Products and Parameters Reference

$Revision: 2.15 $

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

Table of Contents

1 Processor Generation 3
1.1 exe — Executable Version of the Processor 3
1.2 so — Shared library Version of the Processor 3
1.3 source — Source Version of the Processor 3
1.4 fwGen — Obtain the Product Files Generated by FunnelWeb 4
1.5 fold — Make the Processor Case-Insensitive 4
1.6 define — Set C Pre-Processor Symbols 4
1.7 cc_flags — Set C Compiler Options 4
1.8 cc — Choose a Particular C Compiler 5
1.9 ignore — Do Not Verify include Files 5
1.10 parser — Choose the parser generator 5
1.11 nomain — Omitting the main program 5

2 Generating Specifications 7
2.1 consyntax — Concrete Syntax 7
2.2 abstree — Abstract Tree Grammar 7
2.3 absyntax — Abstract Syntax 7
2.4 pgram – Parsing Grammar 7
2.5 kwd — Recognize Specified Literals as Identifiers 8
2.6 inst — Instantiate a Generic Module 8
2.7 ExpInfo — Information about remote attribute access 9
2.8 OrdInfo — Information about attribute dependence 9
2.9 OptimInfo — Information about attribute storage optimization 10
2.10 show — LIDO Table Viewers showFe and showMe 10
2.11 instance — Name an Instance of a Generic Module 10
2.12 referto — Relate Instances of Generic Modules 10

3 Diagnosing Specification Inconsistencies 11
3.1 warning — Warning Messages and Error Reports 11
3.2 error — Error Reports 11
3.3 help — Cross-Reference to Documentation 11
3.4 parsable — Verify that the Grammar is Parsable 12
3.5 gencode — Code Derived from the Specifications 12

4 Testing a Generated Processor 13
4.1 cmd — Command Line to be Executed 13
4.2 stdout — Standard Output from Processor Execution 13
4.3 output — Files Resulting from Processor Execution 13
4.4 run – Execute the Processor in the Current Directory 13
4.5 Debugging — Debug a Program Interactively at the Source Level 14

4.6 Monitoring — Monitor a program at the specification level 14
4.7 monitor — Request monitoring support 15
4.8 arg — Supply Command Line Parameters 15
4.9 debug — Request debugging information in object files 15
4.10 printtokens — Request Token Printing Code 15

5 Producing Formatted Documents 17

5.1 ps — PostScript file 17
5.2 dvi — Device-independent TeX typesetter file 17
5.3 fwTex — TeX input file 17
5.4 fwHtml — HTML file 17

6 Obtaining Information About the Derivation 19

6.1 inputs — Objects on which a Given Object Depends 19
6.2 outputs — Objects Depending on a Given Object 19
6.3 test — Check Whether an Object has been Modified 19
6.4 redo — Ask Eli to Carry Out a Derivation 19

Index 21

A *product* is a Unix file or directory that can be requested from Eli (see *Eli User Interface Reference Manual*). *Parameters* allow the requestor to control some characteristics of the requested product. This document defines the available products and associated parameters. It is intended as a reference manual rather than a tutorial. The products and parameters are grouped by function; use the index to find a particular product or parameter by name.

Eli's products and parameters can be divided into six groups: The *processor* group allows a user to generate a processor and extract it from Eli, in either executable or source form, or to copy the complete set of files defining a processor. Inconsistencies in the specifications cause Eli to set either warning or abort status for the request being made; additional information about these inconsistencies can be obtained via the *diagnostic* group of products. Even though a set of specifications is consistent, it may not be correct. The *testing* product group should be used to exercise the processor, verifying that the specifications solve the proper problem. Products and parameters in the *document* group provide formatted versions of specifications that can be put on paper or rendered as hypertext. Finally, the *derivation* group allows the user to obtain information about the details of how products are derived and influence the derivation process in certain ways.

1 Processor Generation

Eli's task is to create a text processor that implements a set of specifications. Two forms of that processor can be extracted from Eli: an executable form and a source form. The processor can also be tested without extracting it from Eli (see Chapter 4 [Testing], page 13). Regardless of the ultimate disposition of the processor, its behavior can be influenced in a number of ways by parameters.

