CL Programmer's Manual

Elwood Downey
Douglas Tody
George H. Jacoby

Kitt Peak National Observatory*
December 1982
(revised September 1983)

ABSTRACT

This document serves as a programmer's manual for the IRAF Command Language version 1.0. CL tasks, packages, parameter files, modes, expressions, statements, abbreviations, environment variables, command logging, error handling and directives are discussed. The special CL parameters are listed. A example of a complete CL callable program is given.

This manual is a programmer's guide, not a user's guide or a technical specification of the CL. Information about other programming tools within the IRAF system, such as the SPP language and compiler and the program interface, is given only to the extent required to introduce the examples.

NOTE: Somewhat out of date, but still useful.

*Kitt Peak National Observatory is operated by the Association of Universities for Research in Astronomy, Inc. under contract with the National Science Foundation.

Contents

1. **Introduction** 1

2. **Terminology** 1
   - 2.1. Physical and Logical Tasks, Scripts 2
   - 2.2. Packages 2

3. **Parameter Files** 2
   - 3.1. Location and Name of Parameter Files 3
   - 3.2. Parameter File Format 3
     - 3.2.1. name 3
     - 3.2.2. type 3
     - 3.2.3. mode 4
     - 3.2.4. value 5
     - 3.2.5. minimum and maximum 5
     - 3.2.6. prompt 5

4. **Modes** 6
   - 4.1. Determining Modes 6
   - 4.2. Setting and Changing Modes 6
   - 4.3. Recommended Mode Settings 7

5. **Expressions** 7
   - 5.1. Constants 7
   - 5.2. Parameter References 8
   - 5.3. Intrinsic Functions 8
   - 5.4. Operators 9

6. **Statements** 10
   - 6.1. Assignment Statement 10
   - 6.2. Commands 10
     - 6.2.1. Command Arguments 10
     - 6.2.2. Pipes and Redirections 11
   - 6.3. Immediate Statement 11
   - 6.4. Flow Control 11
   - 6.5. Abbreviations 12

7. **Environment** 12

8. **Log File** 12

9. **Error Handling** 13

10. **CL Initialization** 13

11. **CL Directives** 13
    - 11.1. **bye** 14
    - 11.2. **cache** *lt [, lt2, ...]* 14
    - 11.3. **cl** 14
    - 11.4. **keep** 14
    - 11.5. **lparam** *lt [, lt2, ...]* 14
    - 11.6. **package** *packname* 14

11.7. **redefine** *[lt1, lt2, ...] lt = pt* 15
11.8. **set** *[name = value]* 15
11.9. **task** *[lt1, lt2, ...] lt = pt* 15
11.10. **update** *lt [, lt2, ...]* 15
11.11. **version** 15
11.12. **?** and **??** 15

12. **CL Parameters** 15

13. **An Example** 16

CL Programmer's Manual

Elwood Downey
Douglas Tody
George H. Jacoby

Kitt Peak National Observatory*
December 1982
(revised September 1983)

1. Introduction

The Command Language, or **CL**, serves as a command and runtime supportive interface between the user at his computer terminal and the application programs he is executing. The user types his commands to the CL and it does whatever task and file manipulations are necessary to carry out the commands.

The user and the applications task do not communicate directly; they communicate only through the CL. Once started, a task requests parameters by name from the CL and the CL responds with the value of the parameter. To get that value, the CL may have had to read a parameter file, query the user, do range checking, extract a value from a command line or perform other actions.

All CL/task communications take place via an interprocess communications link between the CL process and the process containing the applications task. Standard input, output, error, and plotting channels are multiplexed on this link and managed by the CL. The CL process and the applications package process execute concurrently.

This arrangement relieves each new application program from having to provide user interface functions that are often rewritten directly each time, such as command line parsing, command and parameter abbreviations, and levels of interaction to accommodate both novice and experienced users. In addition, the CL provides a common environment for running all tasks with services such as executing programs with their input and output redirected to files or to other programs, managing parameters for each command, handling lists of values in lieu of a simple parameter, logical device and file name assignments, and help facilities. The CL is a simple programming language in its own right, with conditional and repetitive command execution, parameter expressions and a calculator.

While intended to support scientific reduction and analysis applications at Kitt Peak and elsewhere, the CL can serve any project that involves running programs as commands with arguments. Every effort has been made to make the CL as portable as possible. The link between the CL and the task it is running is character oriented and allows the task to to be run directly without any support from the CL if desired. This link may be simulated by any program that wants to run as a task under the control of the CL. However, any task written in the SPP language (which is Fortran based) will automatically include all the i/o facilities required to interface to the CL.

2. Terminology

This section defines most of the terminology associated with the CL. Words in **boldface** are part of the actual terminology of the CL. Those in *italics* are more descriptive in nature and serve only to name a representative item.

---

*Kitt Peak National Observatory is operated by the Association of Universities for Research in Astronomy, Inc. under contract with the National Science Foundation.

2.1. Physical and Logical Tasks, Scripts

A task runnable under the CL is a file containing either the executable program itself, or a text file containing a **script** written in the CL language. Either of these is referred to as a **physical task** since they are true files on the host computer. The executable form consists of one or more **logical tasks** but the script file is always considered exactly one logical task. The general terms **command** and **program** refer to one of these logical tasks. In order to know just how to go about running the command, the CL has a "task" declaration that indicates in which file the task resides, and whether it is in an executable or script form. Once declared, the logical task commands are used the same way regardless of whether they reside in an executable object file or a script.

