts are executed by a command interpreter (*i.e.*, they are not compiled executables), their behavior may be altered in the background if the user changes the value of any hidden parameters of any task used by the script while the script is running. Note that the only example in this document of a script running in background is for MKSCRIPT (§2) where **every** parameter of each task is specified in the script itself.

   Thus, if the script writer wishes the script to execute flawlessly while in background, all the **critical** parameters for any IRAF task used should be specified in the script.

†You can ignore the difficulties covered in this appendix if you plan to use SECTIONS (a very useful task) to expand an image template, but do not plan to use the *nimages* parameter to count images.

‡Note that tasks that are cached within a script will be removed from the cache when the script exits. To explicitly remove a task from the cache, *unlearn* it: `unlearn imgets`. The parameter file (on disk) for a cached task is only updated under some circumstances, for instance, when a script exits normally. To explicitly update a task, use the *update* task: `update imgets`. This will normally not be necessary.

Appendix II

File and Image Templates in Scripts

Many IRAF tasks have parameters that accept file or image *templates*. As in many computer command languages, a selection of meta-characters (*wildcards*) accomplishes a variety of *file expansion* functions. The file expansion, or template, mechanism can be used to enhance your scripts, once certain subtleties in usage are mastered. The template expansion syntax, itself, is explained more fully in *A User's Introduction to the IRAF Command Language*. We will limit ourselves to a discussion of applying that syntax in a reliable fashion once the desired template is chosen.

Two basic questions are encountered with tasks that use more than one file (or image) template parameter.

1) How does one express an output template?

   In IRAF, as in UNIX and VMS, templates are expanded on the set of pre-existing files in the file system. A *wildcard* character, such as '`?`' (match a single character) or '`*`' (match any sequence of characters) only has meaning relative to the files that already exist on the disk.† The following spurious command illustrates this:

```
cl> imrename nite1*.imh n1*.imh
ERROR: Different number of old and new image names
```

   The problem is that there are no images matching the template, `n1*.imh`, when the command is issued, which is the whole point in this instance! The VMS DCL provides a partial solution to this difficulty through the notion of a **temporary default**. Certain of the DCL wildcards (usually the '`*`'), when used in an **output** template, will pick up their value from the corresponding field of the command's **input** template. This is only a partial solution since only the entire field (filename, extension or version) can be defaulted. The UNIX Bourne shell and C-shell don't really address the problem at all. Note that in any of these command languages, a multiline script can be written to solve the problem.

   IRAF offers several possibilities for creating output file lists. The most straightforward is to simply list the unambiguous (*i.e.*, no wildcards) filenames, separated by commas:

```
cl> imrename nite1.0001,nite1.0002 n1.0001,n1.0002
```

   Note that spaces separate arguments in command mode, but that commas (with NO spaces) separate specific members of a list. In compute mode this would be written:

```
cl> imrename ("nite1.0001,nite1.0002", "n1.0001,n1.0002")
```

   where the file lists have been recognized as string literals and are thus quoted. Note that string variables can be used instead of literals.

†One corollary of this is that a user must consider the image file extension (*e.g.*, '`.imh`') when constructing an image template. Usually an image template will have a terminal '`*`', *e.g.*, `nite1.*`, which automatically includes the `.imh`. In cases in which the '`*`' is embedded within the image names or in which a '`?`' is used to match a single character, the extension must be explicitly written: `nite1.000?.imh`. This may also apply when using the '`[]`' wildcards and others.

A second option is to use *listfiles*:

```
cl> imrename @nite1.list @n1.list
```

The two lists should be carefully constructed to ensure that corresponding lines match. In IRAF version 2.8, a restriction was relaxed requiring that listfiles be in the current directory. Now a remote pathname (the *tmp$* directory in this case) will work:

```
cl> imrename @tmp$nite1.list @tmp$n1.list
```

It has always been possible to use remote pathnames within the list itself.

The third option is to use the IRAF *template operators* to make the new names "on the fly":

```
cl> imrename nite1.*.imh %nite%n%1.*.imh
```

The '%' (*replacement*) operator expands the template using the string between the first two percent signs but replaces this string with that between the second two percent signs before using the names. Another example:

```
cl> imrename nite1.*.imh new//nite1.*.imh
```

The '//' (*concatenation*) operator works to prepend (as in this instance) or append a string to an already existing filename. Note that this will work with a *listfile*, also:

```
cl> imrename @nite1.list new//@nite1.list
```

2) How does one guarantee that each of the templates will remain in step?

IRAF file templates are typically expanded in a machine dependent order, depending on the particular task. Some systems will provide a sorted list of matching files, and some will not. Problems arise when a command is constructed with multiple templates. Each template can potentially be expanded in a different sort order, resulting in the matching together of unintended files or images. An example is the command:

```
cl> imarith num.* / den.* %num%quo%.*
```

Note that we have taken the ideas of the previous section to heart and have constructed an output image list. The problem is that the two input templates, `num.*` and `den.*`, while presumably containing the same number of images, may not expand in the same order. One solution to this is hinted at by the usage of the output template. By explicitly tying the output image names, one by one, to the input (numerator) image names, this problem has been avoided. A corrected command line might be:

```
cl> imarith num.* / %num%den%.* %num%quo%.*
```

The more usual way to avoid this gotcha is to use listfiles for each image template:

```
cl> imarith @num.list / @den.list @quo.list
```

This is especially true for use in scripts, where the listfiles and the individual images may have been created (and may be deleted) within the script. An example using the compute mode of the CL, with string variables containing the individual list names:

```
          ...

nlist = mktemp ("tmp$num")
dlist = mktemp ("tmp$den")
qlist = mktemp ("tmp$quo")

     in some fashion, write the image names into the files

imarith ("@"//nlist, "/", "@"//dlist, "@"//qlist)
delete (nlist//","//dlist//","//qlist, ver-, >& "dev$null")

          ...
```

Note the explicit concatenation of the '@' to the listfile names and the use of MKTEMP and DELETE to manage the temporary files.