1.1 exe — Executable Version of the Processor

```
:exe
```

A binary file containing the executable program generated from the given specifications. During specification development, `:exe` is used as a means of collecting error reports (see Section 3.3 [help], page 11):

```
pascal.specs:exe:help
```

When this derivation yields an empty result, indicating that no errors were detected in the specifications, the generated processor can be tested by requesting the `stdout` (see Section 4.2 [stdout], page 13), `output` (see Section 4.3 [output], page 13), `dbx`, `gdb` (see ⟨undefined⟩ [debugging], page ⟨undefined⟩), `mon`, `mongdb` (see ⟨undefined⟩ [monitoring], page ⟨undefined⟩) or `run` (see Section 4.4 [run], page 13) products. After testing has been completed, `exe` can be used to extract the executable processor from the cache for use in arbitrary contexts:

```
pascal.specs:exe>pc
```

1.2 so — Shared library Version of the Processor

```
:so
```

A shared library containing the executable program generated from the given specifications. Shared libraries do not usually contain a main program, and therefore `:so` is almost always derived using the `+nomain` parameter (see Section 1.11 [nomain], page 5).

1.3 source — Source Version of the Processor

```
:source
```

A directory containing the source language files and header files necessary to construct the generated processor. This directory is completely self-contained, unless a library specification was given by the user. All Eli library routines used are included in source form. There is also a makefile, so an executable processor can be constructed by entering the directory and giving the command `make`.

To extract the directory from the cache, use the following Eli request:

```
pascal.specs:source>dir
```

Here *dir* is the name of an existing directory. The directory may or may not be empty. If it is empty, Eli will copy all of the files needed to construct the processor (except user-specified libraries) from the cache to *dir*. Otherwise, Eli will copy only those files that differ from the files with the same name in *dir*. This means that if you alter the specifications for the processor and then request a new source directory, only the files actually affected by your specification change will be updated. Therefore if you run `make` in *dir*, unaltered routines will not be recompiled.

1.4 fwGen — Obtain the Product Files Generated by FunnelWeb

```
:fwGen
```

A directory containing the files generated by the FunnelWeb "tangle" phase. See Section "Output Files" in *FunnelWeb User's Manual*. This derivation can only be applied to type-`fw` files. The resulting directory will contain all files generated by `@O` macros in the FunnelWeb file. For example, to generate the "tangled" files from the file '`pascal.fw`', use the following Eli request:

```
pascal.fw:fwGen>dir
```

Here *dir* is the name of an existing directory.

1.5 fold — Make the Processor Case-Insensitive

```
+fold
```

Eli normally produces a compiler that treats upper and lower case letters as distinct characters in identifiers and keywords. If `fold` is used, implementation of the processor `mkidn` is altered so that it treats the upper and lower case versions of a letter as being identical.

The `fold` causes all identifiers to be stored in their upper-case form. When an identifier is output, therefore, it will appear in upper case regardless of its representation in the input text.

1.6 define — Set C Pre-Processor Symbols

```
+define='item ...'
```

All C compilations required by the derivation will be provided with flag `-D`*item* for each item in the argument. An item is any sequence of non-white-space characters other than apostrophes. Items are separated by white space. Any number of items may be specified.

The following items are pre-defined:

`STOPAFTERBADPARSE`
: Generate a processor that will terminate after parsing if syntax errors are detected (see Section "Improving Error Recovery in the Generated Parser" in *Syntactic Analysis*).

1.7 cc˙flags — Set C Compiler Options

```
+cc_flags=flag ...
```

All C compilations required by the derivation will be provided with each *flag* in the argument. A flag is any sequence of non-white-space characters, possibly enclosed in apostrophes. Flags are separated by white space. Any number of flags may be specified.

This parameter is often used to specify include files for components that are not generated by Eli. For example:

```
+cc_flags='-I/opt/icedtea-bin-6.1.11.1/include'
```

1.8 cc — Choose a Particular C Compiler

```
+cc=command
```

All C compilations required by the derivation will be carried out by executing the command *command*.

1.9 ignore — Do Not Verify include Files

```
+ignore='path'
```

Eli normally verifies the presence of files requested by `#include` directives in any file that it processes. The `ignore` parameter tells Eli not to verify the presence of such files if their full names begin with *path*.

If a derivation does not specify any `ignore` parameters, `+ignore='/usr/include'` is assumed. (This assumption is not made if any `ignore` parameters are given explicitly.)

1.10 parser — Choose the parser generator

```
+parser
+parser=pgs
+parser=cola
+parser=none
```

Eli incorporates two parser generators, *PGS* and *COLA*. Both produce parsers from LALR(1) grammars, and the parsers they produce recover automatically from syntactic errors. A PGS-generated parser does not reduce by chain productions. This means that it accepts some grammars that are not LALR(1), and the generated parser is faster than a COLA-generated parser for languages with many precedence levels in expressions. Parsers generated by COLA include more information in error messages about expected tokens than those generated by PGS do.

