arch

The arch Revision Control System

visit the Hackerlab at www.regexps.com

Thomas Lord

1 Introducing arch

`arch` is a source code management, revision control, and configuration management tool.

1.1 Advantages of arch

What makes `arch` better than other revision control systems?

There are minor advantages, and a a few major advantages.

The minor advantages are things like: regular, clean, interfaces; small code size; on-line help; automated ChangeLog generation; features for browsing change sets with a web browser; software tools architecture. Some of these are arguably matters of taste – worth mentioning but not arguing for.

There are at least six major advantages to arch: of revisions *a file tree library of revisions*, *renaming*, *distributed repositories*, easy operation *robust and easy operation*, *configurations*, and merging *sophisticated branching and merging*.

"a file tree library of revisions" means that all revisions can be made accessible in a (space efficient) library of revision trees. That means that you can use ordinary tools like `diff`, `find`, and `grep` to explore past revisions and revisions on other branches.

"renaming" means that you can rename files and directories and `arch` keeps up just fine: it rearranges the patches between revisions in the corresponding way. And it's convenient: you can use `arch` in a mode that doesn't require you to run `add`, `remove`, or `move` commands every time you add, delete, or rename a file or directory.

"robust and easy operation" means that `arch` repository transactions have ACID properties and repositories are stored as ordinary unix files. Commits may be concurrent with reads and are atomic. There is no overhead from having to administer a relational or hash-table database. The transient locks held by `arch` never become permanently wedged (requiring by-hand repair): they can be broken remotely to recover from interrupted commands.

"Distributed repositories" means that you never need write access to a repository in order to start branches from projects stored in that repository. You can store branches in any repository. That means every programmer can have private repositories for day-to-day work or as scratch areas for working out complex merges. Every sub-group working on a project can have their own repository. If two groups want to fork a project, but still loosely cooperate, they can each have their own trunk in their own repository – selectively merging changes between the two trunks. Without wishing to put too much hype on it, as far as `arch` is concerned, there is exactly one repository database for the entire world – we all share it.

"configurations" are a mechanism for defining "meta-projects" that are a combination of multiple "sub-projects". For every meta-project, you need to be able to define *configuration rules*, which explain which revisions from what branches to check out to get a particular revision of the meta-project. `arch` provides a flexible mechanism for that, allowing you to pick and choose pieces from various branches and repositories with varying degrees of specificity.

"Sophisticated branching and merging" has to do with arch's support for asynchronous development on branches, coordinated by a shared trunk. Commonly, people arrange branches in a "star topology". There is a trunk in the middle, surrounded by branches, as in this (hypothetical) example:

```
patch-reviewer A                      patch-reviewer B
  (who merges patches                 (another patch reviewer)
   contributed by people             /
   who don't have write-access      /
   to gcc-main)                    /
        \                         /
         \                       /
           gcc-main (the trunk)
          /                    \
         /                      \
        /                        \
C 99 team                       Fortran team
  (who work on features          (who work on fortran)
   required by the new
   standard)
```

And we might imagine many more branches than are actually drawn here.

Ideally, each of the surrounding branches will sometimes be merged into `gcc-main`. And as `gcc-main` changes, it wll sometimes be merged back into the surrounding branches.

When two versions each merge with the other repeatedly, forming a merge without creating spurious conflicts is a tricky problem. (For a more detailed description of the problem, see Chapter 14 [Star Topology Branching and Merging], page 71.)

In CVS, the problem is solved by using tags and not generating conflicts for changes that appear to be redundant. Unfortunately, tags are expensive, work only within one repository, and worst of all, are complicated to use correctly. If you forget to create a tag at the right time (or worse, if your tagging operation is interleaved with other operations) you're hosed.

In `arch`, history sensitivity is taken to the next level. `arch` knows how handle back-and-forth merging in a star topology automatically.

This is an advance in revision control similar to the invention of lockless operation in CVS. With lockless operation, you have a star topology with an archived development path in the middle, surrounded not by branches, but by working copies. Programmers can commit from the working copies to the trunk or merge from the trunk into working copies repeatedly, and everything works beautifully. The trunk is used to syncronize the multiple working copies.

`arch` takes the next step. It allows those satellite working copies to be archived as long-lived branches. The trunk is used to syncronize the surrounding branches.

`arch` automates the merge operation (with the `larch star-merge`) command and gives you control over the precedence-ordering of change sets (which branch takes priority, and which has changes that are rejected if they conflict). You don't have to use tags, and you don't have to figure out the right revision arguments to pass to an `update` command.

Each satellite has its own log history. Each can be used for progress tracking. A satellite can be used to maintain an alternative distribution that tracks the trunk, but sometimes takes the lead with additional features.

In addition to star topology merging, `arch` provides some other fancy merging options too (e.g. see Chapter 17 [Multi-Branch Merging – The reconcile Command], page 81.)

1.2 Global Revision Control Done Right

The foundation of `arch` is two commands: `mkpatch` and `dopatch`.

`mkpatch` computes a patch set describing the difference between two trees. `dopatch` applies a patch set to a tree, gracefully handling the cases when a patch doesn't apply cleanly.

Conceptually, `mkpatch` is similar to `diff -r` and `dopatch` is similar to `patch`. Unlike `diff` and `patch`, though, `mkpatch` and `dopatch` can handle the addition or removal of files and directories, the renaming of files and directories, files which are symbolic links, changes to file permissions, and binary files.

An `arch` *repository* is a collection of full-source revisions, and patch sets. Brand new trees are represented as full-source revisions. Modified trees are stored as patch sets. Any revision can be reconstructed by retrieving a full-source revision, intermediate patch sets, and applying those patches.

`arch` repositories have globally unique names and every revision in a repository has an easy to understand, easy to type name. Putting the two names together, `arch` provides a global namespace for revisions.

On the basis of that global namespace, branches and merge operations can span repository boundaries. As far as `arch` is concerned, all of the repositories you can access over the Internet are integrated into one gigantic repository – seemlessly integrated.

In `arch`, there is no such thing as a "working directory". That is to say, there is no distinction between a tree that you download as a source distribution, and a tree that you check out from a repository. `arch` is happy to work with both.

Every tree that `arch` works with has a "patch log" – a record of all of the patches (in the global namespace) that have ever been applied to the tree, along with a record of the full-source revision that the tree started from. In `arch`, a tree never belongs to a specific branch in some specific repository – instead: a source tree is considered to be a part of every branch in every repository for which it has a patch log. At any time, any tree can join any branch with which it has a common ancestor.

`arch` is agile and flexible at handling patch sets. It provides `update` – a patch set manipulation operation that is logically equivalent to the traditional `update` operation of `CVS`; it provides `replay` – a history-sensative merge operation equivalent to the `update` operation of `Subversion` (as documented in the `Subversion` manual); `arch` also provides two rather more sophisticated merge operations – `reconcile` and `i-merge` – which handle very complex (yet quite realistic) branching and merging scenarios gracefully.

In general, the design philosophy of `arch` is to be a very good librarian for whole-tree patch sets, and a very good mechanic for manipulating whole-tree patch sets. `arch` is designed to stay out of your way while you hack, but come to your aid when you `commit`, review, `update`, or merge. The aim is simplicity, clarity of function, and flexibility.

1.3 Introducing arch Project Trees

`arch` manages "project trees". A project tree is a file system tree, usually containing the source code for a project.

What distinguishes a project tree from an ordinary tree is the presence of "arch control files", primarilly stored in a top-level subdirectory called `{arch}`.

The control files include information needed to keep an inventory of the tree, a "patch log" documenting the history of the tree, various default values for `arch` commands applied to the tree, and a local cache of information to speed up some `arch` commands.

When you distribute a tree, usually you will *include* all of the arch control files – they are useful to others.

`arch` has no special concept of a "working copy" (in other revision control systems, a working copy is a tree checked out from the revision control database, as contrasted with a tree from any other source). Any tree that contains `arch` control files can be used as a "working copy". If you download a distribution for a project managed with `arch` and unpack that distribution – you have a working copy.

For more information, see Chapter 4 [arch Project Trees], page 19.

1.4 Introducing arch Inventories

`arch` keeps track of an inventory of the files in a project tree. For example, it can distinguish the files that are officially part of the tree from other files, such as scratch files or editor back-up files. The command:

```
% larch inventory --source
```

prints an inventory of files in a tree.

Every file has two names: its "location" (a path relative to the root of the tree) and its "tag" (a logical, location-independent name for the file). The command:

```
% larch inventory --source --tags
```

prints an inventory of files in a tree, showing the logical name of each file.

`arch` uses tags to keep track, for example, of when files are renamed.

If you are being extra careful, you can use the command:

```
% larch set-manifest
```

to record a list of all the files that are *supposed* to be in a tree and the command:

```
% larch check-manifest
```

to see if any unexpected files have been added, or expected files removed.

For more information, see Chapter 5 [arch Project Inventories], page 21.

1.5 Introducing arch Patch Sets

Experienced programmers should be familiar with the standard command `diff -c -r`, used to create a (standard) "patch set" describing the changes made between two copies of a tree. And they are familiar with the standard command `patch` – used to modify a tree according to the description of changes in a patch set.

Standard patch sets have limitations: they do not cleanly handle file or directory additions, removals, or renames, symbolic links, file permissions, or binary files.

`arch` provides `mkpatch` and `dopatch`, similar to `diff -r` and `patch`, but without the limitations.

For more information, see Chapter 6 [arch Patch Sets], page 35.

1.6 Global Namespaces

`arch` implements a global namespace of projects, taking into account the organization publishing a project, branching of projects, and versioning of branches.

`arch` implements a global namespace of patch sets, building on the global namespace of projects.

Those namespaces give rise to the idea of a "development path". For example, the very first revision of the project `arch`, might be called:

```
lord@regexps.com--arch/arch--0.1--base-0
```

The next three revisions might be called:

```
lord@regexps.com--arch/arch--0.1--patch-1
lord@regexps.com--arch/arch--0.1--patch-2
lord@regexps.com--arch/arch--0.1--patch-3
```

Each of those revision names is the name of a patch set that describes what changed in that revision, compared to the previous revision.

If `arch` forked into a separate development paths, say "intl" (for internationalizing the code), there might be revisions such as:

```
lord@regexps.com--arch/arch--intl--0.1--patch-1
lord@regexps.com--arch/arch--intl--0.1--patch-2
```

`arch` implements a global namespace of all user's of `arch` (layered on top of email addresses). Every patch set has an associated log entry, with a `Creator:` line that contains a user's `arch` id.

For more information, see Chapter 7 [The arch Global Name-space of Users], page 41, Chapter 8 [The arch Global Name-space of Projects], page 43, and Chapter 11 [Basic Revision Control], page 53.

1.7 Introducing arch Archives

`arch` can manage repositories of revisions, storing those revisions as compressed tar files of patch sets or compressed tar files of complete trees.

`arch` archives can be used remotely if they are made accessible by an ordinary FTP server (there is no need for a special `arch`-specific server).

You can (safely, atomically) add a new revision to a repository (`larch import` or `larch commit`). You can reconstruct an arbitrary revision from the files in a repository (`larch get`). You can "branch" a development path to create a new, related development path (`larch tag`). Branches can cross repository boundaries, and to the user, `arch` appears to integrate the two repositories into one.

Repositories can be migrated and, for read-only access, replicated.

`arch` repositories have an easy-to-understand format, amendable to browsing by hand or with special-purpose interface programs.

For more information, see Chapter 9 [Archives], page 47.

1.8 Introducing arch Patch Logs

In every project tree, `arch` keeps a detailed "patch log": a record of what the original source tree was, along with a record for every patch applied to the tree (regardless of what branch the patch came from).

`arch` log entries use RFC822-style formatting. Automatically generated headers record what files were changed by each patch, what repository the patch came from, what user created the patch, etc. The patch log is a very useful source of information for programmers.

`arch` can automatically generate and maintain a GNU-style ChangeLog from its patch log.

When you "merge" two branches (combine the changes made in those branches), `arch` uses the information in the patch log to avoid redundantly applying patches.

For more information, see Chapter 13 [Patch Logs and ChangeLogs], page 67.

1.9 Cheap Branching and Smart Merging

Creating a branch in `arch` is inexpensive in both space and time.

The `arch` commands for merging have many subtle features to help a merge go smoothly.

For more information, see Chapter 12 [Basic Branching and Merging], page 63, Chapter 17 [Multi-Branch Merging – The reconcile Command], page 81, and ⟨undefined⟩ [Idempotent Merging], page ⟨undefined⟩.

1.10 What Does It All Mean?

Putting all those features together, `arch` is an elegant and more featureful replacement for older systems like `CVS`.

For example, using `arch`, every programmer can conveniently have a private repository for day-to-day work rather than burdening a shared repository with per-user branches.

A project can be "multi-homed" – stored in multiple repositories – with different branches in each repository.

It is easy and convenient, when using `arch`, to improve and clean-up projects by re-organizing the files and directories they contain: something that is quite awkward with `CVS`.

The fancy branching and merging commands of `arch` make it convenient to do more development than ever in feature-specific branches, merging features as they are completed, generating a clean, complete, and isolated patch set for each new feature.

`arch` is written in a software-tools style: it is made up of many small programs, each of which does one job well. The commands have very regular option and argument syntax and input and output formats. `arch` is an excellent foundation for process automation and for layering under fancy graphical interfaces. `arch` is self-documenting and extensible.

2 System Requirements

In order to use `arch`, there are some software tools that you must already have available. These don't necessarily need to be on your `PATH` – `arch` can use a separate `PATH` if you need it to.

GNU Make You will need `GNU Make` in order to build `arch`.

GNU Tar You must have `GNU tar`. More specificially, you must have a version of tar that has options:

```
-zcf F          to create a gzip-compressed tar file called
                F, where F may be '-', meaning to write
                the tar file to the standard-output stream

-zxf F          to extract files from a gzip-compressed tar
                file called F, where F may be '-', meaning
                to read the tar file from the standard-input
                stream.

-T -            This option reads a list of files from
                standard input.  Only those files are
                read or written to the archive -- others
                are ignored.

-m              When extracting files, don't restore
                modification times.
```

GNU diff and GNU patch After much deliberation, I've decided to go ahead and rely on the **GNU** versions of `diff` and `patch`. Specifically, you need a version of `diff` that can generate "unified format" output (option `-u`) and a version of `patch` that understands that format and that understand `--posix`. (It would be trivial to use "context diffs" and, thus, standard `diff` and `patch`, however, unified diffs are much easier to read, and I'm hoping that picking specific implementations of these critical sub-components will help contribute to the long-term stability of `arch`.)

