

Eli System Administration Guide

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

Copyright © 2002, 2009 The Regents of the University of Colorado.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation.

Table of Contents

1	Package Management	3
1.1	Versioned packages	3
1.2	Package collections	3
2	Cache Mangement	5
2.1	Name of the cache	5
2.2	Packages in the cache	6
2.3	The <code>odin</code> command	6
3	Installation	7
3.1	Installing Eli	7
3.2	Installing additional packages	8
4	Examples	9
4.1	Using a single additional package	9
4.2	Maintaining distributed packages	9
4.3	Developing additional packages	10
5	Problem Reporting	11
	Index	13

Eli embodies an understanding of how to generate text processors from specifications of their behavior. This understanding has been developed over many years by a host of people, and is expressed in a large number of tools and workflow patterns. The generation process creates many intermediate artifacts, which must be passed among the tools. A change in a specification may make it necessary to rebuild some of these artifacts by executing particular tools.

Odin is a program capable of managing a collection of artifacts and tools to provide a product that is up-to-date with respect to all of the raw material on which it is based. The tools and associated workflow patterns are described to *Odin* by a *derivation graph*. Modularity in the derivation graph specification is achieved by gathering related tools and the workflow patterns connecting them into *packages*.

In order to use *Odin* to derive specific products from specific raw material, a *cache* must be available. A cache is represented in the computer by a single directory. It contains a derivation graph built from some set of packages, plus all of the artifacts created when deriving products defined by that derivation graph. Each user may establish an arbitrary number of caches, and each cache may be shared by an arbitrary number of users.

1 Package Management

Eli is a collection of packages. Each package is represented in the computer by a directory. A package directory must contain a file whose name is the name of the directory with the suffix `dg`. That file specifies the derivation graph for the package. For example, the package defining the tools and workflow for constructing a scanner is represented by a directory named `'gla'`, and contains a derivation graph specification file named `gla.dg`. Some packages contain no tools, and therefore their derivation graphs are specified by empty files.

1.1 Versioned packages

The packages in the Eli distribution directory are *versioned*. A file named `'version'` is included in each package directory. That file consists of a single line that is a three-part version number. It is updated whenever a change to the package affects the behavior of that package; repairing defects in a package's tools does not require a version change.

If the package's workflow is changed without altering its functionality, the third component of the version number is incremented. When the local functionality is changed, the second component of the version is incremented and the third zeroed; when the change is visible to other packages, the first component is incremented and the others zeroed.

Versioned packages allow each user to decide on a project-by-project basis when to upgrade (see [Chapter 3 \[Installation\]](#), page 7).

1.2 Package collections

When the understanding of a process is embodied in a *collection* of packages, as in the case of Eli, all of the directories representing those packages are stored as subdirectories of a single directory. In the Eli distribution there are two such directories, `'Eli/pkg'` and `'Odin/pkg'`. `'Eli/pkg'` embodies the understanding of text processor generation, whereas `'Odin/pkg'` embodies the understanding of more general processes such as linking object files, executing a command line, and formatting a document.

In addition to the package directories themselves, each collection's directory contains a file named `'PKGLST'`. `'PKGLST'` is a text file, each line of which is the name of one of the package directories. Thus `'Eli/pkg/PKGLST'` contains the line `'gla'` to indicate that the `'gla'` directory represents a package.

`'PKGLST'` allows a collection's directory to contain additional files and non-package directories. The names of these files and directories will not appear in `'PKGLST'`, and therefore they will not be mistaken for packages in the collection. It is also possible to temporarily exclude packages from the collection simply by removing their names from `'PKGLST'`; the package directories themselves remain unchanged (see [Section 4.2 \[Maintaining distributed packages\]](#), page 9).

2 Cache Management

Each cache directory contains a derivation graph built from some set of packages, plus all of the artifacts created when providing products defined by that derivation graph. A user may want to have more than one cache for several reasons:

- A cache used on a completed project can be deleted without losing artifacts associated with ongoing projects.
- A user may collaborate with different people on different projects, each having its own cache.
- Different derivation graphs may be appropriate for different projects.

In order to work with more than one cache, the user must be able to refer to each by a unique name; in order to have caches with different derivation graphs, the user must be able to control the set of packages from which a cache is built.