The error reports provided by the two parser generators differ somewhat, and it is sometimes useful to try the other generator when one produces a message you don't understand. Remember, however, that PGS accepts a wider class of grammars. Thus COLA may report errors for a grammar and PGS may not.

One of the two parser generators is selected as the default at the time the Eli system is generated. The default parser generator will be used if the `parser` parameter is absent. A request specifying `+parser` only will cause the non-default parser generator to be used. Specifying `+parser=pgs` will cause PGS to be used, while specifying `+parser=cola` will cause COLA to be used.

The default parser generator can be changed by changing the shell scripts '`parserOut.sh`', '`bnfOut.sh`' and '`infoOut.sh`' in directory '`$elipkg/parser`'.

Sometimes it is desirable to not have a generated parser. For example, the user may be building a tree for attribution using hand-written code or code produced by an application external to Eli. Specifying `+parser=none` will cause Eli not to generate a parser.

1.11 nomain — Omitting the main program

```
+nomain
```

Normally Eli generates a main program that invokes the generated components in an appropriate order. Sometimes the user wants to have more control by providing their own main program. In this situation `+nomain` should be specified to omit the standard Eli main program.

2 Generating Specifications

Sometimes a common problem can be solved by a collection of specifications having a particular structure. These specifications themselves can be generated, given simpler specifications. The products and parameters described in this chapter are most often used in requests that appear in type-`specs` files. They result in specifications that are then used to describe the complete processor.

2.1 consyntax — Concrete Syntax

```
:consyntax
```

Requesting `:consyntax` will result in a type-'`con`' file containing the complete concrete syntax. This includes all concrete syntax rules provided by the user in type-'`con`' files (translated into strict BNF form) as well as rules added to the concrete syntax as a result of the mapping process (see Section "Mapping" in *Syntactic Analysis*).

2.2 abstree — Abstract Tree Grammar

```
:abstree
```

Requesting `:abstree` will result in a type-'`lido`' file containing the complete abstract tree grammar. This consists of all rules supplied by the user in type-`lido` files, plus any additional rules derived from rules appearing only in type-`con` files in the syntax mapping process (see Section "Mapping" in *Syntactic Analysis*).

2.3 absyntax — Abstract Syntax

```
:absyntax
```

Requesting `:absyntax` will result in a type-`lido` file containing the complete abstract syntax. This derivation differs from the `:abstree` derivation (see Section 2.2 [abstree], page 7) in that rules which can only be derived for computed subtrees (see Section "Computed Subtrees" in *LIDO - Reference Manual*) are not included.

2.4 pgram – Parsing Grammar

```
:pgram
```

The parsing grammar requested by the `:pgram` derivation is the input to the parser generator. This includes the integer encodings of the terminal symbols, any directives supplied by the user to direct the automatic error correction of the parser, and each of the BNF rules in the concrete syntax with associated actions. These actions include actions supplied by the user as well as actions generated to construct an abstract syntax tree.

Note that the mapping process may cause the injection of certain chain rules due to `BOTTOMUP` constraints specified in the attribute grammar. These injected rules do not appear in the result of the `:consyntax` derivation, which makes `:pgram` the most appropriate derivation to consult when trying to resolve parsing conflicts which may have resulted from the injection of these chain rules. See Section "BOTTOMUP" in *Syntactic Analysis*.

2.5 kwd — Recognize Specified Literals as Identifiers

```
file.gla :kwd
```

Specifications that force literals whose lexical structure is defined by `file.gla` to be handled specially by the generated scanner.

Normally, each literal found in the context-free grammar is recognized explicitly by the scanner as the specified character sequence. This is in contrast to identifiers, denotations and comments, whose lexical structures are defined by regular expressions. The literals are defined by their appearance in the grammar, and no type-`gla` file describes their structure.

The `:kwd` product is used when literals are representative of the character strings that should appear in the program, but not necessarily identical to them. The classic case in which `:kwd` would be used (and from which its name is derived) is the recognition of mixed-case keywords. A literal `'if'` appearing in a Pascal grammar is representative of the character strings `if`, `If`, `iF` and `IF` but is identical to only the first. Whenever any one of these four character strings appears in a Pascal program, it should be recognized by the scanner as an instance of the literal `'if'`. All Pascal keywords behave in this fashion, and each has the form of an identifier.