Standard Posix Shell Tools The package framework assumes that some standard Posix shell tools are available on your system. At the moment, `sh` must be installed as `/bin/sh`, but this will be corrected in a future release:

```
awk
cat
chmod
date
echo
find
fold
grep
head
ls
```

```
mkdir
printf
pwd
rm
sed
sh
tee
test
touch
tsort
wc
xargs
```

ordinary -exec extension to find Your version of `find` must be able to expand `{}` even in the context of a larger string. For example:

```
find . -exec echo ">>>{}<<<" ";"
```

should print a list of all files and directories, surrounded by '`>>>`' and '`<<<`'. `GNU find` has this property as do, I believe, most implementations. (Posix explicitly requires only that `{}` be expanded in isolation, leaving undefined the meaning of `{}` when embedded in a longer string.)

The null Device Your system must have `/dev/null`. Output directed to `/dev/null` should simply disappear from the universe, in the usual way.

The BSD column program – maybe If the program `column` is on your `PATH` (or the path being used by `arch`), it should be a program which formats its input into columns and, with the option `-x`, fills columns before rows. If this program isn't found, it won't be used.

3 arch Commands in General

Every command in `arch` is accessed via the program `arch`, using an ordinary sub-command syntax:

```
% larch <sub-command> <options> <parameters>
```

A list of all sub-commands can be obtained from:

```
% larch --help-commands
```

The most complete documentation for each command is available via its help message. The help messages in `arch` are roughly comperable to traditional unix `man` pages. For example, try:

```
% larch make-archive --help
create a new archive directory
usage: make-archive [options] (-u | name) directory

 -V --version                  print version info
 -h --help                     display help

 -r --readme file              save FILE as the README
                               for this archive

 -u --update                   update an existing archive


NAME is the global name for the archive.  It must be an
email address with a fully qualified domain name, optionally
followed by "--" and a string of letters, digits, periods
and dashes.

Normally the archive directory must not already exist, but.
there is an exception:

If -u or --update is specified, then attempt to bring
an existing archive up to date with respect to this
version of arch and update meta-info, such as the README.
file.

If -u or --update is specified, the archive name must
not be specified.
```

There is a great deal of regularity among commands regarding option names and parameter syntax. Hopefully, you'll pick this up as you learn the various commands.

3.1 The arch Commands

Here is a synopsis of all `arch` commands: the output of `larch --help-commands`:

```
                        larch sub-commands
                        -----------------

  * User Commands

                        my-id : print or change your id
           my-default-archive : print or change your default archive

             register-archive : record the location of an archive
              whereis-archive : print the registered location of an archive
                     archives : report registered archives and their locations


  * Project Tree Commands

                    init-tree : initialize a new project tree
                    tree-root : find and print the root of a project tree

                 tree-version : print the default version for a project tree
             set-tree-version : set the default version for a source tree


  * Project Tree Inventory Commands

                    inventory : inventory a source tree

               tagging-method : print or change a project tree tagging method
                    tree-lint : audit a source tree
                 missing-tags : list files missing inventory tags
                     file-tag : print the tag of a project tree file

                          add : add an explicit inventory tag
                       delete : remove an explicit inventory tag
                         move : move an explicit inventory tag

             explicit-default : set the default explicit tag for a directory

                 set-manifest : set the inventory manifest from project tree
                     manifest : print the inventory manifest
               check-manifest : compare the inventory manifest to reality
```

```
* Patch Set Commands

                    mkpatch : compare two source trees and create a patch tree
                    dopatch : apply a patch tree to a source tree

               patch-report : generate a report from a patch set

                empty-patch : make a null patch set


* Archive Commands

               make-archive : create a new archive directory

              make-category : create a new category within an archive
                make-branch : create a new branch within an archive
               make-version : create a new version within an archive

                 categories : list the categories in an archive
                   branches : list the branches in an archive category
                   versions : list the versions in an archive branch
                  revisions : list the revisions in an archive version

            cat-archive-log : print the contents of an archive log entry

     archive-cache-revision : cache a full source tree in an archive
   archive-cached-revisions : list full source trees cached in an archive
   archive-uncache-revision : remove a cached revision from an archive

            category-readme : print the =README of a category
              branch-readme : print the =README of a branch
             version-readme : print the =README of a version


* Patch Log Commands

                   make-log : initialize a new log file entry

                       logs : list patch logs in a project tree
                    add-log : add a version patch log to a project tree

                     log-ls : print version patches in a project tree
                    cat-log : print the contents of a project tree log entry
           log-header-field : filter a header field from a log entry

                  changelog : generate a change log from a patch log
              log-for-merge : generate a log entry bodyfor a merge
```

```
                merge-points : report where two branches have been merged
               new-on-branch : list tree patches new to a branch


* Archive Transaction Commands

                      import : archive a from-scratch base revision
                      commit : archive a revision
                         get : construct a project tree for a revision

                   get-patch : retrieve a patch set from an archive

                 lock-branch : lock a branch
               lock-revision : acquire the lock for the next revision

                 push-mirror : update a read-only repository mirror


* Multi-project Configuration Commands


                build-config : instantiate a multi-project tree
               update-config : update a multi-project tree
               replay-config : replay a multi-project tree

               record-config : record a revision-specific configuration
                 show-config : show the revision frontier of a configuration

              config-history : report the history of a configuration


* Commands for Branching and Merging


                      update : merge local changes with the latest revision
                      replay : merge the latest revsion with local changes
                 delta-patch : compute and apply an arbitrary patch
                  star-merge : merge of mutually merged branches

                         tag : create a continuation revision (tag or branch)

              prepare-branch : create a project tree for a new branch
               finish-branch : commit project tree as a new branch

                        join : join a sibling branch

               whats-missing : print patches missing from a project tree
```

```
* Local Cache Commands

             what-changed : compare project tree to cached pristine
               file-diffs : compare file with cached pristine revision

                pristines : list the pristine trees in a project tree
            lock-pristine : lock a pristine revision locking


* Revision Library Commands

      my-revision-library : print or set your revision library path

             library-find : find a revision in a revision library
              library-add : add a revision to a revision library
           library-remove : remove a revision from a revision library

         library-archives : list the archives in the revision library
       library-categories : list categories in the revision library
         library-branches : list branches in the revision library
         library-versions : list versions in the revision library
        library-revisions : list revisions in the revision library

              library-log : print a log message from the library
             library-file : find a file in a revision library


* Web Related Commands

            patch-set-web : create or update a patch-set web
     update-distributions : build or update an FTP area

        distribution-name : revision name -> distribution name


* Notification Commands

                   notify : trigger actions for changes to an archive
              my-notifier : print or set your default notify directory

      mail-new-categories : send email notices about new categories
        mail-new-branches : send email notices about new branches
        mail-new-versions : send email notices about new versions
       mail-new-revisions : send email notices about new revisions
```

```
          notify-library : add new revisions to the library

      push-new-revisions : send email notices about new revisions

          sendmail-mailx : send email with sendmail ala POSIX mailx


For help with a command, try: larch command --help
```

4 arch Project Trees

One of the central organizing concepts of `arch` is the tree *project tree*. For the most part, a project tree is an ordinary tree of files, directories and symbolic links – all files that you create and maintain as part of your project.

In addition to the tree content you create, `arch` itself maintains some control files in a project tree. You should regard these extra control files as part of your project's content. For example, if you distribute source for a program managed by `arch`, you will ordinarilly want to *include* the control files: they are useful to other people.

4.1 Initializing a Project Tree

To initialize a project tree for the first time, from the root of the tree, use the command:

```
% larch init-tree
```

For example, here is how I initialized the project tree containing the source for `arch` itself:

```
% cd ~/wd/arch
% larch init-tree
```

That will create a subdirectory, at the root of the tree, called `{arch}`. Most `arch` control files are stored in the tree rooted at `{arch}`: you should never create, remove, or modify files there by hand.

When you have multiple project trees for related projects it is good practice to make them sibling directories. This is because `arch` sometimes caches information in project trees and those caches can speed up some operations. When looking for information in a cache, `arch` looks not only in the current project tree, but in any sibling project trees. For example, while working on `arch`, I might have several copies of `arch`, each for working on a different set of features:

```
% cd ~/wd
% ls
arch    arch-branches   arch-inventory   arch-reporting
```

In this manual, `~/wd` always refers to a "directory of project trees". (The convention means the same thing as, but is less cumbersome than `${PROJECT_TREES}` – there is no requirement that your directory of project trees be called `~/wd`, or that you only have one such directory.)

4.2 Finding the Root of a Project Tree

From any directory within a project tree, the command:

```
% larch tree-root
```

prints the full path name of the root of the project tree:

```
% cd ~/wd/arch/src
% larch tree-root
/home/lord/wd/arch
```

5 arch Project Inventories

In a project tree, some of the files and directories are "part of the source" – they are of interest to `arch`. Other files and directories may be scratch files, editor back-up files, and temporary or intermediate files generated by programs. Those other files should be ignored by most `arch` commands.

This chapter discusses how `arch` recognizes which files to pay attention to, and which to ignore.

`arch` has flexible facilities for keeping track of all of the files and directories in your project: for taking "inventories" of your project tree. It has these facilities for three reasons:

Distinguishing Source `arch` uses a project inventory to distingiuish files and directories which are part of your project from other files and directories which are temporary files, scratch files, editor backup-files, and so forth.

Additionally, `arch` permits you to overlay projects: store more than one project at a single root. When you do that, `arch` uses inventories to sort out which files and directories belong to each project. (The topic of overlays, however, is deferred until a later chapter.)

Recognizing Renames Every file or directory in an `arch` inventory has two names. One name is simply the location (path) of the file relative to the root of the project tree. The other name is a "logical name" for the file: a name that remains the same regardless of where in the project tree the file is located. When `arch` compares two versions of a project tree, it uses logical names to discover when files or directories have been moved, renamed, deleted, or added.

Canonical Inventories Finally, `arch` permits you to make a record of the "canonical inventory" of your project – all of the files that you believe are **supposed** to be there. `arch` can then tell you whether any files are missing or have been added compared to the canonical inventory.

5.1 Choices Regarding Inventories

For each project tree, you have a choice to make regarding how project inventories work. The options are described briefly here, then in more detail in the sections that follow.

Naming Conventions The simplest (and default) option is to simply use *naming conventions*. `arch` will search your tree for files matching certain naming patterns, and consider all of those files to be source files.

When you use only naming conventions to take an inventory, the logical name of a file and its location name are exactly the same. For that reason, if you rename a file, `arch` will think you deleted a file with the old name, and added a file with the new name. If you delete a file, then add a file with the same name, `arch` will think that the new file is a modified form of the old file. None of those limitations are fatal, `arch` will still work, but they do limit the effectiveness of `arch` at branching and merging. ("Branching" and "merging" are topics of a later chapter.)

Explicit Inventories Another option is to use an *explicit inventory*. Once again, `arch` will search for files that satisfy certain naming conventions – but not every such file or directory is automatically source. Instead, whenever you add, delete or renamed a file, you must inform `arch` of that fact explicitly. For example, after adding the file `foo.c`, you have to tell `arch`:

```
% larch add foo.c
```

and if you rename `foo.c` to `bar.c`, then you must also tell `arch`:

```
% larch move foo.c bar.c
```

Implicit Inventories A third option combines some of the advantages of using naming conventions with some of the advantages of explicit inventories: *implicit inventories*. When you use an implicit inventory, every file that passes the naming conventions is considered source. You **may** explicitly add, delete, and rename files – allowing `arch` to precisely track renames for those files and directories. You also **may** store a *file tag* (the "logical name" of a file) *in* any file. If you don't explicitly tag a file, and use an implicit inventory, `arch` will search for those embedded tags and use them to precisely detect new files, deleted files, and renamed files.

Each of the three options is called a *tagging method*.

There is some advice at the end of this chapter about how to choose among the three tagging methods.

5.2 Specifying a Tagging Method

If you never explicitly specify a tagging method, `arch` will use simple naming conventions, by default. You can also make explicit your choice to use only naming conventions with this command issued in a project tree:

```
% larch tagging-method names
```

Similarly, to use either an implicit or explicit inventory, use one of the commands:

```
% larch tagging-method explicit
% larch tagging-method implicit
```

To find out what method a given project tree uses, use the same command with no argument:

```
% larch tagging-method
names
```

5.3 The inventory Command

The command `larch inventory` is used to print a list of source files. It has many options, including options to print other kinds of file lists (such as a list of all editor backup files, or a list of all files which are not source):

```
% cd source-tree

% larch inventory --source
hello.c
hello.h
library
library/buffer.c
library/buffer.h
...
```

contrasted with:

```
% cd source-tree

% ls
hello.c   hello.c.~1~   hello.h  library
```

(Notice that `hello.c.~1~` is not included in the inventory of source files.)

The naming conventions used by `arch` are as follows:

Control Files A *control file is* part of the source, but control files are not included in the output of `larch inventory` unless the `--all` flag is used. Control file and directory names match any of these patterns:

```
.arch-project-tree
.arch-ids
.owned.*
.common
{arch}
```

Junk Files A *junk file* is *not* part of the source. A junk file or directory name matches the pattern:

```
,*
```

or if it contains any of the characters:

```
<space>
<tab>
<newline>
[
]

?
\
```

Note that if a directory name matches that pattern, then none of the contents of the directory are part of the source, regardless of their names.

Junk files are listed by the command:

```
% larch inventory --junk
```

Arch sometimes creates junk files and directories of its own. When it does, those files and directories have names that match the pattern:

```
,,*
```

You should avoid creating files and directories with names that match that pattern. `arch` will freely delete files and directories with names that match `,,*` whenever it needs to re-use such a name.

Usually, `arch` will delete any junk file it creates before the command that created the junk file terminates. Sometimes, though, when a command fails, `arch` will leave behind junk files or directories matching `,,*`. This is a debugging feature, likely to be removed in a future release. For now, whenever you find such a file (and are confident it isn't being used by a currently running command), you are free to delete it.