In order to manage itself, there are a few commands that the CL does itself, such as the "task" command mentioned above. These built-in commands are referred to as CL **directives**, but only as a means of classifying them as a group. They act and are used very much like regular commands. In this way, they have the same syntax rules and their diagnostics work in the same fashion. Since they are built into the CL program itself to achieve intimate knowledge of its internal data structures or simply to increase efficiency, the set of directives is not extensible by the user.

Thus, there are a variety of ways a command may get executed. It is no accident that there is often no easy way to tell how a command is implemented.

2.2. Packages

Ostensibly, tasks are grouped into packages. This provides a logical framework to organize a large body of commands in a large system and also serves to address the problem of redefinitions. The CL directives and a few utility programs are located in the root package, called **clpackage**, and is always present when the CL starts up. Some of the commands in the root define more packages when run. They are script tasks that define a package and some tasks in that package. By convention, the name of the package defined by a script, the logical task and the script physical task file name are all the same.

Any package defined to the CL may become the **current package**. The prompt issued by the CL includes the first two characters of the current package. When a command is typed to the CL, it looks in the tasks of the current package first, then through all tasks in lower packages towards the root clpackage for the logical task with the given name. Tasks defined in packages defined farther away from the root are searched last. This **circular search path** provides some measure of control over command scope.

Wherever a task name is expected in the CL syntax, the package may be explicitly specified in the form *package.task* so that only tasks defined in that specific package will be considered in the search for the given logical task name. This form allows package names farther away from the root than the current package to be accessed. It also provides an unambiguous way to reference a task when the task name appears in more than one loaded package. If the name of a loaded package itself is given as a command, then it simply becomes the current package (see the package directive in §12).

3. Parameter Files

A separate file, the **parameter file**, may exist for each logical task. It contains a description of each of the parameters used by the task that should be known and managed by the CL. (These are not the same as variables declared in the source program for the task.) The parameter files are the permanent record of task parameters. When a parameter value is permanently changed, as with an assignment or when in learn mode, the CL makes a local copy of the parameter file with the new value. Thus, running tasks imply CL reads and writes to parameter files as well as execution of the task.

A logical task need not have a parameter file. If a task makes a request to the CL for a parameter and the CL knows the task has no parameter file a fake query for the the parameter will be issued by name (see §4 for more on queries). All of the range, prompt, learning and type checking advantages of real parameters will be lost, however. Thus, a parameter reference by a task that does not have a parameter file at all is not considered an error. This is different than a reference to a nonexistent parameter by a task that does have a parameter file, which is an error.

3.1. Location and Name of Parameter Files

The parameter file for a logical task may be in two places. The CL first searches the **uparm** directory, then the directory containing the physical task. All physical tasks for a package, including the script task that defines it, are usually in one directory, often referred to as the **package directory**.

Uparm is an environmental entry used by the CL when accessing parameter files. If it does not exist, the current directory is used. Uparm may either be another environmental reference to a directory or be in host-dependent format (see environment, §8).

The names of parameter files written out, either to uparm or to the current directory, are formed by concatenating the first two and final characters of the package name, an underscore, the name of the logical task, and the extension ".par". For example, when the parameter file for a task *txyz* in package *pxyz* is written, it is named *pxz_txyz.par*. The package prefix is prepended to avoid file name conflicts if two tasks in different packages happen to have the same name. Since local copies have the package prefix, the CL looks for them before ones without the package prefix.

3.2. Parameter File Format

The parameter file for a logical task consists of comments, blank lines, and parameter declarations. These may appear in any desired order. Comment lines are those that begin with the sharp character, #, and signal that it and all remaining characters on that line should be ignored. The maximum line length is 132 characters.

Parameter declarations within the parameter file take the form

name, type, mode, value, minimum, maximum, prompt

where all fields from value on are optional. The comma and the end of the line itself both serve as a field delimiter and thus a comma is not necessary after the last field, whatever it is.

3.2.1. name

This is the name of the parameter. There is no length limit other than the overall line length limit consideration. This is the name by which the parameter will be known to the task and to the CL. It must begin with a letter or a dollar sign, $, but the remaining characters may be any combination of letters, numbers, underscore, _, and dollar, $. Casual use of $ is not recommended, however, as it is used to make environment references (see §8).

3.2.2. type

The type field indicates how the parameter is to be stored. It also implies some information about what values are acceptable and how they are entered, as discussed below under value.

code	*meaning*
b	boolean
i	integer
r	real
s	string
f or f*xx*	file name
struct	structure
gcur	graphics cursor
imcur	image cursor

The codes **b**, **i** and **r** indicate the usual boolean, integer and real types. They are discussed further in the value section, below.

There are several types that manipulate character strings. The characters themselves may be anything from the ASCII set. The type **s** is the simplest and is an ordinary character string. It is typically used for names, flags and messages.

The **f** type is like s except that it is limited to legal file names on the host operating system, after possible environment substitution. The f may optionally be followed by any reasonable combination of the characters **e**, **n**, **r**, or **w**. These indicate that checks should be made of the file name before it is used that it exists, does not exist, that it exists and is readable and that it exists and is writable, respectively. **Struct** is also like s but its value is the entire next line of the parameter file.

Gcur and **imcur** are similar to struct but are expected to be of the form "x y char" to be usable as cursor coordinates. A gcur or imcur parameter will always read from the hardware graphics or image display cursor if it is in query mode.

If the type is preceded by a star, *, the parameter is **list-structured**. When the parameter is referenced, the value will come from a file, the name of which is the fourth field of the parameter declaration. All of the basic types may be list-structured.