Appendix III

An Example Including a Help Page

This script is used in the local package at NOAO/Tucson to change the node name of an image's pixel file that was written into the image's header file when the image was created. This is useful for heavily networked sites, such as NOAO, in which a single large file server supports multiple diskless workstations.

Since IRAF has no special knowledge of the local network configuration, there is no way for it to know that an image that was created on a diskless node actually resides on the file server. Any later access of this image from a different machine will cause IRAF networking to be used for access, rather than the host level networking software, such as NFS. This is typically undesirable not because of any inferiority of the software, but because of increased traffic since the local network will be traversed multiple times. Conversely, sometimes it is desirable to force IRAF networking to be used. This task allows either option to be taken.

The script should be placed in the file *chnode.cl* in a convenient directory, and the usual task statement should be issued, either interactively, in a package script or in the user's *loginuser.cl*:

```
task chnode = path$chnode.cl
```

Chnode.cl:

```
procedure chnode (images)

string  images          {prompt="Images to hedit"}
string  node   = ""     {prompt="New node for pixfile path"}
bool    verify = no     {prompt="Verify header editing?"}
bool    show   = no     {prompt="Print record of edits?"}
struct  *rlist

begin
        string  img, rfile, newnode, pin, pout, junk

        rfile = mktemp ("tmp$tmp.")
        rlist = rfile

        if (node == "") {
            pathnames ("home$", >> rfile)
            junk = fscan (rlist, newnode)
            newnode = substr (newnode, 1, stridx ("!", newnode) - 1)
        } else
            newnode = node

        hselect (images, "$I,i_pixfile", yes, >> rfile)

        while (fscan (rlist, img, pin) != EOF) {
            if (substr (pin, 1, 4) == "HDR$")
                next
            pout = newnode // substr (pin, stridx ("!", pin), strlen (pin))
            hedit (img, "i_pixfile", pout, add-, del-, up+,
                verify=verify, show=show)
        }

        rlist = ""
        delete (rfile,  ver-, >& "dev$null")
end
```

A few comments:

- Only the parameter with no obvious default, *images*, is specified as a query parameter. *Node* is a hidden parameter, with the current node (the NULL string) as the default.

- An alternate design could have used the *$nargs* predefined variable to branch based on the number of arguments used when the task is run. In this case, *node* would be declared as a query parameter (*i.e.*, included in the procedure argument list along with *images*) and the following would replace the *if* statement:

  ```
  if ($nargs < 2) {
  ```

  The present coding and behavior were chosen since the majority of users were expected to choose to default to the current node.

- Note how the pathname of the user's *home$* directory is used to find the current node.

- Note how MKTEMP and DELETE are used to manage the temporary file and how the file is placed into the *tmp$* directory to keep the user's own directory tidy.

- Note how HSELECT can produce nicely formatted tabular output and can include the current image file name with the '`$I`' macro.

- The script appends the output from HSELECT to the same temporary file that was (potentially) used to scan the node name. This **only** works because the previous FSCAN was not allowed to read to the EOF. When the list directed parameter, *rlist*, is assigned the temporary file name as a value, that file is opened for reading. The file remains open until:

  a) It is explicitly closed with `rlist = ""` or by assigning a new file name.

  b) It is scanned to the end of file (`EOF`).

  When in doubt, **always close list files** by explicitly assigning the NULL string, `""`, to the list directed parameter. This will prevent trouble if you try to delete the file later, since the system will not allow you to delete an open file.

  Note that reassigning the same file name to a list has the effect of **rewinding** the file to the beginning.

- The script checks explicitly for images that use the *HDR$* syntax to specify that the pixel file is in the current directory or in some subdirectory of the current directory; the script simply skips to the `next` file in this case.

- Note how the two boolean parameters, *verify* and *show*, are passed through to the HEDIT task, allowing the user to decide how the task is run.

- There are other ways to address this problem and other networking issues, in general. These are lightly touched upon under **Networking** in Appendix V.

This is the verbatim text of the help file, *chnode.hlp*, which may be installed as described in section §8 above. Type `help lroff` for a description of the formatting commands.

Chnode.hlp:

```
.help chnode Apr89 local
.ih
NAME
chnode -- change the pixel file node prefix in image headers
.ih
USAGE
chnode images
.ih
PARAMETERS
.ls images
The images whose headers are to be edited.  A list or template is allowed.
.le
.ls node = ""
The new node prefix to be edited into the pixel file pathname in each
header.  If \fBnode\fR is "" (null) then the task will use the local
node.
.le
.ls verify = no
Verify each header modification?
.le
.ls show = no
Show each header modification?
.le
.ih
DESCRIPTION
IRAF places the pathname of the pixel file into each image header.
This pathname includes the local node on which an image was created.
Since IRAF doesn't have any special knowledge about the configuration
of the local network, there is no way to distinguish between images
that are truly remote and those that reside on a file server and are
being accessed from a variety of other nodes (workstations).
.sp
CHNODE allows the pixel file node prefix to be changed in a list of
image headers.  This is useful for controlling the type of file access
that will be used in a file server environment.  If the node prefix is
the local node, then any pixel access will occur using the host
operating system facilities (e.g. local disk I/O or NFS).  If the node
prefix is other than the local node, then pixel access will occur using
iraf networking to the remote machine.  How the remote machine accesses
the pixel file itself is, of course, site and machine dependent and may
also involve operating system level network access.
.ih
EXAMPLES
1)  Change the pixel pathnames to include the local node, showing
each header modification:

        chnode *.imh show+

It is assumed that the images were created on other machines that use
the same fileserver.

2)  Change the pixel pathnames to include a remote node, for instance
the file server itself, while logged onto a client node:

        chnode *.imh node=orion

This forces IRAF networking to be used for pixel access, instead of NFS
(for example).
.ih
SEE ALSO
imheader, hedit, hselect, netstatus
.endhelp
```