Backup Files If a file is not a junk file, it may be a file *backup file*. Backup files are not part of the source. They match any of the patterns:

```
*~
*.bak
*.modified
*.orig
*.original
*.rej
*.rejects
```

Backup files are listed by the command:

```
% larch inventory --backups
```

Precious Files If a file is not a control file, junk file, or backup file, it might be a *precious file*. Precious files are not part of the source, but `arch` does sometimes treat them specially. For example, when `arch` copies a directory of source for you, it copies not only the source files, but the precious files as well.

Precious files and directories match one of these patterns:

```
+*
.gdbinit
=build*
=install*
CVS
RCS
TAGS
```

Of course, precious files can be listed by the command:

```
% larch inventory --precious
```

Sometimes `arch` will create its own precious files – usually to save some information that you might not want to lose. When it does, it creates a file or directory matching the pattern:

```
++*
```

You should avoid creating such filenames yourself. `arch` won't every delete such a file – but if one happens to get in the way of an `arch` command, that command will fail with an error.

Source Files If a file is not a control, junk, backup, or precious file, it might be an ordinary *source file*. Source files are, of course, the files that `arch` stores in an archive (along with control files).

Source files must match the pattern:

```
[=a-zA-Z0-9]*
```

but must *not* match any of the patterns:

```
*.o
*.core
core
```

Ordinary source files are listed by:

```
% larch inventory --source
```

Some files which are `arch` control files are counted as source even though they don't match the patterns above. However, these files are not listed by default. All source files (ordinary source plus control files) are listed by:

```
% larch inventory --source --all
```

Unrecognized Files Any file that doesn't fall into the above categories is an *unrecognized file*. Unrecognized files can be listed by the command:

```
% larch inventory --unrecognized
```

WARNING The basic pattern for source files is:

```
[=a-zA-Z]*
```

however, you should restrict yourself to file names that do not contain spaces. Filenames containing spaces are likely to trigger bugs in the current release of `arch`.

5.4 Using an Explicit Inventory

If you want to use explicit designation of source filess, rather than naming conventions alone, then use this command:

```
% larch tagging-method explicit
```

Note that you must use that command from within a working directory tree that has already been initialized using `init`.

When using explicit designation, it is (ordinarilly) necessary to add every file and directory in the source to the explicit list using the command:

```
% larch add FILE
```

If `FILE` is a directory, that will create `FILE/.arch_ids/=id`. If it is a regular file or symbolic link, it will create (in the same directory) `.arch_ids/FILE.id`. In either case, the file created will contain an obscure string known as an "inventory tag" (inventory tags are explained in more detail below).

If you remove a regular file or symbolic link, you must use the command:

```
% larch delete FILE
```

That won't remove `FILE` itself, but it will remove the inventory tag for `FILE`.

In order to remove a directory, you must yourself remove the `.arch_ids` subdirectory. That will also implicitly remove the inventory tags of any files that `arch` thinks are stored in that directory.

If you rename a regular file or symbolic link, you can use the command:

```
% larch move OLD-NAME NEW-NAME
```

to move the inventory tag for that file.

If you rename a directory, it's inventory tag (and the tags for all files and subdirectories it contains) move with it automatically (because the `.arch_ids` subdirectory has moved).

When you run `larch inventory` in a working directory using explicit designation, only explicitly designated source files are listed. If you would rather see a list of all files passing the naming conventions for source files, use:

```
% larch inventory --source --names
```

You should also read about `tree-lint` later in this chapter.

5.5 Using an Implicit Inventory

To use implicit tagging, use the following command in your working directory:

```
% larch tagging-method implicit
```

When implicit tagging is used, every file that passes the naming conventions is treated as source. If a file or directory has an explicit tag (created with add), arch will use that explicit tag to recognize when a file has moved. If a file (but not a directory or symbolic link) lacks an explicit tag, arch will look for a tag in the file itself.

A tag within a file has one of two forms. It may be either:

```
<punct><basename><spaces>-<spaces><tag>
```

where <punct> is an arbitrary string of punctuation and spaces, <basename> is the basename of the file, and <tag> an inventory tag for the file. Or:

```
<punct>tag:<spaces><tag>
```

In either case, <tag> should be unique among the files within a directory. A tag within a file must occur within the first 1024 bytes of the file.

A handy convention for source files is to add a comment to the top of every file, briefly stating the purpose of the file:

```
          /* hello.c - 'main' for the hello world program
    ...
```

or:

```
          /* tag: 'main' for the hello world program
    ...
```

Another possible convention is to use a string identifying the author and the time the file was first created (or first tagged):

```
          /* tag: joe.hacker@gnu.org Thu Nov 29 17:25:15 PST 2001
    ...
```

If you use the basename form of an implicit tag, and actually rename a file (rather than simply move it between directories), you do need to remember to update the tag line to reflect the new basename.

When you use implicit tagging, it is ok if a file lacks any tag at all, either explicit or implicit. In that case, if you rename the file, arch will think you've deleted the old file and added a new one – but aside from that, everything will work normally.

CAUTION: Leading and trailing spaces around an inventory tag are not considered part of the tag. Within a tag, every non-graphical character is replaced by _. For example, you write the that tag:

```
'main' for the hello    world program
```

the actual inventory tag is:

```
‘main’_for_the_hello____world_program
```

It is possible that a future release of `arch` will slightly change the rule – so that multiple spaces and tabs are replaced by a single `_`.

5.6 Recognizing Renames – Inventory Tags

If you are using naming conventions only to recognize source files, then if you rename a directory or file, `arch` will conclude that you have deleted the old file, and created a new file.

If you are using an explicit source inventory, `arch` will always recognize when a directory is renamed (presuming that the `.arch_ids` subdirectory is preserved), and it will recognize when a file is renamed if you use `move` (rather than `delete` and `add`). Of course, `arch` can be fooled if you swap two files without swapping their inventory tags.

If you are using an implicit inventory, `arch` will never recognize when an untagged file is renamed (it will think "delete" and "add"). If a file is tagged explicitly, `arch` will recognize when the file is added, deleted, or renamed – just as when using an explicit inventory. If a file is not tagged explicitly, but has an embedded tag, `arch` will recognize when the file is added, deleted or moved.

5.7 Keeping Things Neat and Tidy

The command:

```
% larch tree-lint
```

is useful for keeping things neat and tidy.

If you use explicit tagging, it will tell you of any tags for which the corresponding file does not exist. It will tell you of any files that pass the naming conventions, but for which no explicit tag exists.

If you use implicit tagging, it will tell you of any files for which no tag can be found – either explicit or implicit. It will tell you of any explicit tags for which the corresponding file does not exist.

In either case, or if you are using naming conventions only, `tree-lint` will tell you of any files that don't fit the naming conventions at all.

Finally, if you use explicit or implicit tagging, `tree-lint` will check for cases where multiple files use the same tag. If any two files do have the same tag, you **must** correct that, either by editting the tag (if it is in the file itself) or by using `delete` and `add` to replace a duplicated explicit tag.

5.8 Avoiding Accidental Ommissions and Additions

A *manifest* is an explicit list of the files you believe are **supposed** to be in a project tree. `arch` allows you to maintain a manifest, and to compare it to the actual contents of a tree.

The command `set-manifest` sets the manifest to the current contents of the project tree:

```
% larch set-manifest
```

Note that only regular source files, not `arch` control files, are included in the manifest. To replace an existing manifest, you must provide the `-f` flag (or `--force`) to `set-manifest`.

You can retrieve the manifest with:

```
% larch manifest
```

Each line of the manifest is of the form:

```
<path>\t<tag>
```

and the list is sorted by the `<tag>` field.

You can look for missing, added, or renamed files with:

```
% larch check-manifest
```

which will compare the project tree inventory to the manifest and print a report describing divergences (if there are any).

5.9 The Inventory Tag Abstraction in Detail

When `arch` considers the files and directories in a working directory it builds a one-to-one index mapping path names (relative to the root of the working directory tree) to *inventory tags*.

The inventory tag of a file is its "logical identity". The path is the position of that identity within the particular working dir.

You can see the inventory tag for each source file with the command:

```
% larch inventory --source --tags
```

When `arch` compares two project trees, it bases the comparison on logical identities. If both trees have a file with a particular inventory tag, but the files are in different positions, then `arch` considers the file to have been moved or renamed. Similarly, if an inventory tag is present in one tree, but missing in the other, then `arch` considers the file to have been added or deleted.

If you use naming conventions only, the inventory tag of each file is the same as its path. Thus, when using the `names` tagging method, `arch` never recognizes that a file has been moved or renamed.

When you use the `explicit` tagging method, inventory tags are stored in the `.arch-ids` directories. There is a file in `.arch-ids` for each tagged file (and one file for the directory containing `.arch-ids`), and those files contain the tags.

When you use the `implicit` tagging method, tags in `.arch-ids` directories take precedence (if they exist). If a file is not explicitly tagged, `arch` searches for the inventory tag in the file itself (as described earlier in the chapter). Finally, if a file is not tagged at all, then its path is used as the inventory tag.

5.10 A Warning About Changing Tagging Methods

Be cautious when changing tagging methods for directories already checked-in to an `arch` revision control archive.

For example, if you change from the tagging method `names` to `explicit`, then the inventory tag for every file will change. `arch` will think that you've deleted all of the files in the old tree, and added all of the files in the new tree.

However, there is a work-around for this problem, described in a later chapter.

5.11 Other Ways to Tag Files

In some situations, it isn't convenient to explicitly tag every file or to add an implicit tag to every file.

You can supply a *default tag* for every file that doesn't have an explicit tag with the command:

```
% larch explicit-default TAG-PREFIX
```

After that, every file in that directory which lacks an explicit tag will have the tag:

```
TAG-PREFIX__BASENAME
```

where `BASENAME` is the basename of the file. Default tags created in this way take precedence over implicit tags embedded in files. You can find out the default tag for a directory with:

```
% larch explicit-default
TAG-PREFIX
```

and remove the default with:

```
% larch explicit-default --delete
```

You can also specify a default tag which has *lower* precedence than implicit tags:

```
% larch explicit-default --weak TAG-PREFIX
```

and view that default:

```
% larch explicit-default --weak
```

or delete it:

```
% larch explicit-default --weak --delete
```

5.12 Telling tree-lint to Shut Up

When using implicit tags, you may sometimes have a directory with many files that have no tag (either explicit or implicit), but not want those files to appear in a report of untagged files generated by `tree-lint`. There are two ways to tell `tree-lint` to shut-up about such files:

One is to provide a default explicit tag or weak default explicit tag using `larch explicit-default`, as described above.

The second method is to label the directory as "don't care" directory – which means that `tree-lint` shouldn't complain about untagged files. You can do that with:

```
% larch explicit-default --dont-care set
```

or remove the "don't care" flag with:

```
% larch explicit-default --delete --dont-care
```

You can find out whether the "don't care" flag is set in a given directory with:

```
% larch explicit-default --dont-care
```

5.13 Which Tagging Method Should You Use?

Given the choice of the `names`, `explicit`, and `implicit` tagging conventions, which one should you choose?

The `names` method is best for project trees that you don't control, and for which the maintainer does not include file tags (either explicit or implicit). For such trees, the `names` method will always work, but if you want to use the `explicit` or `implicit` method, you'll have to add file tags yourself.

The `implicit` method is, in my opinion, by far the most convenient. It is easy to get in the habit of adding a `tag:` line to the bottom of each new file and doing a single `larch add` for each directory. After those steps, you can rename files and directories freely – without having to remember to tell `arch` in a separate command.

On the other hand, the `implicit` method has two limitations. One limitation is that you must accept the possibility of accidently adding new files to the inventory. Any file you create that passes the naming conventions counts as source. The other, closely related, limitation is that if you use `implicit` inventories, you will **never** want to compile a program in its own source directory. When you compile a program, that creates intermediate files and executables. Many of those files will almost certainly pass the naming conventions for source – so `arch` will wrongly include them in a source inventory. I use the `implicit` method, but my `configure` scripts have a safeguard that causes them to refuse to compile my programs in the source tree.

Finally, the `explicit` method is the only choice left if you want the benefits of real file tags (therefore you can't use the `names` method) but either insist on compiling in the source tree or can't risk accidently adding the occasional unintended file (so you shouldn't use `implicit`).

5.14 Altering the Naming Conventions

Note: this is a relatively new feature, so the documentation is not yet well integrated with the rest of the manual.

The file `{arch}/=tagging-method` defines the naming conventiosn used for a particular project tree. By editting that file, you can estalish naming conventions that are different from the defaults, which are described above.

That file can contain blank lines and comments (lines beginning with #) and directives, one per line. The permissable directives are:

```
implicit
explicit
names
        specify the tagging method to use for this tree

exclude RE
junk RE
backup RE
precious RE
unrecognized RE
source RE
        specify a regular expression to use for the indicated
        category of files.
```

Regular expressions are specified in Posix ERE syntax (the same syntax used by *egrep*, *grep -E*, and *awk*) and have default values which implement the naming conventions described above.

The `exclude` pattern should match a subset of files matched by the `source` pattern. Files which match `exclude` are printed by:

```
% larch inventory --source --all
```

but not printed by:

```
% larch inventory --source
```

Although you can define your own naming conventions, there are some minor limitations:

The file names `.` and `..` are always ignored by `inventory`.

File names which contain non-printing characters, spaces, or any of the globbing characters (`*`, `[`, `]`, `\`, `?`) are always placed in the category `unrecognized`. This is so that tools which operate on project trees can safely presume that no source file has a name that includes these characters.

File names which begin with ,, are always placed in the category `junk`. This is so that tools which operate on a project tree can safely destroy or create files beginning with ,,.

The default naming conventions are given by:

```
exclude ^(.arch-ids|\{arch\})$
junk ^(,.*)$
backup ^.*(~|\.~[0-9]+~|\.bak|\.orig|\.rej|\.original|\.modified|\.reject)$
precious ^(\+.*|\.gdbinit|=build\.*|=install\.*|CVS|CVS\.adm|RCS|RCSLOG|SCCS|TAGS)$
unrecognized ^(.*\.(o|a|so|core)|core)$
source ^([_=a-zA-Z0-9].*|\.arch-ids|\{arch\}|\.arch-project-tree)$
```

6 arch Patch Sets

It is often extremely useful to compare two project trees (usually for the same project) and figure out exactly what has changed between them. A record of such changes is called a *patch set* or a *delta*.