Suppose that file `pkeys.gla` defines the structure of the literals used in the grammar, and the following line were added to one of the type-`.specs` files for a Pascal compiler:

```
pkeys.gla :kwd
```

This would prevent the scanner of the generated compiler from recognizing Pascal keywords explicitly as the character sequences specified by the literals given in the grammar. Other literals (such as `:=`), which did not fit the definition given by `pkeys.gla`, would still be recognized by the scanner as the specified character sequences.

It is important to remember that the type-'`gla`' file to which the `kwd` derivation is applied defines the form of the literals *in the grammar*, not in the input text.

2.6 inst — Instantiate a Generic Module

```
file.gnrc :inst
```

A specification consisting of one or more files is generated from `file.gnrc`. This product is normally used to instantiate the common problem solutions that have been stored in the library (see Section "Module Instantiation" in *Specification Module Library*):

```
$elipkg/Name/AlgScope.gnrc :inst
```

Here the path '`$elipkg/Name`' accesses the portion of the library devoted to problems arising in the context of name analysis (see Section "Name Analysis Library" in *Specification Module Library: Name Analysis*).

Type-`gnrc` files can be supplied by a user. They are simply shell scripts that carry out whatever actions are needed to instantiate a generic specification. Thus a user of Eli can construct generic specifications appropriate to a specific domain and use them exactly like library specifications.

Scripts appearing as type-`gnrc` files are invoked with up to three parameters. The first parameter is the `sed`(1) program, the second parameter is the string specified by `+instance`, and the third parameter is the string specified by `+referto`. When the script is invoked, its full path name is used. Therefore the script can determine the directory in which it is stored, and access specific files in that directory.

Here is an example of a type-`gnrc` file, '`$elipkg/Name/AlgScope.gnrc`':

```
#!/bin/sh
# $Id: pp.tnf,v 2.15 2012/07/30 22:55:15 profw Exp $
# Copyright, 1994, AG-Kastens, University Of Paderborn

moddir=‘expr $0 : '\(.*\)/.*' \| '.'‘

$1 -e "s/|NAME|/$2/g
s/|KEY|/$3/g" "$moddir"/AlgScope.fw > "$2"AlgScope.fw
```

It first sets `moddir` to the name of the directory in which it resides, then uses `sed` to modify file '`AlgScope.fw`' in that directory. The script replaces the string `|NAME|` with the string specified by the `+instance` parameter of the original request. (If no `+instance` parameter was supplied, `$1` is empty and every occurrence of the string `|NAME|` is simply deleted.) Similarly, it either replaces the string `|KEY|` with the string specified by the `+referto` parameter or deletes it. Finally, the name of the output file depends on the `+instance` parameter of the original request.

The output file will become part of the specification that contained the `:inst` request.

2.7 ExpInfo — Information about remote attribute access

```
:ExpInfo
```

Obtain information about the processing of LIDO specifications, especially information concerning the expansion of remote attribute accesses (i. e. INCLUDING, CONSTITUENTS, and CHAIN). The generated listing describes how each remote access construct can be replaced by a set of equivalent computations propagating the accessed values through adjacent contexts. This file is useful if special difficult cases regarding problems with remote dependences arise.

Additional information about attribute dependences and attribute storage optimization can be obtained by adding the parameters `+OrdI` and `+OptimI`.

```
Example: foo.specs+OrdI:ExpInfo>
```

For a more detailed description of Liga's protocol options and more advanced options, see Section "Liga Control Language Manual" in *Liga Control Language Manual*.

2.8 OrdInfo — Information about attribute dependence

```
:OrdInfo
```

Obtain information about the processing of LIDO specifications, especially information concerning the attribute dependences. The protocol provides for each grammar rule the set of direct dependences between attributes occurring in this rule.

Additional information about remote attribute access and attribute storage optimization can be obtained by adding the parameters `+ExpI` and `+OptimI`.

```
Example: foo.specs+ExpI:OrdInfo>
```

For a more detailed description of Liga's protocol options and more advanced options, see Section "Liga Control Language Manual" in *Liga Control Language Manual*.

2.9 OptimInfo — Information about attribute storage optimization

`:OptimInfo`

Obtain information about the processing of LIDO specifications, especially information on attribute storage optimization. For each attribute this protocol provides information where this attribute is stored. Possible storage locations are "tree node", "global variable" and "global stack".

Additional information about remote attribute access and attribute dependences can be obtained by adding the parameters `+ExpI` and `+OrdI`.

```
Example: foo.specs+ExpI:OptimInfo>
```

For a more detailed description of Liga's protocol options and more advanced options, see Section "Liga Control Language Manual" in *Liga Control Language Manual*.