This is the formatted output from the help command. You should be able to decipher the formatting directives by comparison with the raw text.

```
CHNODE (Apr89)                    local                    CHNODE (Apr89)


NAME
    chnode -- change the pixel file node prefix in image headers

USAGE
    chnode images

PARAMETERS

    images
        The images whose headers are to be edited.  A list  or  template
        is allowed.

    node = ""
        The  new  node  prefix to be edited into the pixel file pathname
        in each header.  If node is "" (null) then  the  task  will  use
        the local node.

    verify = no
        Verify each header modification?

    show = no
        Show each header modification?

DESCRIPTION
    IRAF  places  the pathname of the pixel file into each image header.
    This pathname includes the local node on which an image was created.
    Since   IRAF   doesn't   have   any   special knowledge about the
    configuration of the local network, there is no way  to  distinguish
    between  images  that  are  truly  remote and those that reside on a
    file server and are being accessed from a  variety  of  other  nodes
    (workstations).

    CHNODE  allows the pixel file node prefix to be changed in a list of
    image headers.  This is useful for  controlling  the  type  of  file
    access  that will be used in a file server environment.  If the node
    prefix is the local node, then any pixel  access  will  occur  using
    the  host operating system facilities (e.g. local disk I/O or NFS).
    If the node prefix is other than the local node, then  pixel  access
    will  occur  using  iraf  networking to the remote machine.  How the
    remote machine accesses the pixel file itself is,  of  course,  site
    and  machine  dependent  and may also involve operating system level
    network access.

EXAMPLES
    1)  Change the pixel pathnames to include the  local  node,  showing
    each header modification:

            chnode *.imh show+

    It  is  assumed  that the images were created on other machines that
    use the same fileserver.

    2)  Change  the  pixel  pathnames  to  include  a  remote  node,  for
    instance the file server itself, while logged onto a client node:

            chnode *.imh node=orion

    This  forces IRAF networking to be used for pixel access, instead of
    NFS (for example).

SEE ALSO
    imheader, hedit, hselect, netstatus
```

Appendix IV

More Examples

Making a Mosaic

The following script (*mosaic.cl*) was written to mosaic and label solar magnetograms for a visiting scientist at NSO.

- Note how comment statements can be put at the end of executable commands.
- Note how IMCOPY is used to mosaic the smaller magnetograms into the larger template.
- Note the use of the backslash escaped quotes (\") to put the double quotes into the CRTPICT command string to be executed. An alternate way is to use a single quote within a doubly quoted string:

```
print ("crtpict (´", oimg, "*´) & batch") | cl
```

- Note how the CRTPICT job is set up and submitted as a background task. It is sent to the VMS batch queue so that the background process will not die when the script finishes.

  Although CRTPICT is run in the background (*i.e.*, batch), the MOSAIC task itself may have problems in the background. Since the parameters of CRTPICT are set within the script, the parameter file **will not be updated** if MOSAIC is run in the background. To fix this, either set all of the critical CRTPICT parameters in the submitted command line or remove the CRTPICT interaction entirely. The most interesting and most time consuming part of the script is the mosaicing itself; CRTPICT can be easily executed after the mosaics are complete. Additionally, the *Dicomed* queue at NOAO encounters various bottlenecks as the pictures make their way through the local network. *Dicomeds* are best made explicitly after having checked the queue, rather than at the tail end of a time consuming script.

```
# MOSAIC -- IRAF script to mosaic solar magnetograms (400x256 pixels,
#           20 per mosaic) into a 2060x2000 image, with time and date of
#           exposure written below each, and submit the mosaic to CRTPICT

# Ed Anderson (NOAO) June, 1986 - revised by Rob Seaman

procedure mosaic (images, outimg)

string  images          {prompt="Magnetogram images"}
string  outimg          {prompt="Root name for mosaiced images"}
bool    dicomed         {yes, prompt="Submit mosaic to dicomed?"}
struct  *imglist

begin
        int     x = 2076
        int     y = 1745
        int     nmos = 1
        int     x1, y1, tx, ty, nimg
        string  imgfile, tmplt, img, iimg, oimg

        # Check that the necessary packages are loaded
        if (! defpac("proto") || ! defpac("imdebug") || ! defpac("vtel"))
            bye

        # Get the query parameters
        iimg = images
        oimg = outimg
```

```
        # Expand the image template into a text file list
        imgfile = mktemp ("tmp$mos")
        sections (iimg, option="fullname", > imgfile)

        # Create the template image (not image template)
        tmplt = mktemp ("tmp$mos")
        mkimage (tmplt, 2060, 2000, "s", -300, "magnetogram mosaic")

        # Do the mosaicing
        imglist = imgfile
        for (nimg = 1; fscan (imglist, img) != EOF; nimg += 1) {

            x -= 415

            if (x <= 0)
                { x = 1661; y -= 512 }

            x1 = x + 399; y1 = y + 255
            tx = x + 50;  ty = y - 100

            imcopy (img, tmplt // "["//x//":"//x1//","//y//":"//y1//"]",
                verbose-)
            pimtext (tmplt, img, ref+, x=tx, y=ty, val=200, setbgnd-,
                xmag=3, ymag=3)

            # Setup a new template if required
            if (nimg == 20) {
                imcopy (tmplt, oimg // nmos, verbose-)
                imreplace (tmplt, -300, 0, lower=INDEF, upper=INDEF)
                x = 2076; y = 1745
                nmos += 1
                nimg = 0
            }

        }

        # Rename the last partial image
        if (nimg > 1)
            imrename (tmplt, oimg // str (nmos + 1))

        # Submit all mosaiced images to CRTPICT
        if (dicomed) {
            unlearn crtpict
            crtpict.ztrans = "min_max"
            crtpict.z1 = -200.; crtpict.z2 = 200
            crtpict.image_fraction = 0.95
            crtpict.graphics_fraction = 0.
            crtpict.greyscale_fraction = 0.
            print ("crtpict (\"", oimg, "*\") & batch") | cl
        }

        # Clean up
        delete (imgfile, ver-, >& "dev$null")
        imdelete (tmplt, ver-, >& "dev$null")
end
```

This example is originally from a previous version of this document. It has aged less gracefully than other examples that have carried over. A characteristic of scripts is that they need frequent revision, since the tasks that they use may change from release to release as IRAF matures. In this case there is a new proto task called TVMARK in IRAF V2.8 which might serve better for labeling the individual frames. More generally, there is active IRAF software development proceeding in mosaicing and registration.

