Eli User Interface Reference Manual

$Revision: 2.2 $

Compiler Tools Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO, USA
80309-0425

Table of Contents

Eli focuses a user's attention on the information required to specify a text processor, rather than the tools implementing it, by automating the process of tool invocation. Desired products, such as an executable program or the result of a test run, are regarded as *derived objects*. Eli responds to a request for a derived object by invoking the minimum number of tools necessary to produce that object. Derived objects are automatically stored for re-use in future derivations, thereby significantly shortening the time required to satisfy requests.

This document is a reference manual for the general mechanisms by which a user can describe objects, make requests, and tailor Eli's behavior. For suggestions on using Eli to carry out specific tasks, see Section 'Example' in Guide for New Eli Users; for a complete list of the kinds of objects that can be derived, see Products and Parameters.

Although our primary emphasis here is on interactive use of Eli, all of the material except command line editing and the interactive help facility also applies to non-interactive use. The description of the Odinfile (see Chapter 5 [Odinfile], page 11) is particularly important in that context.

1 Referring to Objects

The objects in Eli's universe are divided into two classes, *source objects* and *derived objects*. Source objects are owned by the user of Eli, and may be manipulated in any way. Derived objects are owned by Eli; the user may inspect them or obtain copies of them on request, but has no direct access to them and cannot change them. Each derived object can be manufactured from some set of source objects, and therefore does not represent any *new* information.

Every object has a *name*. The name of a source object is its Unix file name and the name of a derived object is an *odin-expression*. The remainder of this chapter explains the form and meaning of an odin-expression.

Eli splits each file name into two parts: a *root* and a *type-name*. The type-name is the longest suffix of the file name that matches one of the declared source type suffixes. If no suffix match is found, the type-name is the empty string.

1.1 Lexical Conventions

Lexically, an odin-expression consists of a sequence of *identifier* and *operator* tokens terminated by a newline character. An odin-expression can be continued on multiple lines by escaping each newline character with a backslash. This backslash (but not the newline) is deleted before the expression is parsed. Multiple odin-expressions can be specified on the same line by separating them with semicolons.

An identifier token is just a sequence of characters. The following characters must be escaped to be included in an identifier:

```
: + = ( ) / % ; ? $ < > ! <space> <tab> <newline> # \ '
```

A single character can be escaped by preceding it with a backslash (e.g. `lost\+found`). A sequence of characters can be escaped by enclosing them in apostrophes (e.g. `'lost+found'`).

Unescaped *white space* characters (spaces, tabs, and newlines) are ignored during parsing except when they separate adjacent identifiers. A comment begins with a sharp and is terminated by the next newline character:

```
# this is a comment
```

Each comment is equivalent to white space.

An odin-expression can be surrounded by parentheses. Parentheses are required for nested odin-expressions (such as values of parameters) or for the empty expression `()` which represents an immutable empty file.

1.2 Selection Expressions

A selection expression, indicated by the slash operator, selects a file from a directory. The argument to the slash operator is the file name of the desired file. For example, the following odin-expression selects `prog.c` from the directory `src`:

```
src/prog.c
```

Any special character must be escaped. For example, `src/c++/prog.c` must be escaped, as in `src/c\+\+/prog.c` or `'src/c++/prog.c'`.

1.3 Derivation Expressions

A derivation expression, indicated by the colon operator, is used to specify a derived object. The argument to the colon operator is an *object type* (or *product*) (see Products and Parameters):

```
sets.specs :exe
sets.specs :source :names
```

The first line names the executable program derived from the specifications enumerated in file '`sets.specs`', while the second names a text file containing the names of the C files, header files and Makefile from which that program can be built.