2.1 Name of the cache

When the `eli` command is executed, it specifies the name of a cache directory as follows:

- If the `eli` command has the parameter `-c 'dir'`, the name of the cache directory is `'dir'`. `'dir'` must be an absolute path name.
- Otherwise the name of the cache directory is `$ODIN/$ODINVIEW`. `$ODIN` must be an absolute path name.
 - If the environment variable `ODIN` is not set, the first component of the cache directory name is `$HOME/.ODIN`.
 - If the environment variable `ODINVIEW` is not set, the second component of the cache directory name is the result of the command `uname -n` (`uname -n` normally returns the name of the machine executing the command). If the command `uname -n` returns an empty string, the second component of the cache directory name is the string `local`.

If the specified cache directory does not exist, a new cache is built and represented by a new directory with the specified name.

The environment variable `ODINVIEW` is commonly used in a classroom or laboratory having a number of identical computers that share a common file system. Since the computers share a common file system, every execution of the `eli` command will have the same value of `ODINVIEW`. As noted above, if `ODINVIEW` is not set then the default cache name will incorporate the name of the particular computer on which it was created. Therefore a user will have a different cache for each machine on which they have run the command `eli` without a `-c` parameter. This can be avoided if the user's startup script defines `ODINVIEW` as a constant value.

A different situation arises with a number of clusters of machines of different architectures, all sharing a common file system. Each cache is tied to a specific architecture, so there must be a distinct default cache name for each cluster if the `eli` command is to be used without a `-c` parameter. The user's startup script should therefore define `ODINVIEW` as the result of a command such as `arch`, which returns a string unique to the architecture of the machine running the command, rather than a constant value.

2.2 Packages in the cache

The set of packages from which a cache is built is determined either at the time that an `eli` command with the `-R` parameter is executed on that cache, or at the time a non-existent cache is specified. In either case, the packages making up the set are drawn from four sources, in order:

1. Parameters of the `eli` command, each consisting of `-p` followed by the absolute path name of a directory. An `eli` command may have an arbitrary number of such parameters, and they are considered in order from left to right.
2. The colon-separated list of absolute path names in the `ODINPATH` environment variable.
3. The standard Eli packages.
4. The standard Odin packages.

Each of the directories may represent a single package or a collection of packages with a ‘`PKGLST`’ file (see [Chapter 1 \[Package Management\]](#), page 3).

The order in which the directories are considered is important, because only the first package with a given name is entered into the set from which the cache is built. Thus it is possible to override standard Eli and Odin packages by providing replacements whose names are the same as the names of the packages to be replaced (see [Section 4.2 \[Maintaining distributed packages\]](#), page 9).

If an `eli` command specifying an existing cache is executed without the `-R` parameter, then any `-p` parameters are ignored. The set of packages available is the set specified when that cache was originally created, or when it was last specified by an `eli` command with the `-R` parameter.

2.3 The `odin` command

Odin can be invoked directly with the `odin` command, although there is no need to do so except to create a cache without using any of the standard Eli packages.

Unless the `-R` parameter is given, the behavior of the `odin` and `eli` commands are identical when they specify an existing cache. If the specified cache does not exist, or if the `-R` parameter is given, then the sets of packages created by the two commands are different. When an `odin` command is executed in either of those cases, the packages making up the set are drawn from two sources, in order:

1. The colon-separated list of absolute path names in the `ODINPATH` environment variable.
2. The standard Odin packages.

This differs from the behavior of the `eli` command (see [Section 2.2 \[Packages in the cache\]](#), page 6). The `odin` command does not accept `-p` parameters and does not consider the standard Eli packages. The effect of `-p` parameters is obtained by building an appropriate colon-separated list and making it the value of the environment variable `ODINPATH` before executing the `odin` command.

3 Installation

Eli is distributed as source text, which must be configured and compiled before the system is usable. An extra (optional) *installation* step creates a library that takes advantage of versioned packages (see [Section 1.1 \[Versioned packages\], page 3](#)). There are two reasons for installing Eli:

- Existing packages are not overwritten by new versions, leaving current caches unchanged until they are reset by a `-R` parameter on the `eli` command (see [Section 2.2 \[Packages in the cache\], page 6](#)). Thus the timing of an upgrade can be tailored to individual projects.
- The distribution directory contains a lot of “scaffolding” that is used to build tools, but is not required to execute Eli. By installing the system and then deleting the distribution directory, you can eliminate all of this extraneous material.

3.1 Installing Eli

As described in the Eli distribution’s ‘README’ file, you can build the system by running `configure` followed by `make`. A subsequent invocation of `make install` will install Eli in the directory that you specified by the `--prefix` parameter when you configured the distribution. Typical settings are `--prefix=/usr/local` and `--prefix=/opt/eli`. (Your home directory is used if no `--prefix` parameter is supplied to the `configure` command.)