Pixel by Pixel Fitting

The next example will perform a linear least squares fit at each pixel of a set (stack) of images. Thanks to Mike Merrill for the underlying concept.

- The usefulness of this script hinges on the data having been acquired in a certain fashion. By sampling (reading out but not clearing) a CCD at rapid intervals while making an exposure, a measure of the *slope* and *intercept* of the incident light can be constructed on a pixel by pixel basis.
- Note that the large number of IMARITH references may be replaced by a small number of corresponding IMCALC references, once that task is finished.
- The calculation of error images is left as an exercise for the interested student. The image statistics (Poisson, *etc.*) and the choice of measures (standard error, χ^2, uncertainty in the coefficients) vary enough to make the implementation non-trivial.

```
# IMFIT  --  perform a pixel by pixel, linear least squares fit
#            to a set of images.

#       The function is:  y = a + b*x
#
#       The coeffs are:   a = (SUMY * sumx2 - sumx * SUMXY) / delta
#                         b = (nimg * SUMXY - sumx * SUMY)  / delta
#       where:        delta = (nimg * sumx2 - sumx * sumx)
#
#       UPPERCASE are coadded images, lowercase are scalars.

procedure imfit (images, intercept, slope)

string  images          {prompt="Images to fit"}
string  intercept       {prompt="Output intercept image"}
string  slope           {prompt="Output slope image"}

string  keyword = ""    {prompt="Keyword for x values"}

struct  *ilist
struct  *xlist

begin
        string  img, slp, intcpt, sumy, sumxy
        string  ifile, xfile, tfile, tmp1, tmp2
        real    x, delta

        real    sumx  = 0.0
        real    sumx2 = 0.0
        int     nimg  = 0

        # Make names for temporary images
        sumy    = mktemp ("tmp$imf")
        sumxy   = mktemp ("tmp$imf")
        tmp1    = mktemp ("tmp$imf")
        tmp2    = mktemp ("tmp$imf")

        # Make names for temporary files
        ifile   = mktemp ("tmp$imf")
        tfile   = mktemp ("tmp$imf")
        xfile   = mktemp ("tmp$imf")

        # Prompt for the images and expand the template
        sections (images, option="fullname", > ifile)
```

```
        # Prompt for the other query parameters
        intcpt  = intercept
        slp     = slope

        # Read the image headers and compute the scalar sums
        ilist = ifile; xlist = xfile
        while (fscan (ilist, img) != EOF) {
            # Temporary images, 1 per input image
            print (mktemp ("tmp$imf"), >> tfile)

            hselect (img, keyword, yes, >> xfile)
            if (fscan (xlist, x) != 1)
                error (1, "Keyword ‘" // keyword // "’ not found in " // img)

            sumx  += x
            sumx2 += x * x
            nimg  += 1
        }
        ilist = ""; xlist = ""

        delta = nimg * sumx2 - sumx * sumx

        # Calculate SUMY
        imcombine ("@" // ifile, sumy, option="sum", outtype="real",
            logfile="", exposure-, scale-, offset-, weight-)

        # Calculate SUMXY
        imarith ("@" // ifile, "*", "@" // xfile, "@" // tfile,
            pixtype="real", calctype="real", verbose-, noact-)
        imcombine ("@" // tfile, sumxy, option="sum", outtype="real",
            logfile="", exposure-, scale-, offset-, weight-)
        imdelete ("@" // tfile, ver-, >& "dev$null")

        # Calculate the intercept
        imarith (sumy, "*", sumx2, tmp1, pix="r", calc="r", verb-, noact-)
        imarith (sumxy, "*", sumx, tmp2, pix="r", calc="r", verb-, noact-)
        imarith (tmp1, "-", tmp2, intcpt, pix="r", calc="r", verb-, noact-)
        imarith (intcpt, "/", delta, intcpt, pix="r", calc="r", verb-, noact-)
        imdelete (tmp1//","//tmp2, ver-, >& "dev$null")

        # Calculate the slope
        imarith (sumxy, "*", nimg, tmp1, pix="r", calc="r", verb-, noact-)
        imarith (sumy, "*", sumx, tmp2, pix="r", calc="r", verb-, noact-)
        imarith (tmp1, "-", tmp2, slp, pix="r", calc="r", verb-, noact-)
        imarith (slp, "/", delta, slp, pix="r", calc="r", verb-, noact-)
        imdelete (tmp1//","//tmp2, ver-, >& "dev$null")

        # Clean up
        imdelete (sumy//","//sumxy, ver-, >& "dev$null")
        delete (ifile//","//tfile//","//xfile, ver-, >& "dev$null")
end
```

Appendix V

Topics Not Discussed

In the interest of completeness, several topics that have eluded discussion will be covered here. These are a mixed bag of advanced topics or fringe capabilities (sometimes it just depends on your point of view) that may come in handy some rainy day.†

Parameter Names

Throughout most of this manual we have treated a CL parameter as a simple thing, *e.g.*, as the analogue of a FORTRAN *variable*. Parameters are more complicated than this, however, as will be discussed under the next few headings. Type `help parameters` for more information.

Users have access to the parameters of all **loaded** packages and tasks. For example, to display the current right margin used by the HELP task, type: `= help.rmargin`. The most general form of a parameter reference is '*package.task.parameter*'. In addition to this unambiguous form of reference, a search path in the order *task* - *psets* - *package* - *cl* (psets are discussed below) is followed when an simple parameter name is used. For example, most packages have a *version* parameter; the command `= version` will print the version of the last package that was loaded (*i.e.*, the one whose prompt is shown).

Parameter Attributes

Parameters are more than simply names for memory locations, they have *attributes* that affect their behavior and that are directly addressable. Perhaps the most obvious attribute (after the *value*) is the *prompt*. To inspect the prompt for the CL parameter, *szprcache*, type: `= szprcache.p_prompt`. Some attributes are read only.