3.2.3. mode

This field indicates what actions are performed when the parameter is referenced or assigned. The topic of modes is important to the CL and is covered more thoroughly elsewhere (§4) Briefly, query mode generally causes the user to be queried each time the parameter is referenced. Learn means that all changes to the parameter will be permanent. Auto mode means that the effective mode of the parameter should be whatever the mode is of the task that is using the parameter; auto mode defers mode selection to the task, or CL level. Hidden means that the existence of the parameter will not be evident to the user unless its value is not acceptable.

The mode field may be any reasonable combination of query, learn, auto and hidden. These may be spelled out and separated with plus signs, +, or abbreviated to one character and run together. For example,

```
...,auto+learn,...
```

and

```
...,al,...
```

and equivalent.

3.2.4. value

This field is optional. The value field is the initial or **default** value of the parameter. It has various characteristics depending on the type of the parameter. If it is absent, the parameter will be marked as undefined and will cause an error if used in an expression. A special entry, **indef**, is allowed that marks the parameter value as being indefinite, but not undefined. It may be used with all types. Acceptable constants in the value field are like those allowed by the CL in expressions (see §5.1).

For boolean parameters, it should be either the three characters **yes** or the two characters **no**.

Integer and real parameters are as one would expect. Real constants need not include a decimal point, ., if not required.

For string and file name parameters, the field extends from the comma following the mode field to the next comma, or the end of the line if none. It may be surrounded by single or double quotes, ’ or ", but these are not necessary unless the string is to include a comma. The length of the storage allocated for the string will be the minimum of 30 characters and the length of the initial value, up to a maximum of 132. Later changes to the value of the string will be silently truncated to the amount thus established.

Structs and the cursor types use the value field to indicate the number of characters of storage to allocate to hold the value of the parameter. The value is a string consisting of the entire next line of the parameter file. If no number is given in the value field, then just enough storage to hold the next line will be allocated. If the number is larger, this allows the value to grow longer than the length of the next line. Since dynamic string storage is not used in the CL, the length of all strings is fixed and using the value field in this way permits a short initial value but allows for later growth. The length of string storage is limited to 132 characters. It is an error to explicitly specify a storage length shorter than the initial value.

The value field for list-structured parameters is the name of the file containing values for the parameter. This name is subject to the same restrictions as a parameter of type fr and environmental references are allowed.

Thus, the value field entry for a parameter in a parameter file has several different uses, depending on the type of the parameter. The term **value** refers to that which is used when the parameter is used in an expression and **value field** refers specifically to the fourth field of the parameter specification. Because of this multiple usage, the CL recognizes this field with several names, as described under parameter references (§5.2).

3.2.5. **minimum** and **maximum**

These two fields work together to specify a validity range for the value of the parameter. They are ignored for all types except integer, real, and file name parameters and follow the same rules as the value field for these type parameters. Their application to filenames is to test for a simple lexical ordering. If they are both set when the parameter is referenced, then a query will be generated if the value of the parameter is not within range. No range checking is done if either the minimum or maximum are undefined or if min > max. If the parameter is list-structured, then the range checking is applied to the entry read from the file.

3.2.6. prompt

This field behaves like a string and extends from just after the sixth comma in the parameter spec to the end of the line. It may be quoted. As explained more thoroughly under query mode, its purpose is to provide a meaningful prompt for the parameter during a query. If no prompt string is given, then the query will just use the name of the parameter. As with strings, the length of the prompt implies the amount of static storage to allocate; later changes to the the prompt will be silently limited to this length.

4. Modes

The CL supports three modes of operation, query, learn and auto.

Query mode is the most interactive and is the standard mode when the CL is being used interactively. It causes each parameter referenced by a task, or script, to produce a query on the terminal consisting of the prompt string for that parameter, its current value and minimum and maximum values, if set. If there is no prompt string, then the name of the parameter is used. When the user sees this query, he may type a simple return to accept the current value or type a new value. New values that are entered in this way are checked for validity immediately with regard to type and range, and the query repeats until a reasonable value is entered.

A query will be generated regardless of the effective mode of the parameter if it does not meet its range requirements. On the other hand, a query will be prevented if the parameter was set on the command line, again assuming it is not out of range. Thus, the CL relieves the application program from some of the burden of verifying its parameters.

Learn mode retains the values of parameters across task runs and even across CL sessions. The default values of parameters come from their entries in the task's parameter file. If learn mode is not in effect, changes to parameter values by way of command line arguments to the task or queries do not cause the parameter file to be updated and so the values revert back to their defaults as soon as the task ends. Learn mode makes these changes permanent by updating the parameter file for the task.

Hidden mode applies only to parameters. It prevents queries from being generated even if the effective mode for the parameter is query, unless its value is out of range. Hidden mode also prevents the default value from ever being "learned". The only way to change the default value of a hidden parameter is by an assignment statement. Hidden mode is useful for parameters that are rarely if ever changed to hide their existence from all but experienced users.

4.1. Determining Modes

The modes exist independently in a three level hierarchy: the parameter, the current task, and the CL itself. Whenever a parameter is referenced, its **effective mode** is calculated. To determine the effective mode, the mode settings of the three levels are used starting with the parameter level. If the mode of the parameter is query or learn, that is the effective mode. If the parameter's mode is **auto**, then the effective mode is that of the current task unless it too is in auto mode in which case the effective mode is that of the CL. If all levels are auto, the effective mode is auto and neither query nor learn effects will occur.

Thus, each layer of the hierarchy, starting at the parameter level, defers to a higher level until it finds either query or learn (or both). Note that the presence of hidden mode at the parameter does not alter this process but rather serves to override query mode, should it be found at any given level. As a practical example, all the auto-mode parameters in a task can effectively be put into query mode at once by setting the mode once at the task level to query.

