FAQ: Frequently Asked Questions and Answers

Eli Maintenance Team

University of Paderborn
D-33098 Paderborn
FRG

$Revision: 3.3 $

This document answers frequently asked questions about Eli. The first chapter refers to the Eli Installation available locally; the others are cover more general concerns.

The document will be updated periodically as questions arise and are answered.

1 Where to Find Eli Locally

Eli Version 4.1 is installed in directory '/usr/local' on Computer Science machines running Solaris. If you have '/usr/local/bin' in your path, then the commands `eli` will invoke Eli.

Eli will use the cache directory specified by the `-c` parameter of the `eli` command if one is given. If the `-c` parameter is omitted, and the environment variable `ELICACHE` is defined, that variable specifies the cache directory to be used. Otherwise the cache directory is assumed to be 'ELICACHE' in the current directory.

Another startup-script to invoke Eli is 'Eli', also installed in '/usr/local/bin'. This script distinguishes creation of a cache and its usage and is designed primarily for use of developpers that want to replace certain components of Eli by modified variants. A usage of this script can be requested by calling 'Eli' with the parameter '-h'.

2 How to Interact with Eli

2.1 Why is my cache so big?

An Eli cache contains all of the intermediate information needed to build the products you have requested since that cache was created or reset. If you continually request the same product, based on the same set of specifications, perhaps changing some of the specification content between requests, the cache size will first grow and then stabilize.

When you switch to another set of specifications, the old set and the intermediate information based upon them remains in the cache. Thus as you do a sequence of different projects, the cache grows because it retains all of the information for earlier projects in addition to the information needed for the current project.

All of the information held in the cache was derived by Eli from something that you provided. Thus deleting that information will not affect the correctness of future derivations; it will only increase the time required because the information must be re-derived.

The command `eli -r` clears the cache of information before invoking Eli. If you use this command on the first invocation of Eli for a new project, information belonging to old projects will be deleted and the cache will subsequently contain only information derived in connection with this project.

In the case of a shared machine where disk space is limited, we recommend that you clear the cache when you terminate your session and do not expect to start another session soon. This can be done via the command `eli -r ''`, which clears the cache and then invokes Eli to do nothing. Eli will terminate immediately.

2.2 How do I get new or improved components?

From time to time there will be upgrades to Eli components. These upgrades do not affect existing caches, which continue to use the version of the component that was in effect at the time they were created or last upgraded.

The command `eli -R` clears the cache of information and upgrades to the latest set of components before invoking Eli.

2.3 Is the file '.odinrc' still read?

> I can't find any description of this file in the help system.
> How can I now tell Eli which editor I want to use?

Eli doesn't read the file '.odinrc' any more. The information that was previously obtained from '.odinrc' is now provided by environment variables. The EDITOR environment variable specifies the editor Eli should use in response to the < command.

Each of the other Odin variables listed by the Eli ?= request is associated with an environment variable whose name is ODIN followed by the upper-case name of the Odin variable. For example, the default value of the Odin variable LogLevel will be the value of the environment variable ODINLOGLEVEL (if that environment variable is defined).

For additional information about Odin variables, see Section 'Variables' in User Interface Manual.

2.4 Eli didn't notice that I changed my input file!

If you alter a file in an editor session started by the < command, Eli will check whether the file on which the request was made was changed during that session. Other checks are controlled by the Odin variable VerifyLevel:

```
-> VerifyLevel=?
0: No source file change verification.
1: Check for changes to source files at the beginning of each session.
2: Check for changes to source files before each interactive command.
```

The default value of VerifyLevel is 2, but if the environment variable ODINVERIFYLEVEL is defined then that value will be used as the default. You can check the current value of VerifyLevel by querying it explicitly, and you can also set it interactively. For more details, see Section 'Variables' in User Interface Manual.

If VerifyLevel is not 2, you can inform Eli that a specific file 'foo.fw' has changed by requesting the :test utility:

```
-> foo.fw !:test
```

If you omit the file name, Eli will check all files for changes. (This is equivalent to having `VerifyLevel` set to 2.)

3 Problems Reported by Implementors

3.1 Eli reports `Remote IPC not available on this host`

This error report indicates that Eli was unable to use the inter-process communication mechanism effectively. It is usually seen on Linux systems that are not connected to a network.

Eli's default inter-process communication mechanism is TCP/IP. If TCP/IP is not available, the environment variable `ODIN_LOCALIPC` can be set and Unix domain sockets will be used instead. Setting `ODIN_LOCALIPC` to 1 usually avoids the error report.