Parameter Sets

These are discussed more in *Named External Parameter Sets in the CL*. Basically, a *pset* is a type of task that is used to maintain a set of coherent parameters for some use. *Psets* are used quite effectively, for example, in the APPHOT package to group together logically associated parameters that are used by more than one task in the package. A *pset-task* is declared much like a CL script except that the filename extension of the referenced file is `.par` rather than `.cl`. The pset **datatype** allows you to define parameters that can pass around pset names.

Interactive Prompting

It often proves useful (perhaps too often) for a program to interactively ask the user questions. IRAF addresses this need through the parameter mechanism. In most cases, the best way to ask the user for information is NOT to explicitly print a question and then explicitly read the answer. Rather, a script (or a compiled task) should declare a query mode parameter whose prompt is the desired question. To ask a question, just use this parameter within an expression.

†A final disclaimer: a script that requires the extreme capabilities of the system to function is probably too complicated. Scripts are intended either for elegant (simple yet complete) solutions to general problems or for idiosyncratic solutions to specific (*i.e.*, one time) problems. Before writing a script, you should consider whether another solution (*e.g.*, an SPP or IMFORT program) is more appropriate.

Instead of:

```
begin
        bool    answer
        int     junk

        print ("Do you want to continue?")

        # Scan must be called as a function!
        junk = scan (answer)

        if (answer) {
             do something
        }
end
```

use:

```
bool    answer  = yes   {prompt="Do you want to continue?", mode="q"} †

begin
      if (answer) {
         do something
         }
end
```

This is simpler to implement and also allows for a default answer, in this case, `yes`. If you are debugging the script, the interactive prompt can be turned off simply be specifying the answer on the command line:

taskname [*arguments*] answer=yes

Note that the very nature of the parameter mechanism makes such interactive prompts less important. By concentrating the configuration details of a particular "program" (*i.e.*, **task**) into the parameter file for the task, the particular runtime details can be chosen and set by the user before running the task.

Environment

As in UNIX (and similarly to the VMS *logical name* mechanism), IRAF maintains an *environment* that provides a mechanism external to each task to manage the names (**printer**, **stdimage**, *etc.*) of system resources. The *path variables* discussed elsewhere in this manual are one application of the environment. To reference an environment variable, use the *envget* function: `s1 = envget ("printer")`. The *set* and *show* tasks are used to modify and inspect the value of environment variables. It is also possible to redirect the output of a *set* of a *show* command into a file for further processing.

Immediate Execution

The CL has the capabilities of a scientific calculator. The *immediate execution* command, '=', can be used to evaluate expressions of arbitrary complexity involving either the predefined CL "variables" (really parameters) or "variables" that are declared along the way. The *control flow* commands (**if-else**, **while**, **for**, *etc.*) that are used in scripts may also be used "on the fly". Several mathematical functions (`help mathfcns`) are available. This interactive use of the CL (see also §3) is also useful for previewing your scripts as they are developed.

†Here we chose to explicitly set the parameter to **query** mode. The same results can be obtained, without using the mode assignment, by including *answer* in the argument list of the procedure.

Numerical Formats

IRAF has builtin capabilities for handling a variety of numerical formats.

- Sexigesimal numbers such as `DD:MM:SS` of declination or `HH:MM:SS` of right ascension can be entered wherever a real number is expected; for example, `12:30:36` equals `12.51`.
- Octal integers can be indicated by appending a '`b`' or '`B`'; and hexidecimal by an '`x`' or '`X`'.† On output, the *radix* function will convert to any base: `= radix(7,2)` will convert decimal seven to base two.
- A more obscure capability is the automatic conversion of characters to their (ascii) values: `i = "Z"; = radix(i,10X)` will show that the ascii code (in hex) for the character `Z` is `5A`. Note that using `"Z"` as the argument to *radix* (or even the *int* function) will not work.
- The last curious capability involves adding strings and integers. Strings are usually joined by using the *concatenation* operator (`//`); they can also be joined by "adding" the strings: `s1 + s2` is the same as `s1 // s2`. An integer can be added on the right hand side of a string: `"junk" + 7` will make `"junk7"`. If the string already ends in a sequence of digits, the digits will be added as integers before the resulting strings are concatenated: `"junk7" + 4` will make `"junk11"`, **not** `"junk74"`. Two constraints are that the integer cannot be added on the left hand side of the string and negative numbers are not handled correctly.

  The example on page 9 in §4 can be improved by using this operation; the example uses this block of code to construct some image names, while handling the place-holding zeros properly:

```
if (i < 10)
    s1 = "data00"//i
else if (i < 100)
    s1 = "data0"//i
else
    s1 = "data"//i
```

  The variable *i* is incremented from 1 to 550 within a loop. These same image names can be constructed by replacing the block of code above by the single statement:

```
s1 = "data000" + i
```

Redefining Tasks

If you ever need to reissue a task definition statement, you should use **redefine** instead of **task**. The two accomplish the same thing, except that they have complementary warning messages: **task** complains if the task being defined already exists, **redefine** complains if the task does **not** exist. They will both succeed in either case.

Note that it is perfectly valid to have tasks with the same name in separate packages.

Hidden Tasks

All of the examples in this manual are of single, monolithic scripts, nowhere does one script reference another script as a subprogram. Nothing intrinsically prevents this, although it is generally thought to be counter to the goal of producing simple and easily maintained tasks. If you decide that you have a well defined application for internal sub-scripts the notion of a *hidden* task should prove useful. A hidden task is one that will not

†This is another reason not to begin filenames with digits; a name consisting of digits followed by a `B` or an `X` will be interpreted as its decimal equivalent - don't scoff, it's happened!

appear when a package menu is displayed with the '?' or '??' commands. Hidden tasks are typically given unlikely names, usually beginning with an underscore, '_'. The *hidetask* command (`help hidetask`) is used following the definition of a task to make the task hidden.

The Process Cache

Especially demanding scripts may require the management of IRAF tasks at the level of the host operating system process. This would usually involve locking a very frequently accessed task into the *process cache* (described below) if this task were being repetitively flushed from the cache for some reason. Conversely, a process might be explicitly flushed (if it is not currently needed) to free system resources.