4.2. Setting and Changing Modes

The modes themselves are set in different ways at the parameter and task level. The mode for a particular parameter is accessed as a field of that parameter called **p_mode**. It may be abbreviated. The mode of a task is in a parameter **mode**, of type string, that contains any reasonable combination of the letters **q**, **l**, **a** and **h**. This parameter may be declared and initialized as desired in the parameter file for the task just like any other parameter. If it does not appear in the parameter file for a task when it runs, it will be manufactured and supplied with a default setting of 'ql'. This is the only case of a parameter added by the CL to a parameter list for a task. One of the parameters to the CL itself is also **mode**, and this serves as the mode of the CL, the highest level in the mode hierarchy.

As a convenience for naming modes, four CL string parameters **query**, **learn**, **auto** and **hidden** are defined to be the single-character strings ’q’, ’l’, ’a’ and ’h’. Examples of setting modes at the CL, task, and parameter levels:

```
mode = ’ql’                          # set CL mode to query, learn
package.task.mode = ’a’              # set given task mode to auto
package.task.param.p_mode = ’ql’     # set given parameter’s mode
mode = query + learn                 # use pre-defined string params
mode += query                        # add query
```

The mode of a parameter may also be changed during a query for that parameter. If the response to the query begins with a percent, %, then the mode for the parameter may be set using the same format as that used in the parameter file mode field (see § 3.2). This is useful during program development for making a parameter hidden once its default value has been determined.

4.3. Recommended Mode Settings

The recommended default modes are auto and learn for the CL itself, query for each task and auto or hidden for the parameters. Auto mode for all non-hidden parameters in a task allows them all to be changed at once by changing the mode of the task. The user will rarely do more than change a task’s mode to auto, hide a parameter (by use of the %h response to a query, §4.2), or reset all parameters of a task to their original default by deleting its parameter file from the uparm directory (see §3.1).

5. Expressions

The CL allows expressions wherever a simple variable might appear. This applies only to the language, however, not, for example, in the parameter files. Expressions are the usual kinds of combinations of constants, variables, intrinsic functions, operators, parentheses and expressions (recursively).

5.1. Constants

Boolean constants are entered as the three characters "yes" or the two characters "no". There are no true and false constants.

Integers are an uninterrupted sequence of digits; a trailing ‘b’ denotes an octal constant.

Floating point constants are as in most languages but a decimal point is not necessary if not needed. 5, 5., 5e0, .5e1 and 5.e0 are all equivalent. Sexagesimal notation may also be used to create a floating point value. A negative value is indicated by a leading minus sign, -, leading zeros are not necessary and the seconds field is optional. 1:23:4.56, -12:3:4.5, 1:2:3 and -12:34 are all acceptable.

Strings are zero or more characters surrounded by single or double quotes, ’ or ". The quotes are not needed in two cases. One is in response to a query. In that case, everything up to the end of the typed line is taken to be the string. If the quotes are used, however, they will be discarded. The other case is when specifying the value of a parameter on the command line when running a task. If the corresponding parameter is of type string, filename or is list-structured and the string need not be used in an expression, then the quotes are optional.

An additional constant, **indef**, is known to the CL. This is a special setting that means indefinite, as opposed to being truly undefined. The latter causes an abortive error if encounted during the evaluation of an expression. A parameter that is merely indefinite does not result in an error or a query and is useful for indicating the value should be ignored, but propagated through an expression.

See the discussion of the intrinsic scan function (§ 5.3) for two additional constants, EOF and stdin.

5.2. Parameter References

The "variables" in CL expressions are task parameters. To reference a parameter, the most general form is *package.task.param.field.* This form may be used anywhere a parameter is legal. Only the parameter name portion is required. If the package and task are not specified, the parameters for the current task, then the current package and finally those of the CL itself are searched. The parameter is not found if it does not exist in one of these three places.

If the field is not specified, then the meaningful value of the parameter is used, as explained under the discussion for the value field of a parameter (see §3.2). The possible fields are p_name, p_type, p_mode, p_value, p_minimum, p_maximum and p_prompt. In addition, the value field may also be given as p_length, p_default or p_filename. These are intended for use with parameters of type struct or cursor, integer or real, or filename (or list-structured). These aliases are not strictly enforced but are provided to improve readability and reliability in CL commands, particularly within script tasks. Each portion of the parameter reference may be abbreviated separately (see §7).

The result of using a logical operator is either the boolean true or false. These values are represented internally as 1 and 0, respectively. Although it is bad programming practice to make use of that fact in further arithmetic operations, it is not prohibited.

5.3. Intrinsic Functions

The CL provides a set of standard intrinsic functions that may be used in expressions. They are much like those found in most math libraries and are listed here only for reference. As with commands, they may be abbreviated but unlike commands their arguments must be enclosed in parentheses. Calling them with illegal arguments or producing underflow or overflow generates an error. Their argument(s) may be integer or real and they will try to return the same type as their argument if no loss of precision would result.