Installation will add the following files and directories to the specified directory (the ‘`man`’, ‘`bin`’, and ‘`lib`’ sub-directories will be created by the installation process if they don’t already exist):

```
‘man/man1/odin.1’
    Man page for Odin.

‘bin/odin’
    Command script

‘bin/eli’
    Command script

‘lib/Eli’
    The library of Eli packages.

‘lib/Odin’
    The library of Odin packages.
```

Each package directory in these libraries contains a subdirectory for each distinct version of that package, plus meta-information defining the latest version. When a cache is built from a set of packages, the meta-information is used to establish links to the latest version in each set. The packages themselves are not copied to the cache.

When a new version of Eli is installed, the following changes are made to the libraries:

1. If the ‘`version`’ file of a package has not changed, but some of the other files in that package have changed, the new files replace the old in the appropriate subdirectory.
2. Otherwise, a new subdirectory of the package directory in the library is created. Its name is the content of the ‘`version`’ file. The necessary files are copied into this new directory, and the meta-information is updated to indicate that the new directory is the latest.

Since each cache links to a specific version subdirectory of the package, changes will only affect it if the package version has not changed. If the package version has changed, the previous subdirectory remains undisturbed. Thus an existing cache will be affected only by defect repairs that do not alter any visible behavior (see [Section 1.1 \[Versioned packages\]](#), [page 3](#)).

3.2 Installing additional packages

Locally-developed packages can be added to existing libraries, or new libraries can be created to hold them. A new library is typically created as a new subdirectory of the directory specified by the `--prefix` parameter when Eli was configured.

For example, you might create the directory `/opt/eli/Packages` to hold additional packages that should be a part of the set from which every cache was built. If you then set the environment variable `ODINPATH` to `/opt/eli/Packages`, the command `eli -R` would create a cache from those packages, the standard Eli packages, and the standard Odin packages (see [Section 2.2 \[Packages in the cache\]](#), [page 6](#)).

You can add a package directory or package collection directory `dir` to a library directory `lib` by the following Eli request:

```
-> dir +d_dest=(lib) :installpkg
```

Directory `dir` must be built (if necessary) and all irrelevant files (e.g. `Makefile`, `README`, `CVS`, source files) removed before this request is made. If `dir` is a package directory, it must contain `*.dg` and `version` files; if it is a package collection directory, it must contain a `PKGLST` file (see [Section 1.1 \[Versioned packages\]](#), [page 3](#)).

Packages installed via `:installpkg` behave exactly like the standard versioned packages: repairs will be effective in existing caches but changes in behavior will not (see [Section 3.1 \[Installing Eli\]](#), [page 7](#)).

4 Examples

Each example uses a cache directory with an explicit name given by a `-c` parameter. If you don't generally work on more than one project at a time, then you may wish to omit the `-c` parameter and use the same cache for every project (see [Section 2.1 \[Name of the cache\]](#), [page 5](#)).

4.1 Using a single additional package

The Eli packages understand LALR(1) grammars and how to use them to generate deterministic parsers. Unfortunately, some languages are very difficult to describe using such grammars.

A package named `bisonglr` is available from the Eli website. It uses the Bison parser generator's "GLR" facility to create parsers for grammars that cannot easily be rewritten to make them LALR(1). This package assumes that you have the Bison parser generator available in your path, under the name `bison`. No change in specifications is required; the package handles all of the necessary translations and integration.

Suppose that you have downloaded the `bisonglr` package and unpacked it to your home directory. Then the following command builds a cache called `GLRCACHE` in your home directory. Assuming that `$ODINPATH` is empty, `GLRCACHE` is built from the `bisonglr` package plus the standard Eli and Odin packages (see [Section 2.2 \[Packages in the cache\]](#), [page 6](#)).

```
eli -c $HOME/GLRCACHE -R -p $HOME/bisonglr
```

Note that absolute path names are used for both the cache directory and the package directory.

If you frequently worked on languages whose grammars required the `bisonglr` package, you could add it to the Eli package library by making the request:

```
-> $HOME/bisonglr +d_dest=(/opt/eli/lib/Eli) :installpkg
```

After making this request, the next time you build a cache the standard Eli packages will include `bisonglr`.

4.2 Maintaining distributed packages

The text for the Eli system is kept in a CVS repository. For the purposes of this example, assume that you have checked out a copy of the distribution directory '`Elidistrib`'. The name of your working directory will therefore be '`Elidistrib`'. The directory containing the standard Eli package collection is then '`Elidistrib/Eli/pkg`', and the directory containing the standard Odin package collection is '`Elidistrib/Odin/pkg`'.