2.10 show — LIDO Table Viewers showFe and showMe

```
:showFe
:showMe
```

Obtain a list of files that contain internal representations of LIDO text translated into readable text. `:showFe` shows LIDO text after the processing by the frontend of the Liga-System, `:showMe` shows the same information after attribute evaluator construction.

These informations can be useful for debugging a LIDO-Specification or to understand LIGA-Processing in more depth.

See Section "Overview" in *SHOW - Debugging Information for LIDO*, for more details.

2.11 instance — Name an Instance of a Generic Module

`+instance='`*`string`*`'`

Use *string* to name the instance generated by the request. No spaces are allowed within the *string*, and characters meaningful to the shell that interprets the associated type-`gnrc` file may cause problems.

Not all generic modules allow distinct instances to be created. See the documentation of each such module for the precise effect of `+instance`.

2.12 referto — Relate Instances of Generic Modules

`+referto='`*`string`*`'`

Use *string* to relate the current instance of a generic module to some specific instance of another generic module.

Not all generic modules allow relationships to be specified. See the documentation of each such module for the precise effect of `+referto`.

3 Diagnosing Specification Inconsistencies

Inconsistencies in the specification are detected by Eli in much the same way as a compiler detects inconsistencies in a normal program. Depending upon the nature of the inconsistency, Eli will set the status of the requested processor to either "warning" or "abort". In either case, the user is informed of the status of the requested product. If the status is "abort", the product will not be delivered. To obtain further information about the inconsistencies, a request should be made to derive one of the diagnostic products discussed in this section from the product whose status was reported as "warning" or "abort". For example, suppose that the following request (see Section 1.1 [exe], page 3) led to an abort status:

```
pascal.specs :exe
```

Further information could then be obtained by making the request:

```
pascal.specs :exe :help
```

Note that the original request is repeated exactly, with `:help` being simply added to the end. (See Section "Referring to Objects" in *Eli User Interface Reference Manual*, for more information about the form of a request.)

3.1 warning — Warning Messages and Error Reports

```
:warning
```

A listing of all error reports and warning messages delivered by any step in a derivation. For example, the following request might be used during development of a set of specifications for a Pascal compiler:

```
pascal.specs:exe:warning
```

The product being requested here is the reports of anomalies encountered during the derivation `pascal.specs:exe`. These reports indicate inconsistencies in the specifications listed in the file '`pascal.specs`'.

3.2 error — Error Reports

```
:error
```

A listing of all error reports delivered by any step in a derivation. During development it is preferable to use the `help` product (see Section 3.3 [help], page 11), because it provides more information than `error`. If a particular derivation produces a number of uninteresting warnings, however, `error` is useful.

3.3 help — Cross-Reference to Documentation

```
:help
```

Starts an interactive session with the hypertext reader (see Section "System Documentation" in *Guide for New Eli Users*). This session is an explanation of reports delivered during a derivation. The session provides a menu of files containing the reports, and for each file a menu of hypertext document nodes pertaining to the errors in that file.

Any file supplied as part of the specification can be altered (using the edit command of the hypertext reader), and when the session ends Eli will take account of those alterations.

Generated files can be examined with the edit command, and altered copies created for testing purposes, but those alterations will *not* be permanent.

There is no use in requesting the `help` product unless a derivation has errors or warnings.

3.4 parsable — Verify that the Grammar is Parsable

```
:parsable
```

Verifies that the grammar satisfies the conditions necessary to generate a parser.

The result of this derivation should be directed to the screen or an editor (see Section "Extracting and Editing Objects" in *Eli User Interface Reference Manual*). Using the `:help` derivation is *not* recommended, because the output is hard to read.

3.5 gencode — Code Derived from the Specifications

```
:gencode
```

A directory containing the original specification files and all files generated from them. This directory is useful for exploring the relationships among the generated code files when certain inconsistencies are detected.

Although the directory can be extracted from the cache (see Section "Extracting and Editing Objects" in *Eli User Interface Reference Manual*), it is usually examined by requesting the `viewlist` product:

```
sets.specs :gencode :viewlist
```

This derived object is an interactive execution of the user's preferred shell in the derived directory `sets.specs :gencode`. It allows the user to employ all of the usual tools (`lint`(1), `ctags`(1), editors, and so forth) to perform arbitrary analysis on the files. Any file supplied as part of the specification can be altered directly, and when the session ends Eli will take account of those alterations. Generated files can be examined, and altered copies created for testing purposes, but those alterations will *not* be permanent.

At the beginning of the session, the directory contains only symbolic links. If you want to alter a file temporarily, simply delete the symbolic link and replace it with the copy of the file. You may also create other files in the directory as you see fit. If you create subdirectories of the local directory, you should delete those subdirectories before exiting the shell. Plain files need not be deleted.