A derived object can be a directory, in which case it is called a *derived directory*. Elements of a derived directory are selected with the same slash operator used to select elements of source directories. For example, if `examples/sets.specs:source` is a derived directory containing the source code files that implement a text processor, and this directory contains three files named '`Makefile`', '`driver.c`', and '`HEAD.h`', then these three files are named by the odin-expressions:

```
examples/sets.specs:source/Makefile
```

```
examples/sets.specs:source/driver.c
examples/sets.specs:source/HEAD.h
```

1.4 Parameterization Expressions

A parameterization expression, indicated by the `+` operator, extends an object with additional information that affects the derived objects produced from that object. The argument to the `+` operator is a *parameter type* (see Products and Parameters), optionally followed by an `=` operator and a *value* consisting of a sequence of one or more identifiers and parenthesized odin-expressions:

```
test +cmd=(sets.specs +fold :exe) +cmd=data :run
test +cmd=(sets.specs +fold :exe) data :run
test +cmd=(sets.specs +fold=' ' :exe) data :run
test +cmd=(sets.specs +fold :exe) +cmd=data +cmd=data :run
```

(We shall see that all of these lines are equivalent.)

If the parameter value is omitted, it is equivalent to specifying the identifier consisting of a single space `' '` as the value.

The parameter values of a given parameter type in an odin-expression form an ordered set, where the order of the values is the order specified in the odin-expression. If multiple copies of the same parameter value appear, only the first of the multiple copies is kept. The parameter value lists of each parameter type are disjoint, therefore, the order of parameters of different types is not significant.

If a parameter has a value that is a sequence, that value is only considered the same as another identical sequence. Thus the following odin-expression is *not* equivalent to those given above:

```
test +cmd=(sets.specs +fold :exe) data +cmd=data :run
```

Although `data` is an element of the parameter value list for parameter type `cmd`, it was introduced as part of a sequence. Its second appearance is as a single value, which is *not* equivalent to the previous sequence. Therefore it will not be considered a duplicate element, and the final list will be `(sets.specs +fold :exe) data data`.

Frequently, several odin-expressions share a common set of parameters. To support this kind of sharing, a parameterization expression can take the form of a `+` followed by a parenthesized odin-expression. The common set of parameters are then stored in the file named by the odin-expression. For example, suppose the file '`my.prms`' contained the text:

```
+debug +lib_sp=(/local/lib) +monitor +fold
```

The odin-expression `my.prms` denotes the file '`my.prms`' in the current directory, so the following odin-expressions would be equivalent:

```
sets.specs +(my.prms) :exe
sets.specs +debug +lib_sp=(/local/lib) +monitor +fold :exe
```

2 Bringing Objects Up To Date

A command consisting simply of an odin-expression requests Eli to bring the object named by the odin-expression up to date and report information (such as error and warning messages produced by tool steps that were run) concerning the object's status. The level of detail of this information is controlled by the value of the `ErrLevel` and `WarnLevel` variables (see Section 7.5 [ErrLevel], page 17).

2.1 Status of Objects

Associated with each object is a status level, where the status level is one of `OK`, `WARNING`, `ERROR`, `CIRCULAR`, `NOFILE`, and `ABORT`. `OK` is the maximum status level and `ABORT` the minimum.

The status of a given derived object depends on the results of the tools that produced that object. If any tool generated warning messages, the status level of the given object is at most `WARNING`. If any tool generated error messages, the status level of the given object is at most `ERROR`. If an object that was needed to create the given object is the object itself, the status level of the given object is at most `CIRCULAR`. If any object that was needed to generate the given object did not exist, the status level of the given object is at most `NOFILE`. If any object that was needed to generate the given object had status level `ERROR` or less, then the status level of the given object is set to be `ABORT`.

The status of a source object is `NOFILE` if the host file does not exist, the status of the value of its target value (see Chapter 5 [Odinfile], page 11) if it is a target, and otherwise `OK`.