You will use your working directory to develop and test changes to the standard packages, ultimately checking in the final version. During the development process, you want to run Eli with certain package directories from your working directory substituted for the corresponding directories in '`/opt/eli`'.

For example, suppose that your working directory is a subdirectory of your home directory and you need to debug the `gla` package. The following command will build the cache '`DBGCACHE`' in which the `gla` package from your working directory takes the place of the standard `gla` package:

```
eli -c $HOME/DBGCCACHE -R -p $HOME/Elidistrib/Eli/pkg/gla
```

If you need to work on several packages at once, a good strategy is to create a new package collection directory and populate it with symbolic links to package directories in your working directory:

```
mkdir $HOME/pkg
ln -s $HOME/Elidistrib/Eli/pkg/gla    $HOME/pkg
ln -s $HOME/Elidistrib/Eli/pkg/parser $HOME/pkg
ln -s $HOME/Elidistrib/Eli/pkg/pgs   $HOME/pkg
...
```

You then put the names of the packages you want to work on into ‘\$HOME/pkgPKGLST’ and build the debug cache with the following command:

```
eli -c $HOME/DBGCCACHE -R -p $HOME/pkg
```

Once you have finished debugging, you can leave the ‘pkg’ directory with its symbolic links in place. When you need to debug other packages, you add any new links and edit ‘PKGLST’ to name the packages on which you want to work. Links to currently-uninteresting packages remain for future debugging sessions, but because their names are not listed in ‘PKGLST’ they are ignored.

4.3 Developing additional packages

A package being developed requires a package directory. It is often useful to make that directory a subdirectory of the ‘\$HOME/pkg’ directory (see [Section 4.2 \[Maintaining distributed packages\], page 9](#)). It can then be used either singly or in combination with other packages being debugged.

All of the additional packages developed to date are kept in the **Packages** directory of the CVS repository. If you are developing an additional package for Eli, you might decide to check out the **Packages** directory and develop your package in a subdirectory of the resulting working directory. This is generally *not* a good idea. The reason is that you will ultimately want to use a CVS **import** command to add your package to the repository. That command expects a directory distinct from your working directory. A subsequent checkout with the **-d** parameter will add the imported directory to your working directory. If the package directory that you developed is a subdirectory of your working directory, it will have to be moved before the checkout takes place. Thus it is better develop the package in a directory (such as ‘\$HOME/pkg’) that is disjoint from your working directory.

5 Problem Reporting

Should you have a problem with Eli please e-mail a description to:

```
eli-project-users@lists.sourceforge.net
```

When reporting problems, please include the Eli version number (printed when Eli starts up, or available in the distribution in ‘Eli/pkg/version/version.dg’) and the output from running the shell command `uname -a` on your system. Also, please give *complete* output illustrating the problem. If necessary, please include relevant specification files.

In some cases it will also be helpful to identify the version of the software in question. Most distributed files have RCS tags embedded in them. Most executables are generated from sources that have an initialized variable containing version information:

```
char rcsid[] = "$Id: misc.c,v 1.2 88/11/30 12:57:39 bob Exp $";
```

It is easy to identify the source versions from which binaries have been generated by running the RCS `ident` utility on them. For example, running `ident` on ‘pkg/gla/gla_fe’ would yield something like the following:

```
gla_fe:
  $Id: driver.c,v 1.15 1996/03/22 19:28:48 kadhim Exp $
  $Id: dfltclp.c,v 1.8 1994/12/08 03:09:52 tony Exp $
  $Id: source.c,v 2.5 1995/01/16 02:19:21 waite Exp $
  ...
```

Sites that do not have `ident` can extract this information by one of the following commands:

```
Source      grep '\$Id:' 'file-name'
```

```
Executable strings 'file-name' | grep '\$Id:'
```


Index

C

cache 1
 cache name 5
 cache packages 6
 collections, of packages 3
 commands, eli 5
 commands, odin 6
 commands, uname 5
 common file system 5

E

eli command 5
 Eli package directory 7
 Eli, installing 7
 environment variable, *ODIN* 5
 environment variable, *ODINPATH* 6
 environment variable, *ODINVIEW* 5

F

file, 'PKGLST' 3, 6

I

installation 7
 installing Eli 7
 installing packages 8

N

name, cache 5

O

odin command 6
ODIN environment variable 5
 Odin package directory 7
ODINPATH environment variable 6
ODINVIEW environment variable 5

P

package 1
 package collections 3
 Packages 3
 packages, cache 6
 packages, installing 8
 packages, versioned 3
 'PKGLST' file 3, 6
 problem reporting 11

U

uname command 5

V

versioned packages 3