If you exit the shell and then re-derive `gencode`, any files you created may or may not be present. Thus you should not expect that files you create will be retained; any file you wish to keep should be written to another directory.

4 Testing a Generated Processor

After all of the inconsistencies detected by Eli have been removed from a set of specifications, the generated processor must be run with typical input data to verify that the specifications actually describe the desired behavior. Since improper processor behavior is always due to a specification error, the specifications must be altered whenever such behavior occurs. This means that a new processor must be generated and testing continued. The products and parameters described in this section allow the user to remain within the Eli system during this entire process.

4.1 cmd — Command Line to be Executed

```
+cmd
```

A specification of a command line to be executed by the host machine. This parameter is a list of the words of the command line. Each word is specified by an odin-expression, and may be either a string or a file. Files are specified by parenthesized odin-expressions; strings are not parenthesized:

```
+cmd=(sets.specs :exe)
+cmd=diff (result) (expected)
+cmd=sort -u
```

4.2 stdout — Standard Output from Processor Execution

```
:stdout
```

A file containing the result of applying a filter to another file. The filter is specified by the `+cmd` parameter attached to the filtered file. If no `+cmd` parameter is attached to the filtered file then the result is simply the filtered file itself.

A typical situation is to use the generated processor to filter a data file:

```
data +cmd=(sets.specs :exe) :stdout
```

4.3 output — Files Resulting from Processor Execution

```
:output
```

The execution directory after applying a filter to a file. The filter is specified by the `+cmd` parameter attached to the filtered file. If no `+cmd` parameter is attached to the filtered file then the result will be an empty directory.

A typical situation is to use the generated processor to filter a data file. If the generated processor creates files as a side effect of processing the data, these files will be in the derived directory:

```
data +cmd=(sets.specs :exe) :output
```

4.4 run – Execute the Processor in the Current Directory

```
:run
```

Execution of a specified command line in a specified directory. This derivation must be applied to a directory object, and requires a `+cmd` parameter (see Section 4.1 [cmd], page 13).

```
. +cmd=(sets.specs :exe) (input) :run
testdir +cmd=myprog -u (input (sets.specs :exe) :stdout) :run
```

Every request for `run` re-executes the command, regardless of whether anything on which the execution depends has changed. No other products can be derived from `run`. In particular, it is not possible to obtain warning messages or error reports by appending `:warning` or `:error` (see Section 3.1 [warning], page 11) to this derivation. If a warning message or error report is generated, Eli will specify the derivation step for which warning or abort status was set. The target of the derivation should be changed to this step, followed by `:warning` or `:error`, and the modified request submitted to Eli.

4.5 Debugging — Debug a Program Interactively at the Source Level

```
:dbx
+core=( file ) :dbx
:gdb
+core=( file ) :gdb
```

`dbx` starts an interactive session with the source-level debugger of the machine on which Eli is running. This session allows controlled execution of the generated processor. Execution takes place in the current working directory, and all of the source files of the generated compiler are made available. `dbx` must be used with the `debug` parameter (see Section 4.9 [debug], page 15).

Alternatively, `gdb` can be used to use the GNU debugger `gdb` instead of `dbx`. Both `dbx` and `gdb` check the environment variable `ELI_DEBUGGER`. If `ELI_DEBUGGER` is set, then its value is taken as the name of the debugger to be used. The debugger whose name is the value of `ELI_DEBUGGER` must accept the same parameters as either the Berkeley debugger dbx or the GNU debugger gdb. After setting `ELI_DEBUGGER`, use either `dbx` or `gdb` to activate your program, depending upon which set of parameters is appropriate.

An existing core file can be supplied by specifying the `core` parameter.

Every request for `dbx` or `gdb` re-executes the generated processor, regardless of whether anything on which the execution depends has changed. No other products can be derived from `dbx` or `gdb`. In particular, it is not possible to obtain warning messages or error reports by appending `:warning` or `:error` (see Section 3.1 [warning], page 11) to this derivation. If a warning message or error report is generated, Eli will specify the derivation step for which warning or abort status was set. The target of the derivation should be changed to this step, followed by `:warning` or `:error`, and the modified request submitted to Eli.

4.6 Monitoring — Monitor a program at the specification level

```
:mon
:mongdb
```

`mon` starts an interactive session with the Noosa monitoring system (see *Monitoring Reference Manual*). This session allows execution monitoring of the generated processor at the level of the specifications used to generate it. This concentration on the specification level contrasts with the use of debuggers to monitor the processor's execution in terms of its source code (see Section 4.5 [Debugging], page 14).