If you have a patch set between an "old tree" and a "new tree", you can "apply the patch" to the old tree to get the new tree – in other words, you can automatically make the editting changes described by a patch set. If you have some third tree, you can apply the patch to get an approximation of making the same changes to that third tree. (see Appendix C [The Theory of Patches and Revisions], page 125).

`arch` includes sophisticated tools for creating and applying patch sets.

6.1 mkpatch

`mkpatch` computes a patch set describing the differences between two trees. The basic command syntax is:

```
% mkpatch ORIGINAL MODIFIED DESTINATION
```

which compares the trees `ORIGINAL` and `MODIFIED`.

`mkpatch` creates a new directory, `DESTINATION`, and stores the patch set there.

When `mkpatch` compares trees, it uses inventory tags. For example, it considers two directories or two files to be "the same directory (or file)" if they have the same tag – regardless of where each is located in its respective tree.

A patch set produced by `mkpatch` describes what files and directories have been added or removed, which have been renamed, which files have been changed (and how they have been changed), and what file permissions have changed (and how). When regular text files are compared, `mkpatch` produces a context diff describing the differences. `mkpatch` can compare binary files (saving complete copies of the old and new versions if they differ) and symbolic links (saving the old and new link targets, if they differ).

A detailed description of the format of a patch set is provided in an appendix (see Appendix D [The arch Patch Set Format], page 133).

6.2 dopatch

`dopatch` is used to apply a patch set to tree:

```
% dopatch patch-set tree
```

If `tree` is exactly the same as the the "original" tree seen by `mkpatch`, then the effect is to modify `tree` so that it is exactly the same as the the "modified" tree seen by `mkpatch`, with one exception (explained below).

"Exactly the same" means that the directory structure is the same, symbolic link targets are the same, the contents of regular files are the same, and file permissions are the same. Modification times, files with multiple (hard) links, and file ownership are not reliably preserved.

The exception to the "exactly the same" rule is that if the patch requires that files or directories be removed from `tree`, those files and directories will be saved in a subdirectory of `tree` with an eye-splitting name matching the pattern:

```
++removed-by-dopatch-PATCH--DATE
```

where `PATCH` is the name of the patch-set directory and `DATE` a timestamp.

At this time, `dopatch` does not support reverse patching, though this is likely to be added in a future release.

6.3 Inexact Patching

What if a tree patched by `dopatch` is not exactly the same as the original tree seen by `mkpatch`?

Below is a brief description of what to expect. Complete documentation of the `dopatch` process is included with the source code.

`dopatch` takes an inventory of the tree being patched. It uses inventory tags to decide which files and directories expected by the patch set are present or missing from the tree, and to figure out where each file and directory is located in the tree.

Simple Patches If the patch set contains an ordinary patch or metadata patch for a link, directory or file, and that file is present in the tree, `dopatch` applies the patch in the ordinary way. If the patch applies cleanly, the modified file, link, or directory is left in place.

If a simple patch fails to apply cleanly, `dopatch` will always leave behind a `.orig` file (the file originally in the tree being patched, without any changes) and a `.rej` file (the part of the patch that could not be applied).

If the patch was a context diff, `dopatch` will also leave behind the file itself – partially patched.

If an (unsucessful) patch was for a binary file, no partially-patched file will be left. Instead, there will be:

```
.orig   -- the file originally in the tree being patched,
           without modifications.
```

```
.rej    -- a complete copy of the file from the modified tree,
           with permissions copied from ‘.orig’.

.patch-orig -- a complete copy of the file from the original
               tree seen by ‘mkpatch’, with permissions
               retained from that original

                -or-


               the symbolic link from the original tree seen
               by ‘mkpatch’ with permissions as in the original
               tree.
```

If an (unsucessful) patch was for a symbolic link, no partially patched file will be left. Instead there will be:

```
.orig   -- the unmodified file from the original tree


.rej    -- a symbolic link with the target intended by the
           patch and permissions copied from .orig

.patch-orig -- a complete copy of the file from the original
               tree seen by ‘mkpatch’, with permissions
               retained from that original

                -or-


               the symbolic link from the original tree seen
               by ‘mkpatch’ with permissions as in the original
               tree.
```

Patches for Missing Files

All patches for missing files and directories are stored in a subdirectory of the root of the tree being patched called

```
==missing-file-patches-PATCH-DATE
```

where `PATCH` is the basename of the patch set directory and `DATE` a time-stamp.

Directory Rearrangements and New Directories

Directories are added, deleted, and rearranged much as you would expect, even if you don’t know it’s what you’d expect.

Suppose that when `mkpatch` was called the `ORIGINAL` tree had:

```
Directory or file:              Tag:


a/x.c                           tag_1
a/bar.c                         tag_2
```

but the `MODIFIED` tree had:

```
a/x.c                           tag_1
a/y.c                           tag_2
```

with changes to both files. The patch will want to rename the file with tag `tag_2` to `y.c`, and change the contents of the files with tags `tag_1` and `tag_2`.

Suppose, for example, that you have a tree with:

```
a/foo.c                         tag_1
a/zip.c                         tag_2
```

and the you apply the patch to that tree. After the patch, you'll be left with:

```
a/foo.c                         tag_1
a/y.c (was zip.c)               tag_2
```

with patches made to the contents of both files.

Here's a sample of some subtleties and ways of handling conflicts:

Suppose that the original tree seen by mkpatch has:

```
Directory or file:              Tag:


./a                             tag_a
./a/b                           tag_b
./a/b/c                         tag_c
```

and that the modified directory has:

```
./a                             tag_a
./a/c                           tag_c
./a/c/b                         tag_b
```

Finally, suppose that the tree has:

```
./x                             tag_a
./x/b                           tag_b
./x/c                           tag_new_directory
./x/c/b                         tag_diffent_file_named_b
./x/c/q                         tag_c
```

When patch gets done with the tree, it will have:

```
./x                             tag_a
        Since the patch doesn't do anything
        to change the directory with tag_a.

./x/c.orig                      tag_new_directory
./x/c.rej                       tag_c
        Since the patch wants to make the
        directory with tag_c a subdirectory named "c"
        of the directory with tag_a, but the tree
        already had a different directory there,
        with the tag tag_new_directory.
```

```
./x/c.rej/b                     tag_b
        Since the patch wants to rename the directory
        with tag_b to be a subdirectory named "b"
        of the directory with tag_c.

./x/c.orig/b                    tag_diffent_file_named_b
        Since the patch made new changes to this file,
        it stayed with its parent directory.
```

7 The arch Global Name-space of Users

For various purposes (such as labelling the author of log messages), `arch` maintains a global name-space of users. Every user of `arch` has an associated user ID, which is (ideally) globally unique.

An id string has two parts: a free-form part, and a unique-id part. The unique-id part is an email address with a fully-qualified domain name. That part of your id string should be unique to you in the world. Here is an example:

```
    Joe Hacker <joe.hacker@gnu.org>
        ^                ^
        |                |
  free form part     unique id
```

The free-form part must match the regexp:

```
[[:alnum:][:space:][:punct:]]*
```

and the unique-id part must match the regexp:

```
<[-.[:alnum:]]+@[-.[:alnum:]]+\\.[-.[:alnum:]]+>
```

You should only need to set your ID once, which you can do with the command:

```
% larch my-id ID-STRING
```

You can check your id with:

```
% larch my-id
Joe Hacker <joe.hacker@gnu.org>
```

or:

```
% larch my-id --uid
joe.hacker@gnu.org
```

Clearly it is a good idea to use a real email address (belonging to you) for the id string, but there is nothing that requires this.

8 The arch Global Name-space of Projects

Another central organizing concept of `arch` is a global namespace of projects.

This chapter describes how projects are named, and how the names are applied to project trees.

8.1 The Structure of Project Names

The `arch` project name-space is designed to reflect the intricacies of how projects evolve in the real world. For example, two or more different organizations may separately develop and distribute a given project. A project can split into multiple development paths. Each path typically evolves through a series of versions. `arch` provides a way to precisely name all of these different instances of a given project.

An `arch` project name has four basic parts, introduced here, and explained in detail below.

An Archive Name The *archive name* identifies the organization that distributes particular versions of a project. Some example archive names might be:

```
hackerlab@regexps.com--hackerlab
hurd@gnu.org--hurd-team
```

The Category Name The *category name* is a generic name for the project. It is what people usually think of as a "project name". Some example category names are:

```
arch
gcc
rx
```

The Branch Label When a project splits into multiple development paths (even if only temporarilly – as a convenience to the maintainers), each path is given a *branch label*. Some example branch labels are:

```
development
experimental
```

The Version Number `arch` uses fairly simple *version numbers*, consisting of a major and minor version number:

```
1.2
2.0
```

The four parts of a project name fit together this way:

```
<archive-name>/<category>[--<branch-label>]--<version-number>
```

As in the example:

```
hackerlab@regexps.com--hackerlab/arch--development--2.0
                        ^            ^         ^        ^
                        |            |         |        |
                  archive name    category   branch   version
                                             label
```

Notice that the archive name is separated from the rest of the project name by `/`. The other parts of the project name are separated by `--`.

When you use `arch`, you often abbreviate project names. For example, you can leave off the archive name and a default archive will be presumed:

```
arch--development--2.0
```

or you can leave off the version number and, depending on context, that means either "the latest version" or "all versions":

```
arch--development
```

If you have one branch which is "primary branch", you can leave out the branch label:

```
arch--2.0
hackerlab@regexps.com--hackerlab/arch--2.0
arch
hackerlab@regexps.com--hackerlab/arch
```

Those last two names are also sometimes used to mean "every branch of arch" or "every branch of arch at the hackerlab archive", respectively.

8.2 Archive Names

An *archive name* designates an organization that develops and/or distributes software. Archive names should be globally unique.

Later in the manual, we'll see that archive names are specifically used to identify revision control archives.

An archive name consists of an email address (with complete hostname), followed by –, followed by an additional string of numbers, letters and dashes. Choose an email address which is exclusively yours (or your project's). That way, your archive name(s) will be globally unique. Here is an example:

```
joe.hacker@gnu.org--test-archive
```

If your organization is going to have more than one revision control archive, you'll use more than one archive name:

```
joe.hacker@gnu.org--gcc-archive
joe.hacker@gnu.org--guile-archive
joe.hacker@gnu.org--2001
```

You can choose an archive name to use as the default for all `arch` commands. When you run a command without explicitly specifying a archive, the default is used:

```
% larch my-default-archive joe.hacker@gnu.org--2001
```

To find out the current default:

```
% larch my-default-archive
joe.hacker@gnu.org--2001
```

In general, `arch` sub-commands accept the option `-A` to specify a non-default archive:

```
% larch my-default-archive -A joe.hacker@gnu.org--test-archive
joe.hacker@gnu.org--test-archive
```

8.3 Category Names and Branch Labels

The *category* of a project name identifies, generally, what the project is. The category is the same no matter who is distributing the project, or which development path is being considered.

The *branch label* is optional. It can be used to distinguish alternative development paths for a given project.

The *category* and *branch label* must match the regexp:

```
[[:alpha:]]([[:alnum:]]*(-[[:alnum:]]+)?)*
```

or in other words, they must begin with a letter and consist entirely of digits, letters, and dashes – but must not contain two dashes in a row, and must not end with a dash.

8.4 Version Numbers

In a somewhat arbitrary but extremely traditional way, branches are divided into a series of versions.

The *version number* must match the regexp:

```
[[:digit:]]+\\.[[:digit:]]+
```

or in other words, it must consist of two strings of digits, separated by a single period. The first string of digits is called the version number *major version number* and the second string of digits is called the version number *minor version number*.

Note: *Version numbers* are not *revision numbers*. In other words, when the `arch` revision control system stores multiple snapshots of the development of your project, it does *not* assign a new version number to each snapshot. Instead, each project version is further subdivided into something called "patch levels", which are explained in detail later in the manual.

8.5 Labelling Project Trees

Every project tree may be labeled with a project name using the `set-tree-version` command, as in this example:

```
% cd ~/wd/arch
% larch set-tree-version arch--1.0
```

That project name becomes the default for `arch` operations within that tree.

You can find out the project name of a tree with `tree-version`:

```
% larch tree-version
joe.hacker@gnu.org--2001/arch--1.0
```

Notice that `arch` used the default archive (returned by `my-default-archive`) when `set-tree-version` was invoked. You can also specify an archive explicitly, as in these two examples:

```
% larch set-tree-version joe.hacker@gnu.org--alt/arch--1.0
```

or

```
% larch set-tree-version -A joe.hacker@gnu.org--alt arch--1.0
```

8.6 Combining Project Trees

You can, in fact, combine project trees: storing the files and directories from multiple projects under a common root. This can be useful if you have separately maintained projects that, nevertheless, are tightly integrated. This is explained in a later chapter (xref!!!).

9 Archives

This chapter discusses *archives* – places where revisions can stored and shared. Archives are used to keep a detailed history of how a project evolves. They are used to help a team of developers stay "in sync" – in agreement about what the official, latest revision of a project really is. Archives are used to help coordinate divergent development paths (branches) and to merge changes between branches.

9.1 Archive Names Revisited

Every archive has a name which *should* be globally unique. These names were previously introduced in the context of project names generally (see Section 8.2 [Archive Names], page 44.)

To briefly review, an archive name consists of an email address (with complete hostname), followed by "–", followed by an additional string of numbers, letters and dashes. Choose an email address which is exclusively yours (or your project's). That way, your archive name will be globally unique. Here is an example:

```
joe.hacker@gnu.org--test-archive
```

9.2 Creating a New Archive

To create a new archive on a local file system, use the `make-archive` command:

```
% larch make-archive NAME LOCATION
```

The `NAME` is the name for the archive. The `LOCATION` is a name for a directory that will be created to hold the archive. For example:

```
% larch make-archive \
       joe.hacker@gnu.org--test-archive \
       ~/{test-archive}
```

9.3 Mapping Archive Names to Locations

`arch` maintains a mapping of archive names to archive locations separately for each user, in the directory `~/.arch-params`.