2.2 Error and Warning Messages

The warning or error messages produced by all tool invocations are saved by Eli. The difference between an error and a warning is that an error prevents the tool from generating its output, while a warning indicates that although output was generated, it might be faulty. An example of an error message from a loader is:

```
Unsatisfied external reference: "proc1".
```

An example of a warning message from a loader is:

```
Multiply defined external: "proc2", first copy loaded.
```

A text file containing a summary of all error messages for an object can be obtained by applying the `:err` derivation to the object:

```
prog.c :exe :err
```

Here file `prog.c:exe:err` contains a summary of all error messages produced by any tool used in the generation of the `prog.c:exe` object. The `:warn` derivation produces a text file containing both warning and error messages for an object.

The `:err` and `:warn` derivations show the error reports just as they are produced by the tools. Eli's `error` derivation relates error messages to the source files causing them, and `:warning` does the same for both error and warning messages. Still more sophisticated anlysis is provided by the `:help` derivation, which starts a browsing session that links the errors to documentation and also makes the appropriate files available to the editor. These derivations have the same form as the derivations yielding the raw reports:

```
prog.c :exe :error
prog.c :exe :warning
prog.c :exe :help
```

3 Extracting and Editing Objects

A command that includes an angle bracket (`>` or `<`) requests Eli to copy the contents of a specified object into another object. The copy is performed only if the status level of the specified object is no lower than `WARNING` (see Section 2.1 [Status], page 6). The destination of the copy must be a source object, because only source objects can be directly modified by a user.

There are two forms of the copy odin-command: *copy-to*, indicated by a right-angle-bracket `>`, and *copy-from*, indicated by a left-angle-bracket `<`. Examples of these two odin-commands are:

```
-> sets.specs +debug :exe > prog
-> prog < sets.specs +debug :exe
```

If the destination object is a directory, the *label* of the specified object is used to name the new copy. The label of a source file is the last component of the pathname of the source file. The label of a derived object is *source-label.type-name* where *type-name* is the name of the output type of the tool that produced it (see Products and Parameters) and *source-label* is the label of the source file from which it is derived. For example, the label of `/usr/src/sets.specs` is `sets.specs` and the label of `/usr/src/sets.specs:exe` is `sets.specs.exe`.

If a list is copied into a directory, each element of the list is copied individually into the directory.

3.1 Copying to Standard Output

If the destination object is omitted from a copy-to odin-command, the specified object is displayed on the current standard output device. For example, the odin-command:

```
-> sets.con >
```

displays the file named `sets.con`.

3.2 Editing with the Copy Command

If only the destination object is specified in a copy-from odin-command, the specified object is given to the host-system editor indicated by the `$EDITOR` environment variable (see Section 7.9

[Environment Variables], page 18) with the `vi` editor the default. For example, if the value of the `$EDITOR` variable is `emacs`, then the following odin-command invokes the `emacs` editor on the file `prog.c`.

```
-> prog.c <
```

4 Execute Commands

A command that includes an exclamation point (`!`) requests Eli to execute a host command. In its most general form, such a command consists of an odin-expression followed by an exclamation-point and a host-command line. Either the odin-expression or the host-command can be omitted:

```
-> input +cmd=(sets.specs:exe) :stdout !more -s
-> ! ls *.c
-> build.specs :exe !
-> commands !
```

The result of the command is to bring the object named by the odin-expression up to date, append its filename to the host-command line, and give the resulting extended host-command line to the host system for execution.

If the host-command is omitted, the object itself is executed. If execute permission is set for the object, it is given to the host operating system for execution; otherwise, the object is assumed to contain `eli` commands that are executed by the interpreter.

The exclamation-point has the special lexical property that if the first non-white space character following it is not a colon, a semicolon, or an equal sign, then the rest of the line is treated as a single escaped sequence of characters. This avoids the confusion resulting from interactions between host-command and Eli character escape conventions. A leading colon, equal sign, or white space character can be included in the escaped sequence of characters by preceding it with a backslash.