`mongdb` allows specification-level monitoring to be mixed with source-level debugging using gdb (see Section 4.5 [Debugging], page 14). When using this mode of monitoring, the source-level debugger has control of the executing program and runs as a child of the monitoring system. Consequently, the monitoring system is inactive whenever the debugger is at its prompt level. These derivations can be affected by the setting of the `ELI_DEBUGGER` environment variable. If this variable is set, it is assumed to be a gdb-compatible debugger. Thus you can use other debuggers that support the same command-line options. For example, graphical debuggers such as ddd can be used.

The `mon` and `mongdb` products imply the `monitor` parameter (see Section 4.7 [monitor], page 15) which causes the system to produce a processor that includes the monitoring code. The `arg` parameter (see Section 4.8 [arg], page 15) can be used to specify command-line arguments to the program being monitored.

The following two derivations have the same effect:

```
sets.specs +monitor +arg=(input) :mon
sets.specs +arg=(input) :mon
```

4.7 monitor — Request monitoring support

```
+monitor
```

If the `monitor` parameter is used, Eli includes monitoring support in the generated processor. Processors containing monitoring support can be monitored at the specification level using the Noosa monitoring environment (see *Execution Monitoring Reference*) via the `mon`, or `mongdb` products (see Section 4.6 [Monitoring], page 14), which automatically imply the `monitor` parameter. For `mongdb` to be useful, the `debug` parameter must also be used (see Section 4.9 [debug], page 15).

4.8 arg — Supply Command Line Parameters

```
+arg='item ...'
+arg=(file)
```

The specified items are supplied as command line parameters to the generated processor when it is executed by `mon` or `mongdb`. An item is any sequence of non-white-space characters other than apostrophes. Items are separated by white space. Any number of items may be specified.

4.9 debug — Request debugging information in object files

```
+debug
```

If the `debug` parameter is used, all compilations are given the `-g` flag. This causes the compilers to provide additional information for the source-level debuggers `dbx(1)` and `gdb`. See Section 4.5 [Debugging], page 14. In conjunction with the `monitor` parameter (see Section 4.7 [monitor], page 15), `debug` enables the use of `mongdb`. See Section 4.6 [Monitoring], page 14.

4.10 printtokens — Request Token Printing Code

```
+printtokens
```

If `printtokens` is specified, the generated processor will print the source text coordinates, internal code and intrinsic value for each basic symbol as it is read. This listing is useful to verify that the lexical analysis specification is correct, and also to obtain a frequency distribution of the kinds of basic symbols appearing in a program.

5 Producing Formatted Documents

Eli supports two mechanisms for producing formatted documents: *Texinfo* and *FunnelWeb*. Both on-line hypertext and printed documents can be produced from Texinfo (type-`tnf`)files; arbitrary specification files and printed documents can be produced from FunnelWeb (type-`fw`) files. In this chapter we consider only the production of printed documents.

5.1 ps — PostScript file

```
:ps
```

A file containing the result of formatting a type-`tnf` file.

5.2 dvi — Device-independent TeX typesetter file

```
:dvi
```

A file containing the result of formatting a type-`tnf` file. This file cannot be printed directly, but must be used as input to some typesetter-dependent program.

5.3 fwTex — TeX input file

```
:fwTex
```

A file suitable for input to TeX, containing the result of *weaving* a type-`fw` file:

```
doc.fw :fwTex > doc.tex
```

Note that `fwTex` can *not* be applied to a type-`specs` file.

If the documentation text of the type-`fw` file contains TeX or LaTeX commands, it should also contain the following FunnelWeb typesetter pragma:

```
@p typesetter = tex
```

This pragma is not required if the documentation text contains no TeX or LaTeX commands, but including it will have no effect. If no typesetter pragma is given, any TeX and LaTeX commands in the documentation will simply appear in the output as they stand in the input.

5.4 fwHtml — HTML file

```
:fwHtml
```

An HTML containing the result of *weaving* a type-`fw` file:

```
doc.fw :fwHtml > doc.html
```

Note that `fwHtml` can *not* be applied to a type-`specs` file.