You can see the list of archives for which you have a recorded location by using the `archives` command, as in this example:

```
% larch archives
joe.hacker@gnu.org--test-archive
        /home/joe/{test-archive}
```

The command `register-archive` is used to add, remove, or update the location of an archive:

```
% larch register-archive ARCHIVE-NAME LOCATION
```

records a new archive location or changes an old one. The command:

```
% larch register-archive -d ARCHIVE-NAME
```

removes the record of an archive location.

If you want to know the registered location of a particular archive, you can use:

```
% larch whereis-archive ARCHIVE-NAME
LOCATION
```

9.4 Remote Archives

If an archive directory is accessible via `FTP`, then `arch` can access that archive remotely, simply by specifying an FTP URI for the archive's location:

```
% larch register-archive joe.hacker@gnu.org--test-archive \
        ftp://anonymous@ftp.gnu.org/users/joe/{test-archive}
```

The URI can include a password. When `arch` saves a location in your `.arch-params` directory, it is made unreadable to anyone but you – weak security, but security nonetheless.

9.5 Your Default Archive

Using `arch` is generally simplified by setting a *default archive* – the archive to use by default when no other is specified.

The command:

```
% larch my-default-archive [options] [archive]
```

can be used to set or check your default archive. (This command was previously introduced. See Section 8.2 [Archive Names], page 44.)

In general, `arch` sub-commands accept the option `-A` to specify a non-default archive:

```
% larch categories -A joe.hacker@gnu.org--test-archive
```

10 Development Paths

An `arch` archive is organized around *development paths*. Each development path is a succession of revisions, each (usually) differing from the previous revision by a simple patch.

Every development path has a project name with version number, such as:

```
hello--devo--1.0
```

Project names were previously discussed (see Chapter 8 [The arch Global Name-space of Projects], page 43).

As we'll see in the next chapter, each development path is further subdivided into specific revisions, each having a *patch level name*, as in these examples:

```
hello--devo--1.0--patch-3
hello--devo--1.0--patch-4
hello--devo--1.0--patch-5
```

10.1 Creating a Development Path

Supposing that you have set your default archive, you can create a new development path in three steps:

```
% larch make-category CATEGORY-NAME


% larch make-branch BRANCH-NAME


% larch make-version VERSION-NAME
```

as in this example that creates the version `hello--devo--1.0`:

```
% larch make-category hello
% larch make-branch hello--devo
% larch make-version hello--devo--1.0
```

and this example that creates `hello--1.0`:

```
% larch make-category hello
% larch make-branch hello
% larch make-version hello--1.0
```

Note that in the second example, the branch label is empty.

10.2 Examining an Archive

You can review what's in an archive using the commands:

```
% larch categories
        - print a list of categories

% larch branches CATEGORY-NAME
        - print a list of branches within a category

% larch versions BRANCH-NAME
        -print a list of versions within a branch
```

10.3 Fully Qualified Version Names

Every version is associated with a particular archive. Ordinarilly, if you write a simple version name, your default archive is presumed. However, wherever a version name is called for, you can use a qualified version name *fully qualified version name* of the form:

```
ARCHIVE-NAME/VERSION-NAME
```

as in the example:

```
joe.hacker@gnu.org--test-archive/hello--devo--1.0
```

That can be useful when operating on a remote archive that is not your default archive. For example, you could use the name in the example to retrieve the latest revision from Joe Hacker's archive, in preparation for creating a branch of that version in your local archive.

11 Basic Revision Control

This chapter introduces the fundamental operations for storing revisions in archives, retrieving them, doing clever things with patches, and managing project trees for archived projects.

11.1 The First Revision

When beginning a new project, the first step is to check in the very first revision of the tree.

Prepare the Archive We've already seen how to prepare the archive by creating the category, branch, and version for the project (see **Creating a Development Path** in Chapter 10 [Development Paths], page 51). You have to perform those steps before checking in the first revision:

```
% larch make-category CATEGORY
% larch make-branch BRANCH
% larch make-version VERSION
```

Prepare the Project Tree We've also seen how to turn an ordinary directory into a project tree using the `init-tree` command (see Section 4.1 [Initializing a Project Tree], page 19). And we've seen how to set the default version of a project tree using `set-tree-version` (see Section 8.5 [Labelling Project Trees], page 46). Both of these steps must be completed before checking in the first revision:

```
% larch init-tree
% larch set-tree-version VERSION-NAME
```

In addition, you must create for the project tree a "patch log" – where log entries for revisions in the development path `VERSION-NAME` will be stored:

```
% larch add-log VERSION-NAME
```

Patch logs are explained in detail in the next chapter. For now, just accept the necessity of that command on faith.

Prepare a Log Message In the root of the project tree, the command:

```
% larch make-log
```

creates a template for a log file (look for file whose name begins with `++log`). The details of what should go in a log message are project specific. In general, the format of a log message uses `RFC822` style headers followed by a free-form body. When `arch` stores a log message, it will add some headers of its own.

Archive the Tree Again, in the root of the project tree, use the `import` command to archive the tree:

```
% larch import
```

`import` will print the headers of the log message (including headers added by `arch`), store the tree in the archive (as a compressed tar file), and update the patch log of the working directory to reflect the commit.

After commiting a revision to, say, `hello--devo--1.0`, you can use the command `revisions` to see a list of archived revisions:

```
% larch revisions hello--devo--1.0
base-0
```

`base-0` is the "patch level name" for the first revision. Patch levels are described in greater detail below.

11.2 Successive Revisions

Suppose that you have continued to edit your working directory after checking in the initial revision. Successive revisions can be checked in with the command `commit`, issued at the root of the working directory. As with `import`, you must first use `make-log` to prepare a log message for each commit:

```
% larch make-log
```

```
[...edit log message...]
```

```
% larch commit
[output omitted]
```

After several commits, you'll have a number of patch levels:

```
% larch revisions hello--devo--1.0
base-0
patch-1
patch-2
patch-3
...
```

As a point of interest, the base revision is stored as a compressed tar file of the entire tree. Each `patch-N` is stored as a compressed tar file of just a patch set that describes how to derive that revision from the previous revision.

11.3 Patch Levels and Development Phases

Within a version, there is a sequence of revisions, each of which is a different *patch level*. Every patch level has a *patch level name*, derived from the version name by adding a suffix:

```
hello--devo--1.0--base-0
hello--devo--1.0--patch-1
hello--devo--1.0--patch-2
hello--devo--1.0--patch-3
...
```

Just as with version names, there is also such a thing as a qualified patch level name *fully qualified patch level name*:

```
joe.hacker@gnu.org--test-archive/hello--devo--1.0--patch-3
```

Development within a version may be optionally divided into four phases: *base revision*, *pre-patches*, *version revision*, and *post-patches*. Conceptually, "pre-patches" are revisions made before the version is "done". "Post-patches" are revisions made after the version is done (e.g. "bug fix patches").

At the beginning of a development path is the *base revision* (called `base-0`). Between the pre-patch and post-patch phases is the *version revision* (called `version-0`). Post-patches have names like `versionfix-1`, `versionfix-2`, etc.

So the total sequence of revisions within a development path has the form:

```
The Initial Revision:

        base-0

Pre-Patches:

        patch-1
        patch-2
        patch-3
        ...
        patch-N

The Version Revision:

        version-0

Post-Patches:

        versionfix-1
        versionfix-2
        versionfix-3
        ...
```

Typically, the `version-0` revision is what would be released under the version number and the post-patch revisions are fixes made after the release.

The ordinary `commit` command creates pre-patches (`patch-N` revisions).

To create the version revision, use `commit --seal`:

```
% larch commit --seal
```

After the version revision exists, ordinary commit will no longer work. To create a post-patch revision, use `commit --fix`:

```
% larch commit --fix
```

It is perhaps worth mentioning that patch level names can be sorted easily using:

```
sort -t - -k 1,1 -k 2,2n
```

or in reverse:

```
sort -t - -k 1,1r -k 2,2rn
```

If you are familiar with other revision control systems, a four-phased development process may at first seem somewhat arbitrary and needlessly complicated. Two points are worth mentioning:

First, phased development is entirely optional. Nothing requires you to ever `--seal` a version, and if you never seal a version, you never need to use `--fix`. Instead, every revision (after `base-0`) will be a `patch-N` revision.

Second, phased development is a handy way to prevent accidents when organizing more complex projects. Sealing a version is a way to set a flag that says, in effect, "ordinary development in this version has stopped – you might not really want to make a new revision here." You can make a new revision if you insist (by specifying `--fix`), but the requirement that you insist helps alert you to the fact that your new revision might need to be merged with later versions of the same project; or that that version you are revising has already been released. Phased development is a bookkeeping convenience: for the most part, `arch` treats all revisions equally, regardless of their phase.

11.4 Getting a Revision

To retrieve a revision from an archive, use `larch get`:

```
% larch get REVISION DIR
```

as in:

```
% larch get hello--devo--1.0--patch-4 hello
```

to retrieve the revision `patch-4` and store it in the new directory `hello`.

An abbreviation can be used to obtain the most recent revision of a version:

```
% larch get hello--devo--1.0 hello
```

or to obtain the most recent revsion of the highest-numbered version:

```
% larch get hello--devo hello
```

A fully-qualified name can be used to obtain a revision from someplace other than the default archive:

```
% larch get joe.hacker@gnu.org--test-archive/hello--devo
```

11.5 Optimizing Archives for get

The way that `get` ordinarilly works is that it searches backwards from the desired revision to find the nearest full-source base revision. It gets the compressed tar file for that base revision and creates a source tree. Then, for each intermediate patch level, it gets a compressed tar file of the patch set, uncompresses and un-tars the patch-set, and applies the patch-set to the source tree.

If there are many intermediate patch-sets, that process can be slow. In such cases, you can ask `arch` to cache a full-source copy of an arbitrary revision, with the command:

```
% larch archive-cache-revision [ARCHIVE/]REVISION
```

That command first builds the requested revision, then it builds a compressed tar file of the revision, then it stores the tar file back in the archive. Subsequent attempts to `get` the same revision (or any later revision) will use the cached tree.

To remove a previously cached tree, use:

```
% larch archive-uncache-revision [ARCHIVE/]REVISION
```

For each user, `arch` also maintains a "client side" cache of revisions that can speed up `get` (and other operations). The details of client side caching are documented in a later chapter (xref!!!).

11.6 Finding Out What Changed

Before performing a `commit`, you might want to check to see what has actually changed – that is, find out exactly what patch set your `commit` will create.

You can do that with the command `what-changed`:

```
% larch what-changed
[...patch set report...]
```

`what-changed` computes a patch set between your modified project tree and the latest patch level for which your project tree is up-to-date. In other words, it tells you what changes have been made to your tree compared to the tree in the archive.

`what-changed` leaves behind a directory containing the patch set and patch report, which you can usefully browse by hand.

```
% ls
,,what-changed.arch--devo--0.5--patch-14--lord@regexps.com--arch-1
[...]
```

The default output of `what-changed` is formatted for use with the `outline` mode of **GNU Emacs**.

You can ask `what-changed` to also generate an HTML-formatted report with:

```
% larch what-changed --url
URL of patch report
```

The HTML report has links to each context diff, added, and removed file. The output of the command is a `file:` method URL. For example, a useful command for `netscape` users is:

```
% netscape --remote "openURL(`larch what-changed --url`)"
```

11.7 The whats-missing Command

Suppose that more than one programmer is checking revisions into a version, Alice and Bob for example.

Alice and Bob both start with working directories and both make some changes. Alice commits several changes. Now Bob's working directory has fallen behind archived development path.

The command `whats-missing` can be used to tell Bob which patches he is missing:

```
% larch whats-missing --summary
patch-N
        summary of patch N
patch-N+1
        summary of patch N+1
...
```

For each patch that Bob is missing, `whats-missing --summary` prints the name of the patch and the contents of the `Summary:` header from its log message.

The `whats-missing` command is explained in greater detail in a later chapter (see Chapter 13 [Patch Logs and ChangeLogs], page 67).

11.8 Update

So Bob is behind by a few patches, but also has his own modifications.

Diagramatically, we have something like:

```
Patch Levels    Bob's Working
in the          Directory
Archive:
------------------------------

base-0


patch-1
patch-2
patch-3
patch-4 --------> bob-0 (Bob's initial working directory)
patch-5             |
patch-6             |
patch-7             V
patch-8           bob-1 (Bob's working dir with changes)
```

Bob is missing patches five through eight.

The command `update` can be used to fix the situation:

```
% larch update OLD-DIR NEW-DIR
```

as in:

```
% larch update bob-1 bob-2
```

`update` works in several steps. First, it gets a copy of the latest revision (`patch-8` in this case). It also gets a copy of the revision from which `OLD-DIR` is dervied (`patch-4` in this case). Then it uses `mkpatch` to compute the differences between `OLD-DIR` and its source revision, and applies those differences (using `dopatch`) to the latest revision.

In the example, update will create the directory `bob-2`, with the source:

```
delta(patch-4, bob-1)[patch-8]
```

(For information about this notation, see Appendix C [The Theory of Patches and Revisions], page 125.)

Applying that patch might cause conflicts. In that case, `update` will print a message telling Bob to look for `.rej` files.

As a convenience, `update` also copies all "precious" non-source files from `OLD-DIR` to `NEW-DIR` (see Chapter 5 [arch Project Inventories], page 21).

Of course, if Bob only wanted to "partly update", he could do that with an extra parameter to `update`, as in the example:

```
# update, but only up to patch level 6:
#

% larch update bob-1 bob-2 hello--devo--1.1--patch-6
```

Finally, `update` can replace `OLD-DIR` with the updated directory if given the `--in-place` flag:

```
% larch update --in-place OLD-DIR
```

11.9 Replay

`update` isn't the only way to catch-up with a development path. Another option is `replay`:

```
% larch replay old-dir new-dir
```

Using the same example:

```
Patch Levels    Bob's Working
in the          Directory
Archive:
------------------------------

base-0


patch-1
patch-2
patch-3
patch-4 --------> bob-0 (Bob's initial working directory)
patch-5             |
patch-6             |
patch-7             V
patch-8           bob-1 (Bob's working dir with changes)
```

and the command:

```
% larch replay bob-1 bob-2
```

`replay` will first copy `bob-1` to create `bob-2`. Then it will apply each missing patch in succession until `bob-2` is up-to-date, or until a merge conflict occurs.