The *process cache* is IRAF's way of limiting the overhead of process startup. IRAF groups the compiled code for several tasks into a single executable file, usually each package corresponds to one executable. When a task is run, the command language will check to see if the executable containing the task is already present in the cache, if so that process is reactivated, if not a new process is created which will be kept alive after the task finishes. There is a limit (typically 3 or 4) to the number of processes in the cache, when the limit is reached the process that has been unused the longest will be flushed from the cache. Processes may be locked into the cache to prevent them from being flushed, this should be used with discretion since it defeats the purpose of the cache and may lead to a lockout. Note that the **system** process is locked in by default.

To see what processes are currently in the cache:

```
cl> prcache
```

To lock an executable into the cache, give the name of a task that is in the executable:

```
cl> prcache dir
```

To flush the process cache:

```
cl> flprcache dir       (flushes dir whether it is locked in or not)
cl> flpr                (flushes all unlocked processes)
```

Foreign Task Interfacing

CL scripts serve as an ideal front end to existing host (non-IRAF) programs. Most users have compiled (pun intended) a large suite of useful programs that run outside the IRAF environment. The user interfaces for these programs are sometimes rudimentary and difficult to use either "standalone" or in conjunction with other programs. The IRAF *foreign task* facility (or a script) can often simplify the access to such programs, as well as allow free communication between the programs and IRAF.

The declaration of foreign tasks is discussed in §3.3 of *A User's Guide to Fortran Programming in IRAF, The IMFORT Interface*. Foreign tasks (as opposed to host OS *escapes* using the "bang" character, '!') may be abbreviated, have access to command line arguments, and participate in I/O redirection and pipes. They may **not** be used in the background and may **not** have parameter files.

Host level programs require that host filenames be supplied, not IRAF virtual filenames. There are two ways to translate the virtual filenames that users specify on the IRAF command line into host filenames that the programs will accept. The first is to use the foreign task *argument substitution* notation, as described in the *IMFORT* manual. The second is to use the CL function *osfn* which returns the host name for a given IRAF virtual name.

There are two basic paths to follow when interfacing an existing program. The first is to use the IMFORT interface to modify your program to directly access IRAF images and to read from the IRAF command line (and also the host command line). This will not be described further. The second path is to preserve the current program without modification, using scripts and foreign tasks (IMFORT programs are also accessed as foreign tasks) to bring the errant program within the fold. On some systems (*e.g.*, VMS) it may be necessary to write a host level command procedure (*i.e.*, another kind of script) to provide alternate access to the files or devices that are explicitly opened within the program. For instance a program, *scale*, might have a VMS command file, *scale.com*, that assigns a filename (supplied as an argument) to the user's terminal, which is directly accessed within the program:

```
$ assign/user 'P1' TT:
$ run scale
$ exit
```

The *scale* program is executed with its input taken from the specified file, rather than from the terminal. A task definition for *scale* might be:

```
task $scale = "$@usr0:[seaman.stars]scale.com $(1)"
```

The first dollar sign ($) indicates that the task (like all foreign tasks) has no parameter file. The second dollar sign indicates that this indeed is a foreign task. The third is part of the *argument substitution* notation along with the parentheses which cause the **host** pathname to the file to be used for the argument. A second program, *offset*, which does not require a command file, might be declared in either of these two ways:

```
task $offset = "$run usr0:[seaman.stars]offset.exe"
task $offset = "$offset:==\$usr0:[seaman.stars]offset.exe!offset"
```

Note how the final dollar sign is escaped to include it as part of the command to be executed (a *DCL* foreign command declaration). The *scale* task will be invoked with an argument that specifies a file containing the answers to all of the questions that are usually asked interactively. *Offset* will either be supplied with arguments, if the VMS level program requires arguments, or executed with no arguments, if all input is from specific files. The script that you write to manage these tasks is responsible for creating the correct files for each task.

Reading the Image Cursor

The version 2.8 release of IRAF has support for interactively reading the image cursor for Sun workstations and for the IIS hardware image display. This is implemented like the graphics cursor, through the use of a CL parameter, *imcur*. To read the image cursor, merely use the *imcur* parameter (or more generally, a parameter of the **imcur** datatype) in an expression. To create a *cursor loop* in a script, use the *fscan* function:

```
while (fscan (imcur, x, y, wcs, command) != EOF) {
    key = substr (command, 1, 1)
    if (key == something)
       do something
    else if (key == something else)
       do something else
    else if (key == "q")
        break
    else
        beep
}
```

Here x and y are real variables that receive the cursor coordinates, wcs is an integer that receives the *world coordinate system* (usually 1) and command is a struct (**not** a string) that receives the cursor key or colon command (including spaces, hence the struct datatype) that was typed to terminate the cursor read. There are no builtin *cursor mode* keystrokes such as the capital letter keystrokes for the graphics cursor.

Using Arrays

The use of arrays in the CL is covered in §6.7 of *A User's Introduction to the IRAF Command Language*. Arrays of **real**, **integer**, **bool**, and **string** datatypes can be declared. An array reference can be either to the entire array with implicit looping over the subscripts, to a single array element by indicating the subscripts (separated by commas) in square brackets ([]) following the array name, or to a subarray by using a syntax that is an analogue of the images notation. None of the examples in this manual uses arrays, principally because *listfiles* combined with the *fscan* function duplicate most of their functionality without requiring preset size limits.

Using Pipes

Some of the scripts in this manual have included commands that use the *piping* facility of the CL. This useful concept, borrowed from UNIX, has a few subtleties that deserve mention. First, this is an *IRAF* capability that is **independent** of the underlying operating system. An interesting result is that *foreign tasks* (discussed elsewhere) share the access to this plumbing that native IRAF tasks enjoy. Thus, a VMS command which knows nothing about pipes or I/O redirection can be used in a pipe within IRAF.