The documentation text of the type-`fw` file must contain the following FunnelWeb typesetter pragma:

```
@p typesetter = html
```

6 Obtaining Information About the Derivation

Eli derives the requested product via a sequence of steps. Like any expert system, it is capable of providing information about those steps. Since one of the major goals of Eli is to hide the steps required to derive a particular product, thus reducing the cognitive load on the user, it does not automatically provide this information. By setting the `LogLevel` variable, a user can control the amount of feedback that Eli provides about the derivation as it is being carried out (see Section "Hints on Session Management" in *Guide for New Eli Users*). This feedback only provides information about *what* is happening, not *why*. Sometimes it is important to discover things like the names of intermediate products, what objects a given object depends upon, and what objects a given object influences. That information must be accessed via specific requests.

Eli does *not* bring an object up to date before applying any of the derivations discussed in this chapter. If you wish to execute one of these derivations on an up-to-date object, you must first bring that object up to date with an explicit request.

6.1 inputs — Objects on which a Given Object Depends

```
!:inputs
```

A list of the objects on which the given object directly depends. The number used to name the file in the cache and the Eli derivation is given for each object.

6.2 outputs — Objects Depending on a Given Object

```
!:outputs
```

A list of the objects that directly depend on the given object. The number used to name the file in the cache and the Eli derivation is given for each object.

6.3 test — Check Whether an Object has been Modified

```
!:test
```

The given object is checked to see whether its last modification time agrees with the value held by Eli. This product is used when you have modified an object by some external means, and wish to inform Eli of that modification. For example, suppose that Eli complains that a particular object needed to satisfy the current request is unavailable. You supply the object (say its name is `pascal.lido`) by copying it from some other directory. The following request will inform Eli of your action:

```
pascal.lido!:test
```

6.4 redo — Ask Eli to Carry Out a Derivation

```
!:redo
```

Tells Eli that a particular step in a derivation should be recomputed when it is next requested, even if the that step has already been computed and inputs to that step have not changed.

This utility is useful when a transient error in a derivation step occurs, but the tool that implements the derivation step did not recognize it as a transient error:

```
    sets.specs :level_6_specs !:redo
```

The next time any derivation requires the object `sets.specs :level_6_specs`, it will be recomputed.

Index

-

-g flag 15

‘

‘‘unavailable’’ error report 19

A

abort 11
abstree 7
absyntax 7
altering files 11
arg 15

B

basic symbol 16
BOTTOMUP 7

C

C compilation 4, 5
case-insensitivity 4
cc 5
cc_flags 4
character strings 8
cmd 13
COLA 5
command line parameters 15
constructing generic modules 8
consyntax 7
controlled execution 14
core 14

D

dbx 14
dbxtool 14
debug 15
debugging 14
define 4
dependence among products 19
diagnostic products 11
directives 5
domain-specific modules 8
dvi 17

E

edit command 11
ELI_DEBUGGER 15
ELI_DEBUGGER. 14
error 11
error report 11
exe 3
executable program 3
execution monitoring 14
expert system 19
ExpInfo 9
explanation of reports 11
external modifications, informing Eli 19

F

feedback 19
fold 4
formatted file 17
FunnelWeb 17
fwGen 4
fwHtml 17
fwTex 17

G

gdb 14
GDB 14
gencode 12
generated files 12
generation, parser 12
generic modules, naming instances 10
generic modules, user-constructed 8
GNU debugger 14

H

header file 3
help 11
hypertext 11

I

identifier output 4
ignore 5
include 5
inconsistencies 11
inputs 19
inst 8
instance 10
instantiation 8
interactive 11, 14
intermediate products 19
internal code 16
intrinsic value 16
invoking type-gnrc scripts 8

K

keyword 8
kwd 8

L

last modification 19
letters, case of 4
library 8
library routine 3
LogLevel 19
lower case 4

M

main program, omitting 5
makefile 3
mkidn 4
modification, checking for 19
mon 14, 15
mongdb 14, 15
monitor 15
monitoring 14, 15

N

name analysis 8
naming instances of generic modules 10
nomain 5
Noosa 14, 15

O

omitting the main program 5
omitting the parser 5
OptimInfo 10
OrdInfo 9
output 13
output of identifiers 4
outputs 19

P

parameter 1
parameters, command line 15
parsable 12
parser 5
parser generation 12
parser generator selection 5
parser, omitting 5
pgram 7
PGS 5
printtokens 16
product 1
products, relationships among 19
ps 17

R

redo 19
referto 10
run 13
run with typical input data 13

S

scanner 8
scripts, invocation via inst 8
showFe 10
showMe 10
so 3
source 3
source language file 3
source text coordinates 16
status 11
stdout 13
STOPAFTERBADPARSE 4

T

test 19
Texinfo 17
type-fw 17
type-gnrc files 8

U

upper case 4

W

warning 11
warning message 11
weave 17