Usage	Number of Arguments	Description
abs(x)	1	absolute value
atan2(y,x)	2	arc tangent, with proper quadrant
cos(x)	1	cosine
exp(x)	1	natural exponentiation
frac(x)	1	fractional part
int(x)	1	integral part
log(x)	1	natural logarithm
log10(x)	1	common logarithm
max(x1,x2...)	> 1	maximum
min(x1,x2...)	> 1	minimum
mod(x,modulo)	2	first arg modulus the second
round(x)	1	nearest integer, rounded away from zero
scan(l,p...)	> 1	free-format read; see below
sin(x)	1	sine
sqrt(x)	1	square root
tan(x)	1	tangent

The **scan** intrinsic function reads from its first argument as a string and assigns the pieces, suitably type cast, into the remaining arguments. If the first argument is a list-structured parameter, the next line of the file is read and scanned, unless query mode is in effect in which case the user is always prompted for the line. If the first argument is a string-type parameter,

including filename, struct, gcur or imcur, then the string is scanned. This serves as an in-core read, much like a Fortran decode or a C sscanf function. Spaces, tabs and commas are recognized delimiters. If the last target parameter is a string, it will receive the remainder of the string being scanned.

Scan returns as its function value the number of successful conversions. Reading from a list and encountering eof will cause scan to return a count of zero. There is a pre-defined constant in the CL, **EOF**, which is simply zero; it may be used to make the test more explicit. There is another CL constant, **stdin**, which may be used as the first argument to cause scan to read from the standard input. Examples of scan are

```
# Read gcur and print radii until end of list.
while (scan (gcur, x, y, remainder) >= 2)
    = sqrt (x**2 + y**2)

# Read until EOF is detected.
while (scan (file, line) != EOF)
    = line
```

5.4. Operators

The following is a list of the arithmetic and logical operators available in the CL. They are the same as in the SPP language.

Operator(s)	Type of Result	Function
+, –, *, /	numeric	the usual, but see below for + with strings
**	numeric	raise to power
%	numeric	first expression modulus the second; like mod()
<, >	logical	less than, greater than
<=, >=	logical	less than or equal, greater than or equal
==, !=	logical	equal, not equal
&&	logical	logical ‘and’
||	logical	logical ‘or’
!	logical	logical ‘not’

For those familiar with C, note the absence of =. It is not considered an operator that produces an l-value but may only be used in an assignment statement.

The + operator can be used to concatenate strings. If only one of its operands are strings, the other will be converted first. If one operand is a string, the other is an integer and the string operand contains an integer on the same side as the integer operand, then an arithmetic addition will be performed as well. For example,

’stringa’ + ’stringb’	→	’stringastringb’
’string1’ + ’string2’	→	’string1string2’
’string1’ + 2	→	’string3’
2 + ’string1’	→	’2string1’
2 + ’9string’	→	’11string’
’string’ + boolean_param	→	’stringyes’ (or ’stringno’)

Points, ., in strings with digits are not recognized as floatings so trying to add floatings to strings, while not prohibited, probably doesn’t do anything useful.

6. Statements

Statements fall into the following categories: assignments, commands, immediate and flow control. These will are discussed separately, below.

Statements may be delimited by newline or semicolon, ;, and may be grouped with brackets, { and }. Nesting is supported. Comments begin with the sharp character, #, which indicates that all characters from it to the end of the line are to be ignored. Statements that are too long to fit on a line may be continued by ending the line with a backslash, \, or they are automatically continued if the last character is a comma.

When used from a terminal, the CL issues a continuation prompt, >>>, when the outermost statement has not been completed. This indicates input is still being accepted and parsed. No work will actually be done until the CL sees a complete input statement.

6.1. Assignment Statement

An **assignment** is a statement of the form *parameter = expression*. The parameter is always permanently changed by an assignment statement, whether or not learn mode is in effect.

Two additional forms of assignments are provided that also perform arithmetic, *param += exp* and *param −= exp*. These are equivalent to *param = param + exp* and *param = param − exp*. They are more efficient as well as more convenient. These forms also permanently change the parameter.

All forms of the assignment statements will cause an error if the result of *exp* is undefined. Thus, the CL will never allow a parameter to be set to an undefined state. The only way to get an undefined parameter is by not setting it in a parameter file (see the value discussion in §3.2). Assignment statements are the only way a hidden parameter may be permanently changed.

6.2. Commands

A **command** is the basic means of running logical tasks. It consists of the name of the logical task, possibly with arguments, and pipes to more commands or io redirections. The arguments to the command, if any, may optionally be surrounded by parentheses. These are recommended in scripts. Command lines may be continued on the next line automatically if they end with a comma or a backslash.

6.2.1. Command Arguments

The arguments to a command are given as a comma-separated list and come in two basic forms, positional and absolute. The **positional** form is any general expression. The expressions will be evaluated and assigned one-to-one to the corresponding parameters of the task, as defined by their order in the task's parameter file, not counting hidden parameters. Only the value of the parameter may be set in this manner. A lone comma may be used as a placeholder and skips a parameter without changing it. Parameters not reached in the matching are also not changed.

The **absolute** form is an assignment, *parameter = expression*, where the parameter must be a parameter of the task being run. This is useful when a parameter value is to be changed but its position in the argument list is not known or it would be awkward to arrive at its position by a large number of positional arguments. Since the parameter is explicitly named, fields other than the default value may be changed with the absolute form.

Another form of absolute argument is the **switch**. It is a shorthand way of specifying the truth value of a boolean parameter. A switch consists of the parameter followed by a plus, +, to set it to yes, or a minus, −, to set it to no. Thus, these two forms are equivalent ways of turning off the boolean parameter *option*:

```
task option=no
task option–
```

While they may be used together, all positional arguments must precede absolute arguments. Here are examples of using the positional and absolute forms together: (note the parens in the second example are optional)

```
task1 x, task2.param, op+
task3 (a, b, c, param2=x+y, op3–, param3=task4.x/zzz)
task4 x, y, z, op1+, op2=yes
```

Parameters changed on the command line will have their new values as long as the command is executing. If learn mode is not in effect for the parameters, they will revert back to their original values when the task ends or if the task aborts for some reason.