5 The Odinfile

Eli consults file '`Odinfile`' in the current directory for information about the task at hand. '`Odinfile`' is used to define one or more *targets*. Each target defines some product that can be requested, using the notation *target* `==` *odin-expression*. Here are examples of the three common kinds of target:

`mkhdr == sets.specs :exe`

`mkhdr` is a *file target*. This line specifies that `mkhdr` should always be equal to the derived file object `sets.specs :exe`. If the command `eli mkhdr` is given in a directory with a file '`Odinfile`' containing this line, it will result in a non-interactive Eli session guaranteeing that file `mkhdr` in this directory is up to date. (The same effect can be obtained in an interactive session by responding to the `->` prompt with `mkhdr`.)

`%results == input +cmd=(mkhdr) :stdout`

`%results` is a *virtual target*. A virtual target is simply a name for an odin-expression, and can be used wherever and odin-expression is required. If the command `eli '%results>'` is given in a directory with a file '`Odinfile`' containing this line, it will result in a non-interactive Eli session guaranteeing that the derived object `input +cmd=(mkhdr) :stdout` is up to date, and writing the content to the standard output. (The same effect can be obtained in an interactive session by responding to the `->` prompt with `%results>`.)

`%test ! == . +cmd=diff (%results) (result) :run`

`%test` is an *executable target*. An executable target is a target that is executable. If the command `eli %test` is given in a directory with a file '`Odinfile`' containing this line, it will result in a non-interactive Eli session guaranteeing that the derived object `input +cmd=(mkhdr) :stdout` (named `%results`) is up to date, and executing the `diff` command with this object and the file '`result`' from the current directory as arguments. Execution will take place in the current directory. (The same effect can be obtained in an interactive session by responding to the `->` prompt with `%test`.)

6 The Command Editing Mechanism

During an interactive Eli session, an odin-command may be edited before it is interpreted by typing either control characters or escape sequences. (An escape sequence is entered by typing `ESC` followed by one or more characters. Note that unlike control keys, case matters in escape sequences; `ESC F` is not the same as `ESC f`.)

A control character or escape sequence may be typed anywhere on the line, not just at the beginning. In addition, a return may also be typed anywhere on the line, not just at the end.

Most control characters and escape sequences may be given a repeat count, *n*, where *n* is a number. To enter a repeat count, type the escape key, the number, and then the character or escape sequence:

```
ESC 4 ^F
```

This sequence moves the cursor forward four characters. If a command may be given a repeat count then the text "[*n*]" is given at the end of its description.

Eli accepts the following control characters when editing odin-commands:

`^A` Move to the beginning of the line

`^B` Move left (backwards) [*n*]

`^D` Delete character [*n*]

`^E` Move to end of line

`^F` Move right (forwards) [*n*]

`^G` Ring the bell

`^H` Delete character before cursor (backspace key) [*n*]

`^I` Complete filename (tab key); see below

`^J` Done with line (return key)

`^K` Kill to end of line (or column [*n*])

`^L` Redisplay line

`^M` Done with line (alternate return key)

`^N` Get next line from history [*n*]

`^P` Get previous line from history [*n*]

`^R`	Search backward (forward if [*n*]) through history for text; must start line if text begins with an uparrow
`^T`	Transpose characters
`^V`	Insert next character, even if it is an edit command
`^W`	Wipe to the mark
`^X^X`	Exchange current location and mark
`^Y`	Yank back last killed text
`^[`	Start an escape sequence (escape key)
`^]`*c*	Move forward to next character *c*
`^?`	Delete character before cursor (delete key) [*n*]

Eli accepts the following escape sequences when editing odin-commands:

`ESC ^H`	Delete previous word (backspace key) [*n*]
`ESC DEL`	Delete previous word (delete key) [*n*]
`ESC SP`	Set the mark (space key); see ^X^X and ^Y above
`ESC .`	Get the last (or [*n*]'th) word from previous line
`ESC ?`	Show possible completions; see below
`ESC <`	Move to start of history
`ESC >`	Move to end of history
`ESC b`	Move backward a word [*n*]
`ESC d`	Delete word under cursor [*n*]
`ESC f`	Move forward a word [*n*]
`ESC l`	Make word lowercase [*n*]
`ESC m`	Toggle whether 8-bit chars display normally or with the `M-` prefix
`ESC u`	Make word uppercase [*n*]
`ESC y`	Yank back last killed text
`ESC w`	Make area up to mark yankable
`ESC` *nn*	Set repeat count to the number *nn*
`ESC` *C*	Read from environment variable _*C*_, where *C* is an uppercase letter

If you type the escape key followed by an uppercase letter, *C*, then the contents of the environment variable _*C*_ are read in as if you had typed them at the keyboard. For example, if the variable `_L_` contains the following:

```
^A^Kecho '^V^[[H^V^[[2J'^M
```

Then typing `ESC L` will move to the beginning of the line, kill the entire line, enter the echo command needed to clear the terminal (if your terminal is like a VT-100), and send the line back to Eli.

The command editing mechanism also supports filename completion. Suppose the root directory has the following files in it:

```
bin vmunix core vmunix.old
```

If you type `rm /v` and then the tab key, Eli will finish off as much of the name as possible by adding `munix`. Because the name is not unique, it will then beep. If you type the escape key and a question mark, it will display the two choices. If you then type a period and a tab, Eli will finish off the filename for you:

`rm /v[TAB]`*munix*`.[TAB]`*old*

The tab key is shown by `[TAB]` and the automatically-entered text is shown as *munix*.

7 Variables

The behavior of Eli can be modified by changing the value of an *Odin variable*. The functions affected by Odin variables are the current working directory, the distributed parallel build facility, the help facility, the error and log facility, the file change notification facility, and the maximum total file system space used by derived objects.

A variable assignment odin-command consists of the name of an Odin variable followed by an `=` operator and an odin-expression. For example, the following odin-commands assign the value `../src` to the `Dir` variable and the value `4` to the `WarnLevel` variable (Odin variable names are case-insensitive).

```
-> dir = ../src
-> warnlevel = 4
```

If the value is omitted from a variable assignment odin-command, Eli displays the current value of the specified variable. After the preceding odin-commands, the current value of `WarnLevel` is found by the command:

```
-> warnlevel =
4
```

The Odin variables and their default values are:

`Dir` = eli_invocation_directory

`MaxBuilds` = 2

`BuildHosts` = LOCAL : LOCAL

`Size` = 0

`KeepGoing` = yes

`History` = yes

`LogLevel` = 2

`ErrLevel` = 3

`WarnLevel` = 2

`HelpLevel` = 1

```
VerifyLevel = 2
```

An initial value for an Odin variable can be specified in an environment variable whose name is the the Odin variable name in capital letters preceded by the string `ODIN`. For example, the initial value for `MaxBuilds` is specified in the `ODINMAXBUILDS` environment variable.

7.1 Dir

The current working directory can be changed by assigning a new value to the `Dir` variable. The value of the current working directory is especially significant for Eli, since it identifies source objects by their absolute pathname, and the current working directory provides the absolute pathname for all relative names.

7.2 BuildHosts, MaxBuilds

The `BuildHosts` variable specifies the list of hosts that are used to execute the tools that generate the derived objects. A tool is executed on the first entry in the `BuildHosts` list that does not have a currently executing tool. The name `LOCAL` refers to the local host. The `MaxBuilds` variable specifies the maximum number of tools to execute in parallel.

The hosts in `BuildHosts` must have the same machine architecture and file namespace as the local host.

7.3 KeepGoing

When a build step reports errors, Eli will continue with build steps that do not depend on the failed build step. Setting the value of the `KeepGoing` variable to `no` will cause Eli to terminate the build when any build step reports an error.

7.4 History

The `History` variable specifies whether the command line editing is supported by Eli when it is used as an interactive command interpreter (see Chapter 6 [Command editing], page 12).