Thus, if no conflict occurs, `replay` computes:

```
patch-8 [ patch-7 [ patch-6 [ patch-5 [ bob-1 ]]]]
```

If a conflict occured in, say, `patch-6`, then `replay` would compute:

```
patch-6 [ patch-5 [ bob-1 ]]
```

and after fixing the conflict, Bob could use a second `replay` command to apply patches seven and eight.

As with `update`, `replay` also copies all "precious" non-source files from `OLD-DIR` to `NEW-DIR` (see Chapter 5 [arch Project Inventories], page 21).

Of course, if Bob only wanted to "partly replay", he could do that with an extra parameter to `replay`, as in the example:

```
# replay, but only up to patch level 6:
#

% larch replay bob-1 bob-2 hello--devo--1.1--patch-6
```

You can use `replay` to modify an existing directory rather than creating a new directory:

```
% larch replay --in-place DIR [REVISION]
```

but be careful: if `DIR` contains precious local changes, and conflicts occur, or if you simply decide the `replay` wasn't a good idea, you'll have to do some work to revert the `replay`.

11.10 The Next Version

In simple situations, a version like `hello--devo--1.0` will be followed by the *next version*: `hello--devo--1.1` or `hello--devo--2.0`, for example.

Such a version is called a *continuation* of the previous version and it is created with this sequence of commands (in the root of a working directory):

```
% larch make-version NEXT-VERSION
% larch add-log NEXT-VERSION
% larch set-tree-version NEXT-VERSION
% larch make-log
[...edit log message...]
% larch commit --continuation PREVIOUS-VERSION
```

as in:

```
% larch make-version hello--devo--1.1
% larch add-log hello--devo--1.1
% larch set-tree-version hello--devo--1.1
% larch make-log
[...edit log message...]
% larch commit --continuation hello--devo--1.0
```

Those commands create the `base-0` revision of the new version but instead of storing complete source for the base revision, they store a pointer to the older revision which the base revision is equal to.

On the other hand, if you wanted the next version to start completely from scratch (perhaps because a program is being replaced by an entirely new implementation), the `import` command can create the next version as documented earlier in this chapter.

Finally, there is another command (besides `commit --continuation`) for creating the next version, `larch tag`, described in the next chapter.

12 Basic Branching and Merging

When a single development path splits into two paths, that's called a *branch*. Typically, a branch is followed by a *merge* – adding the changes made in a branch back to the branch from which it diverged.

This chapter explains how to create a branch and the simplest way to merge changes from two branches. Later chapters will explain fancier techniques, useful in more complex situations.

12.1 Tagging

In `arch`, branches are "tags".

A *tag* is an alternative name for some revision. For example, one could use the tag `release-candidate` to name whatever revision people should download for testing purposes, regardless of what branch, version, or patch level the release candidate happens to be on.

In `arch`, tags are revisions on ordinary branches. For example, suppose we are developing `arch--devo--0.5` and want to create a tag `release-candidate` to mark revisions which "early adopters" should test. First, create a development path for the tag:

```
% larch make-branch arch--release-candidate
% larch make-version arch--release-candidate--0.5
```

To tag a specific revision, use the command:

```
% larch tag SOURCE-REVISION TAG-REVISION
```

Note that `source-revision` and `tag-revision` may be abbreviated. For example, to tag the most recent revision of `arch--devo`:

```
% larch tag arch--devo arch--release-candidate
```

or

```
% larch tag arch--devo--0.5 arch--release-candidate--0.5
```

or

```
% larch tag arch--devo--0.5--patch-37 \
            arch--release-candidate--0.5
```

After such a command, you can retrieve the tagged revision in the ordinary way:

```
% larch get arch--release-candidate--0.5
```

Note that when you `get` a tag that way, the default version of the resulting project tree is the tag's version, not the tagged version (see Section 8.5 [Labelling Project Trees], page 46).

You can always update a tag, making it point to a later revision, again using the `tag` command:

```
% larch tag arch--devo--0.5--patch-53 \
           arch--release-candidate--0.5
```

You can see the history of a tag in the usual way, too:

```
% larch revisions --summary arch--release-candidate--0.5
base-0
    tag of joe.hacker@gnu.org--arch/arch--devo--0.5--patch-37
patch-1
    tag of joe.hacker@gnu.org--arch/arch--devo--0.5--patch-53
```

12.2 Creating a Branch

There is another way to create a branch. This technique is slower, but has the side effect of leaving you with a working directory for the new branch. The effect on the repository is the same as when using the `tag` command.

Creating a new branch from an existing branch can be accomplished by this sequence of commands:

```
% larch prepare-branch OLD-REVISION NEW-BRANCH-VERSION WORKING-DIR
% cd WORKING-DIR
[...edit log message...]
% larch finish-branch
```

but it is worth deeply understanding that `prepare-branch` is equivalent to:

```
# This command gets the source tree for the revision we
# are branching from:
#
% larch get OLD-REVISION WORKING-DIR

# These commands switch the working directory to the
# new branch:
#
% cd WORKING-DIR
% larch add-log NEW-BRANCH-VERSION
% larch set-tree-version NEW-BRANCH-VERSION

# These commands create a "continuation" base-revision
# for the new branch:
#
% larch make-log
```

and that `finish-branch` is the same as:

```
% larch make-branch NEW-BRANCH
% larch make-version NEW-BRANCH-VERSION
% larch commit --continuation OLD-REVISION
```

For example, to create a branch `hello--experimental--1.0` from the latest revision of `hello--devo--1.0`, use:

```
% larch prepare-branch hello--devo--1.0 \
                      hello--experimental--1.0 \
                      wd


% cd wd


[...edit log message...]


% larch finish-branch
```

12.3 Distributed Branches

There is no requirement that a branch be stored in the same archive as the revision from which it branched. For example, you can create a private archive, and store some branches there – only merging those changes back into the shared archive when they are ready.

Here's a tip: make your private archive your default archive. Use fully-qualified version and revision names when getting or committing revisions in the shared archive. This makes it less likely that you'll accidently make unintended changes to the shared archive.

12.4 whats-missing Revisited

If you have a project tree for a branch, you might want to know what has happened in the version from which you branched.

The `whats-missing` is used for this. In a working directory for a branch, use:

```
% larch whats-missing --summary ORIGINAL-VERSION
```

where `ORIGINAL-VERSION` is the version name of the version from which you branched. Actually, `ORIGINAL-VERSION` can be any version for which your project tree has a patch log.

The `whats-missing` command is explained in greater detail in the next chapter (see Chapter 13 [Patch Logs and ChangeLogs], page 67).

12.5 update and replay Revisited

Similarly, `update` and `replay` work for any version for which a project tree has a patch log, such as a version from which a branch occurred:

```
% larch update OLD-DIR NEW-DIR [archive/]VERSION

% larch replay OLD-DIR NEW-DIR [archive/]VERSION
```

12.6 Merging After a Branch

The simplest use of branching and merging is this: you have one development path, call it the "trunk". You form a branch from that development path, which we'll just call "the branch".

To make some changes, you do your work on the branch: check out the latest revision from the branch, make changes, commit, make more changes, commit again, etc.

As you work, you might sometimes need to "catch up" to changes made to the trunk. You can do that by using `update` or `replay`.

When you're done, and the branch is fully up-to-date with the trunk, you can check out the latest branch revision, then commit that version to the trunk. All of the changes that you made on the branch will be summarized into a single patch.

There are more complicated and more realistic uses of branches. These are the subjects of the next several chapters.

13 Patch Logs and ChangeLogs

Every project tree has an associated patch log: a collection of log entries for each `commit` or `import` in the history of that patch tree. When you commit a new revision, the log entry you write is saved in two places: it is saved in the archive as a plain text file (for browsing and as a record of complex ancestory relationships), and in the project tree itself (for browsing and to control history sensative merging).

Logs are organized by version. The command:

```
% cd ~/wd/project
% larch add-log [[ARCHVIE/]VERSION-NAME]
```

creates a new (empty) log for the indicated version. (Version names were introduced in Chapter 8 [The arch Global Name-space of Projects], page 43 and their relationship to archived revisions explained in Chapter 10 [Development Paths], page 51.)

The command:

```
% larch logs
```

lists all of the version names for which a project tree has a patch log.

The command:

```
% larch log-ls [[ARCHIVE/]VERSION-NAME]
```

lists all of the patch levels for which a tree has log entries (for revisions in the indicated version). With the `summary` flag:

```
% larch log-ls --summary [[ARCHIVE/]VERSION-NAME]
```

the `Summary:` header of each log entry is printed.

To see the complete text of an entry, use:

```
% larch cat-log [ARCHIVE/]VERSION--PATCH-LEVEL
```

13.1 Branches and Patch Logs

When you form a branch, project trees on the branch have (at least) two patch logs: one for the original development path, and one for the branch itself. When you merge changes from one branch to another, so long as both branches have the same project category, the merged tree has patch logs for both branches. ("Project categories" were introduced in Chapter 8 [The arch Global Name-space of Projects], page 43.)

13.2 Comparing Patch Logs to Archives

You can find out if an archive contains patches that haven't yet been applied to your project tree with this command:

```
% larch whats-missing [[ARCHIVE/]VERSION ...]
<list of missing patches>
```

That command compares the patch log stored in the archived with the patch log found in the project tree and prints the list of missing patches. There may be missing patches if your tree is not up-to-date with respect to the archive, or if when your tree was created, some patches were skipped.

You can see the `Summary:` line of each missing patch with:

```
% larch whats-missing --summary [[ARCHIVE/]VERSION ...]
```

If you want the list to contain fully-qualified patch level names, use the `--full` option.

If you want to know where branch A stands in relation to branch B, one way to find out is with:

```
% larch get A dir

% cd dir

% larch whats-missing B
```

(It is possible to obtain the same information without having to `get` a revision from branch A, using commands already introduced, plus some other shell commands. The details are left as an exercise for the interested reader.)

13.3 ChangeLogs

The command `larch changelog` generates a GNU-style `ChangeLog` file from a patch log:

```
% larch changelog
```

or

```
% larch changelog [ARCHIVE/]VERSION
```

The `ChangeLog` file generated for `arch`, for example, might begin:

```
# do not edit -- automatically generated by arch changelog
#
# tag: automatic-ChangeLog--lord@regexps.com--arch-1/arch--devo--0.5
#

2001-12-17  Tom Lord <lord@regexps.com>

    Summary:
      'update' and 'replay' output format and bug fixes

    'udpate' and 'replay' -- structured output and updated argument
    processing for reasonable defaults.

    'replay': copy precious files before (not after) appling patches
    so they are carried along with directory renames in patch sets.

    'dopatch': don't pipe 'larch heading' into 'larch body-indent'.


    modified files:
     ChangeLog src/arch/=TODO
     src/arch/branching-and-merging/replay.sh
     src/arch/branching-and-merging/update.sh
     src/arch/patch-sets/dopatch.sh

2001-12-17  Tom Lord <lord@regexps.com>

    Summary:

[...]
```

Notice the first line which says `do not edit ...`. If a project tree contains files matching either of the patterns:

```
ChangeLog
ChangeLog.*
```

whose first lines contain the string `larch changelog`, the command `larch commit` will automatically update that file before checking-in the new revision; the commands `update` and `replay` will update the change logs after patches have been applied.

When updating a `ChangeLog`, `larch commit` looks for a line that says `patch log:` and uses the corresponding patch log to update the ChangeLog.

This is especially useful when development takes place on branches that are later to be merged with a primary branch. The top-level `ChangeLog` file always remains on the primary development path. Each branch can add its own file:

```
ChangeLog.BRANCH--VERSION
```

When it comes time to merge, a log entry for the merge `commit` can be written by editting the `ChangeLog.BRANCH--VERSION` file. A useful idiom is:

```
% larch tag BRANCH-FROM BRANCH-TO
[....]

% larch get BRANCH-TO DIR
[....]

% cd DIR

% larch changelog > ChangeLog.BRANCH-TO

% larch make-log
[...edit log file...]

% larch commit
```

14 Star Topology Branching and Merging

A common way to use branches is to form a star topology. At the center is a *trunk* or "primary development path". A number of branches form a "star" around the trunk. There might be one branch for each developer or sub-team; one branch for each large task; one branch for each change reviewer; one branch for each category of tasks, etc. Here's a (fictional) example of using a star topology to work on various aspects of `GNU emacs`:

```
emacs--display                emacs--intl
  (for hacking on               (for lisp-level character-set
   display features)             support)
              \                /
               \              /
                emacs--main (the trunk)
               /              \
              /                \
      emacs--bugs           emacs--guile
        (for simple bug       (for preparing to use
          fixes)                   Guile Scheme)
```

When using a star topology, developers make changes on the branches, merging those changes into `emacs-main` at significant milestones. Periodically, they merge the collected changes on `emacs-main` back to the development branches. The trunk is the official development sources. Work takes place on the branches, and the developers use the trunk to stay in-sync.

This results in a situation where each branch has merged with `emacs-main` multiple times, and `emacs-main` with each branch several times. In such situations, simple `update` and `replay` are inadequate for performing merges.

This chapter illustrates the problem with using `update` and `replay` for merging in a star topology, explains the solution in general terms, and finally presents `star-merge`, a command that implements the solution.

14.1 The Star Topology Merge Problem

Consider the trunk and one of its branches after several revisions and merges on each:

```
          emacs--main                          emacs--display
          -----------                          --------------
               A-0  --------------------> B-0 (a tag)
               A-1                        B-1
               A-2          ,-----------> B-2 (merge w/ A-1..3)
               A-3  --------              B-3
               A-4          ,------------ B-4
  (merge w/    A-5  <-------              B-5
   B-0..4)     A-6          ,-----------> B-6 (merge w/ A-4..7)
               A-7  --------              B-7
               A-8                        B-8
```

Now, suppose we have a working directory that was checked-out from `A-8` and perhaps contains local modifications. Our goal is to merge `B-8` with that working directory, giving a revision that is up-to-date with both branches and that includes any local changes.

```
% larch update --in-place --dir WD emacs--display
```

modifies WD such that:

```
WD := delta(B-4, WD) [B-8]
```

Well, what has changed in `WD` since `B-4`? `B-4` was up-to-date up to `A-3`, but `WD` contains additional changes from `A-4..A-8` – all of those changes are included in the delta computed by `update`.