6.2.2. Pipes and Redirections

A **pipe** connects the standard output of one task to the standard input of another task. A pipe is indicated by separating the tasks with a vertical bar, |. As many pipes in a series may be used as necessary. **Redirections** of the standard input and output of a task from or to files are also supported.

The standard input may come from a file by indicating the filename after the less-than symbol, <, and the standard output from the last task in a pipe sequence may be sent to a file by giving its name after the greater-than symbol, >. Two greater-thans, >>, cause the output to be concatenated to the end of the file. If the output redirection symbol is preceded by an ampersand, &, then the standard error will also be included, as in &|, &> and &>>. Output redirections, but not pipes, are considered absolute arguments to the task so they must follow any positional arguments and must be set off by commas. For example, task1 reading from file t1input piped to task2 writing to file t2output is done as

```
task1 x,y,z, < t1input | task2 x2, y2=a+b, > t2output
```

6.3. Immediate Statement

This is the **calculator** mode of the CL. It consists of the basic assignment statement without the left-hand side parameter, as in "= *exp*". Instead of computing the expression and assigning it to a parameter, the result is simply sent to the standard output. This may in turn be redirected if the calculation is being done from a script.

6.4. Flow Control

The CL provides **if-then-else** and **while** program flow control constructs. These look like

```
    if (expr)
        statement
    else
        statement
and
    while (expr)
        statement
```

This is quite general since the "statement" may be a group of statements in brackets. Also, since if-then-else is itself a statement, they may be chained into if-then-else-if- and so on. The else clause is optional.

6.5. Abbreviations

If the boolean CL parameter **abbreviations** is yes, then packages, commands, intrinsic functions and parameters may be abbreviated. The scope of the abbreviation is limited by its context. For example, if a parameter reference is *task.param*, the only candidates for the param abbreviation are those parameters belonging to the given task; similarly for parameter names given in the absolute form of a task's argument list. Parameter fields, such as p_name and so on, are always considered within their own class so their briefest forms are always p_n, p_t, p_mo, p_v, p_l, p_d, p_f, p_mi, p_ma and p_p (see §5.2). The intrinsic functions are also in their own class.

Abbreviations are not allowed in scripts. They are intended only to streamline interactive work with the CL.

7. Environment

The **set** CL directive, as explained elsewhere (§12), provides a simple string substitution mechanism for filename translations. Most operating systems allow a logical assignment to a physical device name for use in filenames. The CL trys to merge this with its own environment table so that definitions in the host system are available within the CL in addition to new entries added by the CL. Typical uses for the translations are portable names for system-dependent directories and io devices, such as tape.

The CL keeps its environment table in a last-in first-out fashion. New entries hide but do not overwrite old entries. Substitutions take place in strings being used as file names in commands and in parameter files. This includes list-structured parameters and io redirection. Environment references are indicated by following them with a dollar, $. For example, if the following environment definition is made:

```
set mydir = '/usr/myname/dir/'
```

then these uses

```
task x, y, z, > mydir$file1
task2 filename = mydir$file2
```

become

```
task x, y, z, > /usr/myname/dir/file1
task2 filename = /usr/myname/dir/file2
```

Note that the quotes around the value for mydir are necessary since the slashes are not legal in identifiers.

The environment facility is strictly a string substitution mechanism. Directory names and other uses must be complete enough so that a valid filename is the direct result of the substitution; the environment facility has no knowledge of file naming requirements on the host system whatsoever.

8. Log File

If the boolean CL parameter **keeplog** is yes, then each command typed by an interactive CL will be entered into a log. Commands that come to the CL from tasks or scripts are not kept. The name of the file is in the filename CL parameter **logfile**. This parameter is only used when logging is started. To change the name of the logging file after logging has already begun, set keeplog to no, change the value of logfile, then restart logging by setting keeplog to yes. Each time logging starts, the current time is entered in the log file as a CL comment.

9. Error Handling

From the start, the single most important requirement of the CL was that it properly handle error conditions. As one veteran put it, "the error case is the normal case, and the case when the program runs perfectly is the abnormal case".*

To most easily explain error recovery in the CL, the discussion diverges for a moment to explain a bit of its internal structure. Each new logical task that is run pushes a data structure onto a control stack. This structure indicates, among other things, where the standard input and output for the task are connected and process control information. As each task dies, its control structure gets popped off and the exposed task resumes as the active one.

When a task encounters an error, it issues a diagnostic to its standard error and informs the CL. The CL then repeatedly pops tasks, killing them as necessary, until it uncovers one that had its input and output connected to the terminal. Thus, an error condition forces a return to an interactive task, most likely an instance of the cl directive.

As each task is popped, its name and the parameters that were set on the command line when it was run are given as a kind of "stack trace" to aid diagnosis. Parameter files of tasks that abort due to their own errors or because they got killed on the way to restoring an interactive state are not updated. The environment, package and task definitions, and all other extensible data structures, are restored to their state at the time the resumed task was pushed.

The diagnostics from the CL all begin with "ERROR:". This always means that the full abortive procedure outlined above has occurred. If an internal consistency check fails, this becomes "INTERNAL ERROR:". A few diagnostics begin with "WARNING:". Warnings do not invoke the abortive procedure but are merely informative messages generated during command processing.

Perhaps the least helpful error messages are "syntax error" and "parser gagged". These are generated by the parser when it has no idea of what it is trying to crack or when it gets terribly confused. The only advice, until the improved parser of CL2 is available, is to carefully inspect the offending statement. If the error occurs during the interpretation of a script, an approximate line number is also given.