7.5 ErrLevel, WarnLevel, LogLevel

When an odin-command is executed, Eli indicates any errors or warnings associated with the odin-expressions specified in that odin-command. The `ErrLevel` and `WarnLevel` variables specify how detailed this report is. In particular, the user can choose whether to see final status information, to see messages incrementally as they are produced by tools steps, or to see a summary of all relevant messages (including those from previously cached tool steps).

Eli can also produce a variety of information about the activities it is performing, such as a brief description of each tool that is invoked to satisfy a given request. The `LogLevel` variable specifies how detailed these messages are.

7.6 HelpLevel

The `HelpLevel` variable specifies what degree of detail is provided when the user asks for a list of possible file or parameter types (see Chapter 8 [Help], page 19). Normally, only commonly used types are described, but the `HelpLevel` can be increased to have all possible types described.

7.7 VerifyLevel

By default, Eli checks the modification dates of all relevant source files at the beginning of a session and before each interactive odin-command. If all file modifications during the session are performed through copy odin-commands or through an editor that has been upgraded to send a `filename!:test` odin-command to Eli whenever `filename` is modified, the `VerifyLevel` variable can be set to `1` and only the check at the beginning of the session is performed. If all file modifications since the last session have been performed in the above manner, `VerifyLevel` can be set to `0` and the initial check is avoided as well.

7.8 Size

The value of the `Size` variable indicates how much disk space (in kilobytes) is currently being used by derived files.

7.9 Environment Variables

Environment variables can be used in odin-commands given during interactive sessions, but are not allowed in an Odinfile (see Chapter 5 [Odinfile], page 11). For example, if the environment variable `$HOME` has the value '`/u/geoff`', then the following two odin-commands are equivalent.

```
-> $HOME/sets.specs :exe
-> /u/geoff/sets.specs :exe
```

The value of an environment variable can be quoted by immediately preceding it with a quoted identifier. For example, if the value of `$DATA` is `/french/words`, then the following two odin-commands are equivalent.

```
-> sets.specs +monitor +arg='/u/geoff'$DATA :mon
-> sets.specs +monitor +arg='/u/geoff/french/words' :mon
```

An environment variable is given a new value with a variable assignment odin-command of the form: *Variable* `= !` *Value*. (Note the use of `!`, in contrast to an odin-variable assignment. It suspends Eli's lexical conventions – see Chapter 4 [Execute Commands], page 10.) Thus the following odin-command sets the value of the environment variable `$HOME` to the value '`/u/clemm`':

```
-> HOME = !/u/clemm
```

The expressions `~` and `~name` are treated as if they were environment variables, bound respectively to the login directory of the current user and the login directory of the user with login `name`.

8 The Help Facility

A simple context-sensitive help facility is provided to describe the syntax of odin-commands and the currently available object types and parameter types. If a user types a question-mark anywhere in an odin-command, Eli provides a description of what could appear at that location in the odin-command.

8.1 Source Type Help

If a list of the declared source object type-names is desired, a question-mark can be put in place of the extension for a file:

```
-> sets?
?*? Known Suffix Types:
.lex_code ..... Basic symbol coding
.reqmod ....... Names of required modules
.reqsym ....... Entry point symbols of required modules
.dapto ........ Specification of events and messages for monitoring
.delit ........ Literals to be deleted from the finite-state machine
.specs ........ Set of specifications defining the desired processor
.cola ......... Options for the parser generator cola
.finl ......... Operations to be executed after finishing
.gnrc ......... Generic module specification
.head ......... Information to be prefaced to attribution modules
.init ......... Operations to be executed before starting
.libs ......... Libraries to include in the link
.lido ......... Attribute grammar written in LIDO
.perr ......... Parser error recovery information
.roff ......... nroff/troff input
,eqn .......... eqn input
.bib .......... TeX bibliograph database
.clp .......... CLP specification
.con .......... Concrete syntax
.ctl .......... Control options for LIGA processing
.dvi .......... Device-independent formatted file from TeX
.gla .......... Structure of comments and named terminals
.map .......... Concrete/Abstract syntax mapping
.oil .......... OIL specification
.pdl .......... PDL specification
.phi .......... Files to be included at specified places
.ptg .......... PTG specification
.str .......... String table initialization
.sym .......... Symbolic grammar mappings
.tex .......... TeX formatter input
```

```
.tnf .......... Specification of a hypertext document
.ygi .......... input grammar for the Tregrm tree-building parser generator
,vw ........... a view-path system model
.dg ........... Odin Derivation Graph
.fw ........... FunnelWeb specification
.ps ........... Postscript file
.sm ........... system model of source code files
,v ............ RCS version control file
.a ............ object library archive
.f ............ Fortran77 source code
.l ............ scanner grammar
.y ............ YACC input
```