Unlike UNIX pipes, IRAF pipes operate sequentially: one task runs to completion before the next-in-line begins. Because of this, the **tee** command, which is a piece of plumbing that copies the STDIN both to a file and to the STDOUT, loses some of the usefulness of its UNIX analogue. For example, you might want to save the output of some task to a file and also see it at the terminal while the task executes, unfortunately you will only see the output after the task is entirely finished.

In *program* mode write pipes outside the parentheses, unlike I/O redirection.

Most IRAF tasks are not explicitly configured to read from the standard input (STDIN) and sometimes not to write to the standard output (STDOUT). This might seem to exclude them from use within pipes, but is actually quite easy to work around. Most tasks that expect a filename or a file template will automatically sense when the standard input has been redirected. A remaining difficulty is that you may need to specify a place holder for the task's query parameters. A NULL string ("") is usually sufficient for this purpose or the explicit string STDIN can be used. By using STDIN as an argument, a single task (*i.e.*, not a pipe) can also read its data directly from the terminal. To exit from this mode, type an EOF character, usually a ^Z or ^D. A little experimentation will tell you the correct calling sequence for a particular set of tasks. An example that creates a test image with random pixel values:

```
ut> urand nlines=64 ncols=64 | rtext "" test head- dim=64,64
ut> imhead test
test[64,64][real]:
ut> imstat test fields=mean,stddev,min,max verb-
    0.4996    0.2874  9.000E-4    0.9998
```

Redirecting the STDERR separately from the STDOUT

The IRAF CL provides a syntax either for redirecting the STDOUT to a file (>), or for redirecting both the STDOUT and the STDERR to a file (>&). To cause only the STDERR to be redirected to a particular file, the STDOUT must be separately "reredirected" on the command line. For example:

```
cl> head @inlist nlines=1 >& inlist.err > inlist.head
```

This will, in general, create two files: *inlist.err*, containing warning messages for those files, listed in *inlist*, that do not exist, and *inlist.head*, containing a labeled listing of the first lines of each of the files that do exist. Note that this scheme will also work with the *append* type of redirection (`>>&` and `>>`), but will not work if either or both of the redirections is replaced by a pipe. It **is** possible to pipe the STDERR together with the STDOUT (`|&`), but it **is not** possible to pipe the STDERR by itself.

Networking

This is a big topic that affects everything that you do with IRAF. We will only mention in passing that a task statement can point to a file on a different machine. The following command can be issued from any machine that has access to the node *orion*; IRAF networking will be used to read the script when the task is executed:

```
task center = orion!/u2/seaman/test/center.cl
```

Declaring your script tasks (compiled tasks will also work, but the machine architecture and the network bandwidth place constraints on this) in this way allows you to maintain just a single copy of your scripts on the network. Note that the node name is unnecessary if the scripts are kept on a disk that is accessible with host level networking from the various nodes.

The example script in Appendix III addresses the problem of network access of pixel files that were created on other nodes that share the same fileserver. Some other facilities that are also useful for this and for more general networking concerns:

- Nodes that share an identical set of mounted disks (typically a file-server and its diskless workstations) can be aliased in the file, *dev$hosts*. This must be done by your IRAF system manager.
- The special node name, **0** (zero), refers to the local node, no matter what machine you are using. By specifying your *imdir* with '`0!`' prepended, all pixel file access will be treated locally by IRAF.
- Just as your scripts can reside on a remote node, so may your pixel files. This "remote" node can be the fileserver, forcing IRAF networking to be used.
- Node names can be aliased in a user's *environment*. If **pyxis** and **phoenix** share the same fileserver, typing:

  ```
  set pyxis = phoenix   -   on phoenix, or
  set phoenix = pyxis   -   on pyxis
  ```

  will allow transparent access to images that were created on the other node. **Do not redefine the current node!**
- Lastly, the **HDR$** syntax can be used to cause your pixel files to be placed within the current directory or a subdirectory of the current directory:

  ```
  set imdir = HDR$pixels/
  ```

  This will cause the pixel files that are created to be placed in the subdirectory *pixels* of the directory that contains the image headers. The directory, *pixels*, will be created if needed. **The trailing slash (/) must be included in the declaration!**

All of these capabilities should be used with caution: the more complicated the scheme you use, the more likely you are to misplace your files.

Image Sections

This topic is discussed more fully in §4 of *A User's Introduction to the IRAF Command Language*, but a few comments seem pertinent. Users are often puzzled when they need to *flip* or *rotate* an image. They find the IMTRANSPOSE task in the images package which will transpose two dimensional images about their major diagonals, but the ROTATE task doesn't seem to be at all what they want: as a variant of GEOTRAN, it rotates and shifts by any angle. They fail to find a FLIP or IMFLIP task at all.

The solution to this mystery is that the *image section notation*, itself, allows images to be flipped at any point (*i.e.*, as the input to any task and as the output of some), in combination with IMTRANSPOSE, the section notation allows any right angle rotation:

`imtranspose test[-*,*] cw90`	-	rotate 90˚ clockwise
`imtranspose test[*,-*] ccw90`	-	rotate 90˚ counter-clockwise
`imcopy test[-*,-*] rot180`	-	rotate 180˚
`imcopy test[-*,*] vflip`	-	flip about the vertical (y) axis
`imcopy test[*,-*] hflip`	-	flip about the horizontal (x) axis

The section notation also allows the user to *subsample* an image "on the run". This can be used in situations in which the full resolution of an image is not needed and block averaging (using BLKAVG) is not desired. An example of a trick made possible by subsampling is this excerpt from a larger (and very specific) script:

```
# Construct a set of interleaved sections.  Nlines must be a multiple
# of four (in this case, 180) for this to work.  Sect[1] -> sect[4]
# are [*,1:177:2], [*,2:178:2], [*,3:179:2] & [*,4:180:2].
for (i=1; i <= 4; i+=1)
    sect[i] = "[*," // i // ":" // nlines+i-4 // ":2]"

# Each of the (integer) input images consists only of 1's and 0's.
# The output images are assumed to be different from the input images.
# IMARITH won't write into an output section, otherwise the IMCOPY and
# IMDELETE of the buffer image would be unnecessary.
for (i=2; i <= 3; i+=1) {
    imarith (input//sect[i-1], "max", input//sect[i+1], mask)
    imarith (input//sect[i], "min", mask, buffer)
    imcopy (buffer, output//sect[i], verbose-)
    imdelete (mask//","//buffer, verify-, >& "dev$null")
}
```

Can you guess what it does? The input images are solar maps that have been processed through IMREPLACE so that pixels within a certain range of data values are marked with 1's. This range of values is indicative of a coronal hole. All pixels outside this range (either above or below) are set to zero. The point of the script is to filter out all remaining 1's that have no extent in solar latitude (*i.e.*, from line to line), which are presumed to be spurious. The interleaved sections allow the filtering to be performed half an image (*i.e.*, every other line) at a time, rather than line by line or worse yet, pixel by pixel. This version of the script was about fifty times faster than a prior version which worked line by line. The rest of the script is concerned with looping over a set of images and with filtering the top and bottom lines which require special handling.