10. CL Initialization

When the CL starts up, it tries to read two CL script files. The first is in an IRAF system-wide directory and is called **clpackage.cl**. It defines the tasks in the root package clpackage, makes useful environment entries and does other chores such as printing news. The other is called **login.cl** and will be run if found in the current directory from which the CL is started. This serves as a way to personalize the CL on a per-user basis. Typical uses are to set modes, options and uparm, define personal tasks and packages and make environment entries for frequently used directories. Note that login.cl is run as a genuine script and any changes it makes to the dictionary after doing a "keep" will be lost.

11. CL Directives

The following commands are handled directly by the CL. They are always available in the root package, clpackage. They behave as all other commands in that they may be abbreviated and may have their input and output redirected as desired. Arguments in square brackets, [and], are optional.

---

* *Writing Interactive Compilers and Interpreters*, P.J. Brown, page 55.

11.1. bye

Exit the current task and resume the previous one where it left off. If there is no previous task, then the CL ends. Any task declarations, cached parameter files and environment definitions not kept (see keep) will be discarded from core. The same effect may be achieved by typing EOF (control-z on DEC systems). If used in a script task, it causes the script to abruptly end as though the end of the script file had been encountered. Since most packages are defined in scripts that do the cl directive, bye often has the effect of exiting from an entire package (see cl).

11.2. cache *lt [, lt2, ...]*

Read the parameter file for each given logical task(s) into the dictionary. They will remain in core until the current task exits (see bye). This is useful before running a task repetitively to reduce the file i/o required to bring in and possibly update the task's parameter file each time it runs.

11.3. cl

Run the cl itself as a task. This is generally useful in script tasks to stop the script midstream and allow terminal interaction again. The script might start with a package declaration, make some set and task declarations then do "cl()". This would cause the cl to run as a subtask to the script task and allow user interaction, with the new package and tasks. When the cl sees bye or EOF, the script task resumes, doing whatever cleanup it desires and exits, taking the new package, tasks and other dictionary changes with it. Other uses of the cl directive are to run script tasks. Since its input can be redirected, as with any other task, "cl < file" is a way to run a script file. Note: just where the cl gets its input when run without arguments is still being discussed but the above description, as far as it goes, should not change.

11.4. keep

Cause all currently defined tasks and packages and any cached parameter files to remain in core when the current task ends. Normally, all dictionary space used by a task is discarded when the task ends. If any further dictionary changes are made, they will be discarded as keep only retains what was defined at the instant it is used. Keep only effects the current task. When the task from which the current task was called ends, the kept dictionary space will be discarded unless keep was called in the prior task as well.

11.5. lparam *lt [, lt2, ...]*

List all parameters for the given logical task(s), if any. The name, current value, and prompt string is given for each, one per line. The parameters are given in the order in which they should be used as positional arguments in commands. Hidden parameters are listed at the end, surrounded by parentheses.

11.6. package *packname*

Create a new package with the given name. The parameter file associated with the current task, if any, is associated with the package and becomes the package's parameter file. All later task declarations will go into this package. A package declaration normally occurs in a script task, which creates the package and defines are tasks therein. If the package already exists, an error is indicated.

As an aside, if the name of an existing package is itself given as a command, then it is pushed and run as a kind of task; nothing is changed in the dictionary. Bye or EOF will pop this pseudotask and return the current package setting to its previous state. This is useful for temporarily changing the search path for commands when a few commands in a package are needed without having to worry about tasks with the same name in other packages being found instead (see §2.2).

11.7. redefine *[lt1, lt2, ...] lt = pt*

Exactly like the task directive except that redefinitions are allowed. A warning message is still issued, however, if a redefinition does occur.

11.8. set *[name = value]*

Make a new, or redefines an existing, environment entry. If given without arguments, all current entries are simply listed, one per line. Entries are made in the dictionary so are subject to the same rules as other dictionary objects, that is, entries are discarded when the task that does the set ends unless it uses keep. New entries are always made at the top of the list. Since searching also starts at the top, a second entry with the same name as an existing one will make the first entry inaccessible. An attempt is made to merge the environment facilities of the host operating system with the entries managed by set. Examples are given in the environment discussion.

11.9. task *[lt1, lt2, ...] lt = pt*

Define the logical task(s) found in the given physical task. All entries are made in the current package. Pt is the name of the physical task file. It may be in terms of environmental directories or, if quoted, may be given in host-dependent form. If it ends in ".cl", the file is assumed to be a script written in the CL language.

The logical tasks, lt1, lt2 and so on are the logical tasks that can be run from the physical task. At least one must be given. If the logical task name is prepended with a dollar, $, then no parameter file is to be associated with that task. If a newly declared logical task redefines an existing one in the current package, an error message is issued and the entry will not be made. Other logical tasks that do not conflict will still be entered, however. It is not an error to reference a physical task in more than one task command.

11.10. update *lt [, lt2, ...]*

Cause the in-core parameter file for the given task(s) to be written out. This is used in conjunction with cache to force an update of a parameter file before the current task ends. It may also be used to force an update of a parameter file that would not otherwise be updated, that is, when learn mode is not effect.

11.11. version

Give the current version number of the CL. The current implementation gives the time at which the program was built. The "version" of the CL for the near future is always considered to be 1.2.

11.12. ? and ??

The "?" command gives the names of all the logical tasks defined in the current package. The format is an indented, multicolumn block. Entries are read left-to-right top-to-bottom in the order in which they are searched (opposite of the order they were declared). The "??" command is similar but includes all packages. Packages and tasks that lie above the current package, and are thus not immediately accessible, are given in parentheses.