3.2 Eli reports `Cannot connect to Odin server`

This means that there is a problem with TCP/IP socket communication. It can occur in two situations: on initial installation of Eli and after some kind of abnormal termination. If it occurs on initial installation, you need to set the environment variable ODIN_LOCALIPC to 1 before invoking Eli. If it occurs after an abnormal termination, simply resetting the cache by means of the command "eli -r" should clear things up.

3.3 Requirements of Eli

> Does Eli require an installed gcc?
> Does Eli require an installed Tcl/Tk?
> Does Eli require an installed TeX?
> Does Eli require an installed Funnelweb?
> Does Eli require an installed Odin?
> Eli is only available on Unix machines!

Eli doesn't really require support tools beyond standard utilities you'll find on any Unix machine. Tcl/Tk isn't strictly required, but you miss out on a few features if you don't have it (the simplest info browser is implemented using Tcl/Tk as well as Eli's execution monitoring/debugging facilities). Most everything else is included in the distribution (including FunnelWeb and Odin). TeX is only required if you want to be able to format FunnelWeb documents using TeX. Eli doesn't rely on gcc at all. It uses whatever C compiler you have on your system.

While Eli itself has not been ported to systems other than Unix, the code that Eli generates is highly portable C code that can be compiled anywhere. (The generated C code can be compiled by a number of C++ compilers as well.) We have reports of people successfully compiling Eli generated code on Windows platforms.

3.4 What restrictions are placed on the Usage of Eli?

> Is professional support available and what is the licence agreement?

Eli is distributed under the terms of the GNU Public License. Code generated from Eli can for the most part be used without restriction.

We do not sell support, although we try to answer questions from users rapidly.

3.5 Configuration does not find the X-Window-System.

> Running ./configure on Slackware Linux 2.0.30 in XTerm (with DISPLAY=0:0)
> results in "checking for X... no", although X works ok for other
> applications.

The problem is not in running X applications, but in being able to compile X applications. You need the appropriate development include files and libraries.

If your X includes and libraries don't reside in the normal places, you may have to give configure some help. For example, you could invoke configure like this:

```
% ./configure --x-includes=/'somedir'/include --x-libraries=/'somedir'/lib
```

where '`somedir`' was the directory of your X installation.

If you believe your X installation is in a normal place, you could test out the following. To test the autoconf detection mechanism in isolation, you can do the following if you have autoconf installed:

1. Go to some temporary directory and create a file called configure.in with the following contents:

```
AC_INIT(configure.in)
AC_PATH_X
AC_OUTPUT()
```

2. Run the command "autoconf". This will create a configure script in the directory.
3. Run configure and see if it finds the X includes and libraries.

4 Using Specifications Developed with Earlier Versions

Further hints for updates of specifications can be found in New Features of Eli Version 4.1, and in New Features of Eli Version 4.0.

4.1 Eli doesn't find module include files any more

```
    4 |#include "envmod.h"
(test.c) cannot find include file: "envmod.h"
```

Eli no longer automatically extracts all modules that are mentioned somewhere in `#include` files. For example, in order to add the `envmod` module to a specification, some `.specs` file must contain the line `$/Name/envmod.specs`. (`$` in this context is a shorthand for the package directory in the cache currently being used.)

4.2 Migrating the usage of Module `IdPtg`

> Am I correct in noting that the link between the key table and the
> string table containing non-literal strings was possible via the
> following (in Eli3)

```
> ATTR Key:  DefTableKey;
> SYMBOL xName INHERITS IdDefNest END;
> SYMBOL xIdent INHERITS IdPtg END;
>
> RULE Name: xName ::=  xIdent
> COMPUTE
>         xName.Ptg = xIdent.Ptg;
>         xName.Sym = xIdent.Sym;
> END;
```

There are two different tasks:

Consistent renaming for name analysis (that's what Key's are for)

Output text generation with Ptg (based on the string representation)

There is Module Library support for both tasks. In your example you use

`IdDefNest` for name analysis

`IdPtg` for output text generation for identifiers

`IdPtg` computes an inherited attribute `Ptg` containing the PTG representation of the identifier to which `IdPtg` is inherited. `IdPtg` could be inherited to terminals (as you do above) in earlier Eli versions. The precondition for the application of `IdPtg` is that the symbol to which `IdPtg` is inherited provides an attribute named `Sym` containing the string table index of the associated identifier.