`update` applies that delta to `B-8`. Unfortunately, `B-8` already includes the changes in `A-4..7` – so conflicts are guaranteed.

We can't use `replay` either:

```
% larch replay --in-place --dir WD emacs-display
```

tries to apply all the patches missing from `B`. For `WD`, those patches are `B-5..B-8`. Unfortunately, patch `B-6` includes all the changes from `A-4..7` (perhaps with additional changes or conflict resolutions). `WD` already has all of those same changes, so again, conflicts are guaranteed.

14.2 Solving the Star Topology Merge Problem

In the case of the example above, there are three reasonable ways to solve the merge problem, depending on the relative precedence we want to give the two development paths and the working directory.

If we want to give precedence to the changes already in the working directory, we can perform the merge by computing:

```
WD := delta (A-7, B-8) [WD]
```

If we want to give precedence to the changes in `emacs--display`, we can perform the merge by computing:

```
WD := delta (A-7, WD) [B-8]
```

Finally, we might want to merge in two steps: first merging `emacs--display` with `A-8` (giving precedence to `emacs--display`), then adding the local changes of `WD` to that:

```
tmp :=  delta (A-7, B-8) [A-8]
WD := delta (A, WD) [tmp]
```

There are in fact six different ways of precedence-ordering the two branches and `WD` however one of the above solutions works for each of the six possibilities.

Three different solutions are needed if `WD` was checked out from `emacs--display` and our goal is to merge in changes from `emacs--main`. These are:

```
1:      WD := delta (A-7, A-8) [WD]


2:      tmp := delta (A-7, A-8) [B-8]
        WD := delta (B-8, WD) [tmp]

3:      WD := delta (A-7, WD) [A-8]
        (this is an ordinary ‘update’)
```

In the two two-step solutions, legitimate conflicts can occur while building the `tmp` tree: its necessary to resolve such conflicts by hand before performing the second step.

Suitably abstracted, those six merge techniques are sufficient to solve all star topology merging problems with a choice of branch and local change precedence without generating any spurious merge conflicts.

14.3 The star-merge Command

The `star-merge` command figures out how to solve a star topology merge problem and performs the merge. Its syntax is:

```
% larch star-merge [--in-place] A B C [output-dir]
```

where `A`, `B`, and `C` are the names of two branches and a working directory (in any order). (If your working directory name can be mistaken for a branch name, and most of them can, you should prefix the directory name with `./`).

With the `--in-place` option, the working directory is directly modified or else completely replaced with the result. Otherwise, the merge is stored in a new directory (`output-dir`).

The order of arguments determines the precedence of changes. `A`, `B` and `C` have a comman ancestor revision – call it `X`. The order of arguments means (conceptually): start with revision `X`, add the changes from `A`, then the changes from `B`, then the changes from `C`. The specific ordering of changes determines, when there are conflicts, which branch's code is automatically encorporated, and which is left as `.rej` files.

If the merge requires two steps, and conflicts occur in the first step, `star-merge` will stop after the first step leaving the output directory in an intermediate state. After fixing the conflicts, you can complete such a merge with:

```
% larch star-merge --finish [directory]
```

NOTE: In the current release, there is a limitation on star merge. Suppose that `B` is a branch of `A` and that `B` and `A` have never been previously merged. In order to merge them, you must use a working directory checked out from `B` (the branch version) not from `A` (the branched-from version). This restriction will be removed in the next release.

15 Writing Log Entries for Merges

Suppose you have used `star-merge` or some other commands to merge two or more branches. You are ready to commit those changes, but first you have to write a log message.

All of the changes you've just merged are already explained in existing log messages. Your new log message can simply summarize the changes, and point to the more detailed log entries. `arch` has commands which help with this.

The command:

```
% larch new-on-branch [--reverse] [--dir .] VERSION
```

reports a list of patches found in a project tree, but not previously merged with `VERSION`. Those patches are the ones that were added by your merging activity.

You can print a summary of the changes made by those patches by using `xargs` and the `log-ls` command. For example, the main development path for arch 1.0 is called `arch--devo--1.0`. I develop new features on various branches. When it comes time to write a log message before committing a merge to `arch--devo`, I use:

```
% larch new-on-branch --reverse arch--devo--1.0 \
  | xargs larch log-ls --full --summary

lord@regexps.com--2002/arch--lord--1.0--patch-14
    'merge-points' fixes and 'new-on-branch' speed-up
lord@regexps.com--2002/arch--lord--1.0--patch-13
    Use --dir consistently.
lord@regexps.com--2002/arch--lord--1.0--patch-12
    output format touch-ups
```

That idiom is captured, along with a little bit of formatting, by the command `log-for-merge`:

```
% larch log-for-merge arch--devo--1.0


Patches applied:


* lord@regexps.com--2002/arch--lord--1.0--patch-14
    'merge-points' fixes and 'new-on-branch' speed-up

* lord@regexps.com--2002/arch--lord--1.0--patch-13
    Use --dir consistently.

* lord@regexps.com--2002/arch--lord--1.0--patch-12
    output format touch-ups
```

(An advantage of using the formate generated by `log-for-merge` is that it is understood by other `arch` commands which automatically format web pages from `ChangeLogs`.)

That works out nicely as part of a log message if you organize `ChangeLogs` as I do. In the top-level of the `arch` source code, I have an automatic `ChangeLog` for the `arch--devo--branch`:

```
arch/
  arch/ChangeLog
```

I created that once, using:

```
% larch changelog arch--devo--1.0 > ChangeLog
```

and after that, it is automatically updated with every `commit`, `update`, or similar operation.

I also have per-branch `ChangeLogs`:

```
arch/
  arch/ChangeLog.d/
    arch/ChangeLog.d/lord@regexps.com--2002/
      arch/ChangeLog.d/lord@regexps.com--2002/ChangeLog.lord--1.0
      arch/ChangeLog.d/lord@regexps.com--2002/ChangeLog.lord-doc--1.0
```

Note that `lord@regexps.com...` is the name of my archive. Two branches that I develop on are:

```
arch--lord--1.0
arch--lord-doc--1.0
```

and each of those has their own ChangeLog. I created those logs with similar commands. For example:

```
% larch changelog arch--lord--1.0 > ChangeLog.lord
```

To write the log entry for a merge, I start with the output of `log-for-merge` application shown above, and add a tiny bit of extra text for people browsing the ChangeLogs as plain text:

```
Summary: merge with 'lord'
Keywords:

Merge with branch 'lord'.

For details, see

    Changelog.d/lord@regexps.com--2002/ChangeLog.lord--1.0

Patches applied:

* lord@regexps.com--2002/arch--lord--1.0--patch-14
     'merge-points' fixes and 'new-on-branch' speed-up

* lord@regexps.com--2002/arch--lord--1.0--patch-13
     Use --dir consistently.

* lord@regexps.com--2002/arch--lord--1.0--patch-12
     output format touch-ups
```

Note that for each of the `Patches applied`, there is a corresponding `ChangeLog` entry in `ChangeLog.d`.

16 Arbitrary Patching with delta-patch

A general purpose, though low-level tool for merging is the `delta-patch` command:

```
% larch delta-patch FROM TO UPON OUTPUT-DIR
```

computes a patch set from `FROM` to `TO` and applies that patch set to `UPON` storing the result in `OUTPUT-DIR`.

The arguments `FROM`, `TO`, and `UPON` can be the names of (any) revision, or the directory names of project trees.

If `UPON` is a project tree, you can modify that tree directly with the `--in-place` option:

```
% larch delta-patch --in-place FROM TO UPON
```

Note: You pretty much have to know what you're doing to use this command. Normally, you should use `star-merge`, `update` or `replay`.

17 Multi-Branch Merging – The reconcile Command

History sensative merging with `replay` can avoid some avoidable merge conflicts, but not all. One example is a class of merge problems that we'll name "Repeated Multi-Branch Merging": the problem of merging several branches when each of the branches have previously merged with some of the others. Although this kind of merging seems arcane, it can, in fact, easily arise in quite realistic situations (for example, when simultaneously supporting multiple releases of a single project).

Below is an example to illustrate the problem. The set-up in this example is a bit long, but each step along the way is perfectly reasonable, and the end result is quite a tangled knot. The pay-off will be seeing how to cope with the resulting mess.

17.1 The Repeated Multi-Branch Merge Problem

Imagine that we start with a particular version of a particular branch, call it `X`. We'll begin at a particular revision in that branch version: `X-1` (for the purposes of this explanation, calling the revision `X-1` is much less cumbersome than using a real revision name like `foo--mumble--3.5--patch-24`).

Three programmers each form their own branch from `X-1`: call them `A`, `B`, and `C`:

```
                ---> A-0
               |
    X-1 ----+---> B-0
               |
                ---> C-0
```

The plan here is develop on each branch, then merge the changes together to create a new revision of `X`.

Programmer `B` starts off, and creates a series of revisions. Simultaneously, `A` creates a project tree and starts making local changes for his first revision:

```
                ---> A-0
               |      \
               |       A's project tree with local changes
               |
    X-1 ----+---> B-0 -> B-1 -> B-2 -> B-3
               |
                ---> C-0
```

Programmer `A` wants to develop on top of those three patches from `B`, and so forms a merge. At this stage, `A` can do a simple `update` or `replay` to create a merged project tree:

```
                ---> A-0
               |     \
               |      A's merged project tree
               |                 ^
               |      -----------|---------
               |     /                     \
    X-1 ----+---> B-0 -> B-1 -> B-2 -> B-3
               |
                ---> C-0
```

Let's assume that `A`'s merge involved some conflicts: `B`'s code has been slightly rearranged in the merged tree. Now `A` can check in that revision. Meanwhile, `C` starts work:

```
                 ---> A-0 -> A-1
                |             ^
                |             |
                |      -------|-------------
                |     /                     \
X-1 ----+---> B-0 -> B-1 -> B-2 -> B-3
                |
                |       C's project tree with local changes
                |      /
                 ---> C-0
```

`C` decides it would be a good idea to merge with the feature's found in `A-1`. In doing so, he'll also be picking up `B-0..3`. Once again, a simple `update` (or `replay`) is sufficient at this point, though to keep things interesting, we'll again assume that there are conflicts to resolve during the update. And meanwhile, by the way, `B` works on his next patch, and `A` commits a new revision:

```
                 ---> B-0 -> B-1 -> B-2 -> B-3
                |     \                     /\
                |      --------------------- |
                |             |              V
                |             |         B's project tree
                |             V
X-1 ----+---> A-0 -> A-1 -> A-2
                |             |
                |             V
                |       C's merged project tree
                |       ^     |
                |      /      V
                 ---> C-0 -> C-1
```

We're nearly done with the set-up: `B` decides to merge in the changes in `A-2`. This is a slightly interesting merge (although not the primary topic of this chapter). The common ancestor of `A-2` and `B`'s project tree is `B-3`. We previously assumed that when `A` updated against `B-3` there were conflicts that had to be resolved by hand. `B` has a choice. He can use `update` against `A-2` to create a new tree:

```
delta (B-3, B's project tree) [A-2]
```

giving priority to `A`'s resolution of those merge conflicts. Or, he can commit his project tree, `get` revision `A-2`, and update against the committed project tree (or do an equivalent thing by hand, without comitting, using `mkpatch` and `dopatch`):

```
delta(B-3, A-2) [B's project tree]
```

giving priority to `B`'s code, and reconsidering the merge conflicts that `A` handled. The choice is arbitrary and the best answer depends on the particular changes made. `B` might want to experimentally try both merges (perhaps in a scratch repository) before picking one. Either way, after a commit, we'll have something like:

```
         ---> B-0 -> B-1 -> B-2 -> B-3 -> B-4 -> B-5
        |                            /            ^
        |                           /            /
        |                          /            /
        |               ----------             /
        |              /       ---------------
        |             /       /
        |            V       /
X-1 ----+---> A-0 -> A-1 -> A-2
        |             |
        |             V
         ---> C-0 -> C-1


or in English:


A-1 is up-to-date with respect to B-3
B-5 is up-to-date with respect to A-2
C-1 is up-to-date with respect to A-1 and (therefore) B-3
```

Finally, let's assume that the main development path, `X`, has evolved independently of these three branches, and that `A` has added a few more revisions:

```
         ---> B-0 -> B-1 -> B-2 -> B-3 -> B-4 -> B-5
        |                            /            ^
        |                           /            /
        |                          /            /
        |               ----------             /
        |              /       ---------------
        |             /       /
        |            V       /
X-1 ----+---> A-0 -> A-1 -> A-2 -> A-3 -> A-4
 |      |             |
 V      |             V
X-2      ---> C-0 -> C-1
```

17.2 The Challenge

Whew. What an (unfortunately plausible) mess. Now for the challenge:

```
Create X-3, which is up-to-date with A-4, B-5, and C-1
```

There is no one right answer to the challenge: no elegant solution that is guaranteed to avoid merge conflicts. Indeed, there are many ways to perform the merge which differ in terms of what conflicts they'll produce. The goal of `arch` is to arm programmers with plenty of tools to understand the situation, explore, generate and apply patches effectively, and find a reasonable solution with the greatest degree possible of automated assistance.

17.3 The Simple update Solutions

Simple `update` gives us a whole collection of simplistic solutions. For example, `X` could update against `A`, then `B`, then `C` or:

```
intermediate-1 := delta (A-4, X-1) [X-2]
intermediate-2 := delta (B-5, X-1) [intermediate-1]
X-3-candidate  := delta (C-1, X-1) [intermediate-2]
```

That `update` path has some problems, though. `delta (A-4, X-1)` includes the changes in `delta (A-1, X-1)`, and so does `delta (B-5, X-1)`. So creating `intermediate-2` will involve redundant patching and plenty of opportunities for conflicts. Similar problems occur when creating `X-3-candidate`.

`X` could try doing the `updates` in a different order, but similar problems will still occur.