This should be interpreted to mean that Eli will understand the types of the following source files (among others):

```
sets.specs
fortran.con
build.HEAD.phi
```

8.2 Derivation Help

If a list of possible derivations is desired, a question-mark can be put in place of the derivation name, and Eli responds with a list of the possible object types that can appear at that position:

```
-> sets.specs :exe :?
*?* Possible Derivations:
name .......... name of a file
dir ........... directory of a file
exe ........... Executable program
label ......... label of a file
warn .......... warnings
help .......... Hypertext presentation of messages
warning ....... Standard presentation of warning messages
error ......... Standard presentation of error messages
err ........... errors
filename ...... filename of a file
depend ........ source dependencies
profile ....... execution profile
redo .......... redo this object step
redo_errs ..... redo all steps with errors
diff .......... differences between two files
rcp ........... archive
roff .......... nroff/troff input
eqn ........... output from eqn
```

```
tbl ........... output from tbl
nroff ......... output from nroff
stdout ........ standard output from a test run
output ........ output files from a test run
```

This should be interpreted to mean that Eli will understand the following derivations (among others):

```
sets.specs :exe :help
sets.specs :exe :redo
sets.specs :exe :depend
```

8.3 Parameterization Help

If a list of the possible parameter types is desired, a question-mark can be put in place of the parameter, and Eli responds with a list of the possible parameter types that can appear at that position:

```
-> sets.specs :exe +?
*?* Possible Parameters :
ignore ........ Prefix of include file names to be ignored
lib ........... a library name
lib_sp ........ name of a directory in an library search path
prof_data ..... trace file
prof_flags .... prof flags
default ....... default value
other ......... another file
f_dest ........ file destination
d_dest ........ directory destination
mp ............ macro package
cmd ........... host command
need .......... run dependency
```

This should be interpreted to mean that Eli will understand the following derivations (among others):

```
sets.specs :exe +ignore ...
sets.specs :exe +d_dest ...
sets.specs :exe +prof_data ...
```

A more exact form of parameter help can be specified by indicating which derivation you intend to apply to the parameterized object:

```
-> sets.specs :exe +? :profile
*?* Possible Parameters :
prof_data ..... trace file
prof_flags .... prof flags
```

This should be interpreted to mean that Eli will understand the following derivation (among others):

```
sets.specs :exe +prof_data=foo :profile
```

Since the `+cmd` parameter is not relevant to the derivation from `:exe` to `:profile`, it is not listed.

8.4 Variable Help

A list of the available variable names is generated in response to the request `?=`:

```
-> ? =
Dir MaxBuilds BuildHosts Size KeepGoing History
LogLevel ErrLevel WarnLevel HelpLevel VerifyLevel
```

A description of the possible values that can be assigned to a given variable is generated in response to the *Variable* `= ?`:

```
-> LogLevel = ?
0: No log information is generated.
1: Build commands are echoed.
2: And Eli commands.
3: And names of objects with errors.
4: And names of objects generated by tool scripts.
5: And names of objects generated by internal tools.
6: And names of objects deleted.
7: And names of objects touched by broadcast.
```

Index