12. CL Parameters

Some of the parameters belonging to the CL logical task itself have special significance. Many of them have been mentioned elsewhere. These parameters behave according to all the usual rules but they are used internally by the CL or by utility tasks to specify options. All the CL parameters may be viewed with "lparam cl". CL parameters not included in the following list are provided as handy scratch variables. Other parameters will be added as time goes on.

param	*type*	*usage*
abbreviations	boolean	enables abbreviations
keeplog	boolean	enables command logging
logfile	filename	name of logging file
menus	boolean	automatically do a "?" when changing packages
mode	string	sets mode of CL task

13. An Example

This is a complete example of a package, **coord**, written for the CL environment. The package contains two logical tasks, **airmass** and **precession**. Airmass accepts **elevation** and **scale** and computes airmass. The result is printed and saved in a parameter **airmass**. Precession computes the precession between any two years, **year1** and **year2**. The ra and dec for year1 are read from the standard input and the precessed coordinates to year2 are written to the standard output. These two logical tasks are written in the SPP language and are defined in the single physical task, **coord.x**.

The following are examples of actual running programs. The name of the files in each case is given in boldface and is not part of the files. Numerous other examples can be found in the source directories for the IRAF system.

The login.cl file (see §11) defines the logical task **coord** as the script task **coord.cl** in its own directory.

file **login.cl**:

```
# When the CL starts up, it looks for a "login.cl" file in the
# current directory.  The login file should contain any commands
# or declarations which one wants executed upon startup.  A KEEP
# or CL command must be executed after the declarations section
# or the new definitions will go away.

# The logical directory uparm, if defined, is where the CL will
# save updated parameter files.  Other IRAF system routines also
# use this directory to store user-specific database files.

set  uparm = "/usr/jacoby/iraf/tasks/param/"
task $coord = "/usr/jacoby/iraf/tasks/coord/coord.cl"

keep # keep additions to dictionary when login.cl terminates
```

file **coord.cl**:

```
# CL script task to define and run the "coord" coordinate tool
# package.  When this script task runs, it defines the package
# "coord", the package directory "codir", and the two logical
# tasks comprising the package, AIRMASS and PRECESS.  The task
# CL is called to process commands from the user.  When CL
# terminates, COORD will also terminate (since there are no more
# commands in the file), causing the package and its contents to
# become undefined.

package coord

set     codir = "/usr/jacoby/iraf/tasks/coord/"
task    airmass, precess = codir$coord

cl()
```

file **airmass.par**:

```
# Parameters for logical task AIRMASS.

elevation,r,a,1.5708,0.,1.5708,elevation angle in radians
scale,r,h,750.,,,scale height
airmass,r,h,1.,,,computed airmass
```

file **precess.par**:

```
# Parameters for logical task PRECESS.

year1,r,h,1950,,,year from which coordinates are to be precessed
year2,r,a,1982.9,,,year to which coordinates are to be precessed
```

file **coord.x**:

```
# This file is written in the SPP language, which implements a
# subset of the planned IRAF scientific programming language.

# Define CL-callable tasks.
task    airmass, precess = precess_coords


# AIRMASS -- Airmass calculation utility.  Airmass formulation
# from Allen "Astrophysical Quantities" 1973, p. 125, 133.
#
# The logical task airmass has three parameters:
#       elevation       angular height above horizon
#       scale           scale height of atmosphere
#       airmass         calculated air mass

procedure airmass()

real    elevation, scale, airmass, x            # local variables
real    clgetr()                                # functions

begin
        # Get type-real parameters "elevation" and "scale" from CL.
        elevation = clgetr ("elevation")
        scale = clgetr ("scale")

        # Compute the airmass, given the elevation and scale.
        x = scale * sin (elevation)
        airmass = sqrt (x**2 + 2 * scale + 1) - x

        # Print result on the standard output, and output the
        # computed air mass to the CL parameter "airmass".

        call printf ("airmass: %10.3f\n")
            call pargr (airmass)
        call clputr ("airmass", airmass)
end



# PRECESS_COORDS -- Precess coordinates from year1 to year2.
# This task is a filter, which reads coordinate pairs from the
# standard input, performs the precession, and outputs the
# precessed coordinates on the standard output.
```

```
procedure precess_coords()

real    default_year1, year1            # year to precess from
real    default_year2, year2            # year to precess to
double  ra1, dec1                       # input coordinates
double  ra2, dec2                       # precessed coordinates
int     fscan(), nscan()                # formatted input functions
real    clgetr()                        # get real parameter function

begin
        # Get the default "year" parameters from the CL.
        default_year1 = clgetr ("year1")
        default_year2 = clgetr ("year2")

        # Read and precess coordinate pairs from the standard input
        # until EOF is detected.  Format "ra dec [year1 [year2 ]]".

        while (fscan (STDIN) != EOF) {
            call gargd (ra1)
            call gargd (dec1)
            call gargr (year1)
            call gargr (year2)

            if (nscan() == 3)                   # no year2 given
                year2 = default_year2
            else if (nscan() == 2)              # no year1 given
                year1 = default_year1
            else if (nscan() < 2) {
                call fprintf (STDERR, "invalid coordinates\n")
                next                            # do next iteration
            }

            # Call precession subprogram to precess the coordinates,
            # print result on standard output (hms hms yyyy.y).

            call precess (ra1, dec1, ra2, dec2, year1, year2)
            call printf ("ra: %12.1h  dec: %12.1h  %7.1f\n")
                call pargd (ra2)
                call pargd (dec2)
                call pargr (year2)
        }
end
```