Since Eli Version 4.x, inheritance to terminals is no longer possible. That is the only difference for your application. So you simply have to move your PTG computation one level up to the symbol `xName`. This should give you:

```
SYMBOL xName INHERITS IdPtg END;

RULE Name: xName ::=  xIdent
COMPUTE
     xName.Sym = xIdent.Sym;
END;
```

which even looks a bit simpler than the 3.8 solution.

4.3 Why does `C_STRING_LIT` not use `c_mkstr`?

> I'm surprised that C_STRING_LIT now uses mkstr instead of c_mkstr.
> What's the reason?

`c_mkstr` had the problem that if the string in the source text contained a null character, that string was silently truncated:

```
"abc\0def" became "abc"
```

This is a consequence of the string storage module implementation, and could only be avoided by placing a high cost on *all* string operations. To avoid the problem, `c_mkstr` was changed to report an error if it found `"\0"` in a string. The changed `c_mkstr` no longer implements C strings, so it could not be used as the processor for `C_STRING_LIT`.

5 How to solve common problems in Eli

5.1 Is there Support for Intermediate Code in Eli?

> Does Eli support the generation of intermediate code and if so, what
> type of code and in which way?

In general, Eli supports generation of intermediate code in the same way that it supports the generation of any kind of code: You write a PTG specification of the language to be output and invoke the routines to build up your output.

If you want to specify the intermediate language as a tree and perform some attribute computation over it without writing it out and reading it back in, then you can specify the tree in LIDO. Eli produces Mk functions from a LIDO specification much the same way as it produces PTG functions from a specification. You invoke the Mk functions to build your tree and then paste it into the original tree built by the parser. See Section 'Computed Subtrees' in LIDO - Reference Manual, for details. You can see an example using an intermediate language appropriate to the Spim simulator for the MIPS at '`http://ece-www.colorado.edu/~ecen4553/HW/ctgt.html`'.

If you want to target an existing intermediate representation that is implemented as a class library (like Suif, '`http://suif.stanford.edu/suif/suif.html`', for example), then you make the class library available as a part of your specification (usually via a .a file) and invoke its methods directly from your translation specification just like you would invoke PTG or Mk functions.

5.2 C++

> Is there an ELI-generated compiler available for C++?

There is none that I know of. One reason is that `C++` seems to be defined not by a standard but by a collection of front ends. Particular users are partial to particular front ends, and are not interested in a compiler that actually conforms to either the ARM or the draft ANSI standard. This situation might change, but in the interim there seems to be little point in building an Eli specification.

Compounding the problem is the fact that C++ has a number of essential ambiguities that make it very difficult to specify formally. For example, if something looks like both a declaration and an expression then it's a declaration. How do you express that formally?

5.3 How does recognition of delimiters work in Eli?

> I want to use ':=' in the grammar to represent three notations
> ':=', '=', and ':-'.

This could be done as follows (for an explanation of all of the relevant constructs, see Section 'Literal Symbols' in Lexical Analysis):

```
@O@<foo.con@>==@{
S: 'A' ':=' 'B' .
@}

@O@<foo.delit@>==@{
$:=     ColonEqual
@}

@O@<foo.gla@>==@{
        $:=     [mkassign]
        $=      [mkassign]
        $:-     [mkassign]
@}

@O@<foo.c@>==@{
#include "litcode.h"
#include "eliproto.h"

void
#if PROTO_OK
mkassign(char *c, int l, int *t, int *s)
#else
mkassign(c, l, t, s) char *c; int l, *t; int *s;
#endif
{ *t = ColonEqual; }
@}
```

> For Pascal, `:kwd` has to be applied to the file that contains
> nothing else than the specification of `Name: PASCAL_IDENTIFIER`
> Is that the only possible application of `:kwd`?

The `:kwd` derivation is used to specify sets of literal terminals appearing in a grammar that should be recognized by looking them up in the identifier table rather than by incorporating them into the scanner's finite state machine. Mixed-case keywords are the most common use of `:kwd`, but it can also be used simply to cut down the size of the scanner tables at some cost in execution time.

5.4 How can I use external libraries with Eli?

Use `+lib='mylib'` to add the '`-lmylib`'' option to the link. `+lib_sp` can be used to supply a directory for the library (the `-L` option in the link).

You can also use the mechanism used by Eli internally, although it isn't really pretty. You can add a '`.libs`' file to your specification that has something of the form:

```
/ +lib_sp=(libdir) +lib='libname' :library_flags
```

> Is there also a way to tell Eli an include path for > header files?

There is a parameter called +cc_flags with which you can provide the `-I` option to the compilation.

6 Index