The filtering works because the IMARITH *max* operator behaves like the boolean *or*, and the IMARITH *min* operator like the boolean *and*, when the input pixels are binary numbers. Note that this is an extremely costly (and hokey) way to handle *mask* images (the input and output images are masks as well as the intermediate image). A new set of programming interfaces, described in the *IRAF Version 2.8 Revisions Summary*, allows far better manipulation of masks than the CL, although unfortunately no general applications exist yet.

Making Scripts with SECTIONS

The SECTIONS task (or any task that creates a list of images or files) can be used to make a quick and dirty script for a situation in which you want to apply a single task that does not handle an image template to a large number of images. This is especially handy for those users who are highly skilled with a text editor. Just redirect the output of SECTIONS into a file and globally edit the file to include the task name at the beginning of every line, and any parameters at the end of the line.

Making Scripts with HISTORY

The IRAF **history** mechanism is another route for quickly making scripts. If, after having executed several commands, you decide that you would like to re-execute the same commands (perhaps with other data), you can redirect a history listing into a file, edit the file, and use it as a script. For example, to produce a script containing commands from the last 50 that you issued:

```
cl> history 50 > history.cl
cl> edit history.cl
    ...
cl> cl < history.cl
```

STTY Scripts

CL scripts are not the only way to automate a series of IRAF interactions. *STTY* scripts are a way of remembering the exact sequence of keystrokes that are entered by a user and re-executing those keystrokes at some future time. Unlike CL scripts, which may require special *cursor* or *ukey* files (see §5) to handle tasks that use graphics or keystroke driven user interfaces, STTY scripts transparently remember those keystrokes as well as normal command lines. For help (as usual) type: `help stty`.

Why not use these instead of CL scripts? Since STTY scripts do remember every single keystroke (they form part of the *tty* or *terminal* driver, with very low level I/O access) it is difficult to make them general enough for everyday use. STTY scripts are also much more sensitive to changes in the system and are painful to edit after they are created, due to the high level of detail. Because of the nature of the STTY interface, the script will tend to take over the user's terminal, displaying every little keystroke, unless the output is redirected in some manner.

Although somewhat difficult to work with, **STTY scripts are ideal for making demos and for testing** software or machines. If you stopped by the IRAF display at the last few Winter AAS meetings to watch the demos, you have already seen STTY scripts at work.

Note that combining both CL scripts and STTY scripts can produce an entity more versatile than either. An STTY script can call any number of CL script tasks and a CL script, under certain circumstances, can call an STTY script (the STTY script may tend to terminate the script and the order in which the commands are executed may be unexpected). A package of STTY scripts (*e.g.*, demos) is also easy to make. Of course, a combination of CL and STTY scripts is also more fragile than either and a very slight change in the system is likely to break the thing completely.

Logfiles

The *logging* capability of the CL bears a surface resemblance to the creation of STTY scripts. When the CL parameter *keeplog* is set to *yes*, a logfile is kept by the CL. The filename is specified using the CL parameter *logfile*, by default *home$logfile.cl*. This is a much higher level facility than STTY scripting and the log will not include any interaction within a task, only the CL command lines that you issue. The *logmode* parameter controls the type of information that will be logged.

A logfile is useful, both for keeping a log of what you did during a data reduction session (many IRAF tasks will also keep logfiles), and also for quickly sketching in a sequence of commands. There is an overlap in functionality not only with STTY scripts, but also with the MKSCRIPT task, although, while MKSCRIPT allows the user to easily set the task parameters, but not to use tasks without parameters, a *logfile script* (to coin a phrase) will let you use any task, but parameter setting may be tedious.

One peculiarity of scripting with logfiles is that the command that is used to turn logging on, `keeplog = yes`, is itself logged. This can lead to an infinite loop the **second** time the *logfile script* is executed. As an example, first we create and review a script:

```
cl> keeplog = yes
cl> path
orion!/u2/seaman/
cl> dir
logfile.cl      login.cl        loginuser.cl    uparm
cl> keeplog = no
cl> type logfile.cl

# LOGIN Mon 09:38:18 14-Aug-89
keeplog = yes
path
dir
# Logout Mon 09:38:25 14-Aug-89
```

Next, we execute the new script:

```
cl> cl < logfile.cl
orion!/u2/seaman/
logfile.cl      login.cl        loginuser.cl    uparm
```

As a side effect, this command turned logging back on and recorded the command that executed the script (`cl < logfile.cl`). Repeating it starts an infinite loop:

```
cl> cl < logfile.cl
orion!/u2/seaman/
logfile.cl      login.cl        loginuser.cl    uparm
orion!/u2/seaman/
logfile.cl      login.cl        loginuser.cl    uparm
   ...
```

The solution is to edit the line, `keeplog = yes`, out of the script.

Executing IRAF Tasks Outside of IRAF

This topic is exactly counter to the rest of the manual. Version 2.8 of IRAF contains enhanced support for executing IRAF tasks from the host operating system; currently this support only applies to UNIX/IRAF. For some applications it may be more appropriate to adapt IRAF to work within the host OS, rather than to adapt host facilities to work within IRAF. These capabilities are described in the *IRAF Version 2.8 Revisions Summary*.