17.4 The Simple replay Solutions

`X` could `replay` the branches in some order. Suppose he `replays` `A`, then `B`, then `C`:

```
intermediate-1 := A-4 [ A-3 [ A-2 [ A-1 [ A-0 [ X-2 ]]]]]
intermediate-2 := B-5 [ B-4 [ intermediate-1 ]]
X-3-candidate  := C-1 [ C-0 [ intermediate-2 ]]
```

History sensitivity helped a bit there: `replay` knows better than to apply `B-0..3` – eliminating one source of needless conflicts. Still, when we replay `B-5` and `C-1`, there will be plenty of conflicts to make up for that.

It's also worth mentioning that that this solution involves applying nine different patches: we can do better. By differently ordering the `replay` solution, we get by with fewer patches (replay `C` first, then `A`, then `B`, for example). Figuring out the best order in which to apply patches is, ultimately, the subject of this chapter:

17.5 The reconcile Solution

Suppose that X asks, of the tree X-2:

```
% larch whats-missing A B C
```

the answer is:

```
A-0
A-1
A-2
A-3
A-4
B-0
B-1
B-2
B-3
B-4
B-5
C-0
C-1
```

X can also ask the more interesting question:

```
% larch whats-missing --merges A B C
```

which will answer not only what patches are missing, but what patches include other patches:

```
A-0 A-0
A-1 A-1
A-1 B-0
A-1 B-1
A-1 B-2
A-1 B-3
A-2 A-2
A-3 A-3
A-4 A-4
B-0 B-0
B-1 B-1
B-2 B-2
B-3 B-3
B-4 B-4
B-5 B-5
B-5 A-0
B-5 A-1
B-5 A-2
C-0 C-0
C-1 C-1
```

```
C-1 A-0
C-1 A-1
C-1 B-0
C-1 B-1
C-1 B-2
C-1 B-3
```

X can pipe that list into a filtering command, `larch reconcile`, which does some magic (the trick is revealed below):

```
% larch whats-missing --merges A B C \
  | larch reconcile
  C-0
  C-1
  B-4
  B-5
  A-3
  A-4
```

which means that X can perform the merge with just:

```
A-4 [ A-3 [ B-5 [ B-4 [ C-1 [ C-0 [ X-2 ]]]]]]
```

There is still a potential source of conflicts – when applying `B-5` in this case – but the patch set is as small as possible (six patches instead of our earlier nine), and the sources of conflicts are as few as possible.

How did `reconcile` find that solution? What's the magic? Conceptually, `reconcile` works in two steps.

First, `reconcile` computes a subset of all the patches: the *necessary patches*. The set of necessary patches is the smallest set of patches which, applied in some order, is sufficient to bring the tree up to date. (Proof that there is a unique smallest set of patches with that property is left as an exercise for the interested reader.)

Second, `reconcile` repeatedly selects the next "necessary" patch to apply, until none are left. At each step of this loop, candidates for the next patch to apply are the patches all of whose prerequisites are in place. Of those, the next patch is the one that comes first in the first column of the input to `reconcile`.

So, you don't believe this obscure command is useful in real life? See **Even/Odd Versions** in Appendix B [Implementing Development Policies], page 115.

18 Synchronizing Two Branches

Suppose that you have two branches, for example a primary development line and a developer branch from that trunk. For one reason or another, none of those changes which are on the developer branch but not the trunk are needed – perhaps similar changes have recently been made on the trunk.

Now the developer wants to merge the trunk back to his branch, but the desired outcome of this merge is quite trivial: he wants the source in the branch to wind up looking exactly like the source in the trunk, but the patch log in the branch to have all log entries from both the trunk and the branch. In short, it should appear that after diverging, the branch was later editted to agree with the trunk on how to make the branch's changes.

There is a command for this situation:

```
% larch make-sync-tree from-revision to-version [dir]
```

This creates a project tree for `to-version` whose source code is exactly the same as `from-revision` but whose patch log is a combination of the latest patch log from `to-version` and the patch log of `from-revision`.

The project tree thus created can be committed to `to-version`, resulting in in a perfect synchronization between two branches.

19 Multi-Tree Projects

Large projects are often most naturally divided into a number of independently maintained sub-projects. For example, `arch` itself consists of six separate projects:

```
./src
        A generic top-level source file for combining
        projects.

./src/arch
        arch itself.  That is, the 'arch' command and all of
        the shell scripts that implement it.

./src/file-utils
        Generic utilities for operating on files and
        directories.

./src/ftp-utils
        A scriptable FTP client.

./src/hackerlab
        The hackerlab C library.

./src/text-utils
        Generic utilities for operating on text files.
```

In an `arch` source tree, each of those sub-directories is a separate project tree (each has its own `{arch}` subdirectory; each was initialized by the command `larch init-tree`; each is in a separate category in the repository).

To put all the pieces together, use the `arch` commands *configuration commands*:

(TO BE DOCUMENTED:)

Config file format, e.g.:

```
./configs/regexps.com/devo.arch:

#
# Check out an arch distribution from the devo branches.
# Latest revisions.
#

./src                   lord@regexps.com--2002/package-framework--devo
./src/arch              lord@regexps.com--2002/arch--devo
./src/file-utils        lord@regexps.com--2002/file-utils--devo
./src/ftp-utils         lord@regexps.com--2002/ftp-utils--devo
./src/hackerlab         lord@regexps.com--2002/hackerlab--devo
./src/shell-utils       lord@regexps.com--2002/shell-utils--devo
./src/text-utils        lord@regexps.com--2002/text-utils--devo
```

and

```
% larch build-config --help
instantiate a multi-project tree
usage: build-config [options] config-name
```

```
 -V --version                 print version info
 -h --help                    display help

 --silent                     no output (except odd errors)
 --quiet                      brief output
 --report                     default output
 --verbose                    maximal output
 --debug                      debugging output
 -d --dir DIR                 cd to DIR first

Build the named configuration based on the specification
in the "configs" subdirectory at the root of the project
tree.
```

and

```
% larch update-config --help

update a multi-project tree
usage: update-config [options] config-name

 -V --version                 print version info
 -h --help                    display help

 --silent                     no output (except odd errors)
 --quiet                      brief output
 --report                     default output
 --verbose                    maximal output
 --debug                      debugging output

 -d --dir DIR                 cd to DIR first


 --force                      pass the --force flag to update


Update the named configuration based on the specification
in the "configs" subdirectory at the root of the project
tree.
```

and

```
% larch replay-config --help

replay a multi-project tree
usage: replay-config [options] config-name

 -V --version                 print version info
 -h --help                    display help

 --silent                     no output (except odd errors)
 --quiet                      brief output
 --report                     default output
 --verbose                    maximal output
 --debug                      debugging output
```

```
 -d --dir DIR                  cd to DIR first

Update the named configuration based on the specification
in the "configs" subdirectory at the root of the project
tree.
```

and

```
% larch record-config --help

record a revision-specific configuration
usage: record-config [options] config-name new-config-name

 -V --version                  print version info
 -h --help                     display help

 --silent                      no output (except odd errors)
 --quiet                       brief output
 --report                      default output
 --verbose                     maximal output
 --debug                       debugging output

 -d --dir DIR                  cd to DIR first


 -f --force                    overwrite an existing config

Update the named configuration based on the specification
in the "configs" subdirectory at the root of the project
tree.
```

and

```
% larch show-config --help

show the revision frontier of a configuration
usage: show-config [options] config-name

 -V --version                  print version info
 -h --help                     display help

 --silent                      no output (except odd errors)
 --quiet                       brief output
 --report                      default output
 --verbose                     maximal output
 --debug                       debugging output

 -d --dir DIR                  cd to DIR first

Print the revision frontier specified by the named configuration
in the "configs" subdirectory at the root of the project
tree.
```

20 arch Web Site Generation Tools

To be documented:

```
% larch make-library-browser --help
 create or update a library browser
 usage: make-library-browser [options] output-dir [limit]

  -V --version                  print version info
  -h --help                     display help

  --silent                      no output (except odd errors)
  --quiet                       brief output
  --report                      default output
  --verbose                     maximal output
  --debug                       debugging output

  -t --title TITLE              set the page title
  --dists FILE                  specify a distributions spec file
  --configs FILE                specify a configurations spec file

 Create or update web pages describing a revision library.
```

and

```
% larch update-distributions --help
build or update an FTP area
usage: update-distributions [options] [archive/]revision dist-name ftp-dir

 -V --version                  print version info
 -h --help                     display help

 -R --root root                specify the local archive root
 -A --archive archive          specify the archive name
  --cache DIR                  cache directory for locally cached
                                 revisions

 --config NAME                 after checking out the revision, pass
                                NAME to "build-config" to complete
                                the source tree.

 --silent                      no output (except odd errors)
 --quiet                       brief output
 --report                      default output
 --verbose                     maximal output
 --debug                       debugging output

Ensure that FTP-DIR exists and is populated with the indicated revision
from the archive.

The tar file will be named:

      DIST-NAME-${VERSION}${SUFFIX}
```

where VERSION is the version number of the revision being packaged, and SUFFIX is derived from the patch-level name of the revision.

The value of ${SUFFIX} depends on the revision's patch level, as in these examples:

```
revision:                              tar file:


arch--release--1.0--base-0             arch-1.0pre0.tar.gz
arch--release--1.0--patch-3            arch-1.0pre3.tar.gz
arch--release--1.0--version-0          arch-1.0.tar.gz
arch--release--1.0--versionfix-5       arch-1.0.5.tar.gz
```

The top-level directory in the tar file has the same name as the tar file (but without the ".tar.gz" suffix).

21 The Pristine Revision Cache

Each project tree contains a cache of *pristine* revisions: revisions extracted from an archive and stored without modifications. This cache is used to speed up operations such as `commit`.

It is occasionally useful to know what revisions are cached in a given tree, and to add or remove cached revisions.

commands to doccument

```
% larch pristines --help

% larch add-pristine --help

% larch delete-pristine --help
```

22 Revision Tree Libraries

For many purposes, it is useful to have a library containing pristine trees of a large number of revisions – for example, all of the revisions in a particular version. To be practical, though, such a library must be represented in a space-efficient way.

Unix hard-links provide a natural way to store such a library. Each successive revision in a series is a copy of the previous, but with unmodified files shared via hard-links.

`arch` provides commands to help you build, maintain, and browse such a library.

22.1 Your Revision Library Location

To begin a new revision library, first create a new directory (`DIR`) and then register its location:

```
% larch my-revision-library DIR
```

You can check the location of your library with:

```
% larch my-revision-library
```

or unregister it with:

```
% larch my-revision-library -d
```

22.2 Revision Library Format

A revision library has subdirectories of the form:

```
ARCHIVE-NAME/CATEGORY/BRANCH/VERSION/REVISION/
```

Each `REVISION` directory contains the complete source of a particular revision, along with some supplemantary subdirectories and files:

```
REVISION/,,patch-set/

        The patch set that creates this revision from
        its ancestor (unless the revision is a full-source
        base revision).
```

Although the permissions on files in the revision library are determined as determined by patch sets, **you must never modify files int the revision library**. Doing so will cause odd errors and failures in various `arch` commands.

22.3 Adding a Revision to the Library

You can add a selected revision to your revision library with:

```
% larch library-add REVISION
```

That will add not only REVISION to the library, but all directly preceeding revisions (recursively) which are either from the same archive, or from other archives which are also in the same library. ("Directly preceeding" in this case means the previous patch level, for ordinary revisions, or the tagged revision (for revisions which are tags).)

22.4 Finding a Revision in the Library

You can find a particular revision in the library with `library-find`:

```
% larch library-find REVISION
PATH-TO-REVSION
```

The output is an absolute path name to the library directory containing the revision. (Once again, you must not modify files in that directory.)

22.5 Removing a Revision from the Library

To remove a particular revision from the library, use:

```
% larch library-remove REVISION
```

Be aware of the following limitation in the current release: suppose that you add three successive revisions, `A`, `B`, and `C`. Then you remove `B`, then re-add `B`. Now there is a chance that the file sharing between `B` and `C` will be less than optimal, causing your library to be larger than it needs to be. (You can fix this by then removing and re-adding `C`.)

22.6 Listing Library Contents

The command `library-archives` lists all archives with records in the library:

```
% larch library-archive
ARCHIVE-NAME
ARCHIVE-NAME
...
```

Similarly, you can list categories, branches, versions, or revisions:

```
% larch library-categories [ARCHIVE]
% larch library-branches [ARCHIVE/CATEGORY]
% larch library-versions [ARCHIVE/BRANCH]
% larch library-revisions [ARCHIVE/VERSION]
```

22.7 Individual Files in the Revision Library

You can locate an individual file in a revision library with:

```
% archive library-file FILE [REVISION]
PATH
```

or obtain its contents with:

```
% archive cat-library-file FILE [REVISION]
...file contents...
```

Both commands accept the options `--tag` and `--this`. With `--tag`, the argument `FILE` is interpreted as an inventory tag, and the file with that tag is found.

With `--this`, `FILE` is interpreted as a file relative to the current directory, which should be part of a project tree. The file's inventory tag is computed and the corresponding file found in `REVISION`.

22.8 Determining Patch Set Prerequisits

```
% larch touched-files-prereqs REVISION
```

That command looks at the patch set for `REVISION` and at all preceding patch sets in the same version (it searches your library rather than your repository for this purpose). It reports the list of patches that touch overlapping sets of files and directories – in other words, it tells you what patches can be applied independently of others. The command has an option to exclude from consideration file names matching a certain pattern (e.g. *=README* or *ChangeLog*). It has an option to exclude from the output list patches which have already been applied to a given project tree. It has an option to report the specific files which are overlapped.

23 arch Triggers

In some circumstances, it is very useful to trigger actions upon the detection of changes to an archive. For example, you might want to send an email notification whenever new revisions are checked in.

The command `notify` provides this capability:

```
% larch notify NOTIFICATION-DIR
```

That command reads configuration files found in `NOTIFICATION-DIR`. Those configuration files explain what parts of which repositories to monitor, and what actions to take upon changes. When changes are detected, `notify` invokes actions.

This chapter explains `notify` in general. It documents the format of a `NOTIFICATION-DIR` and the configuration files. It explains what `notify` does. Later in the manual is documentation explaining how to use `notify` to achieve certain specific effects.

23.1 The Four notify Configuration Files

|=rules.archives| |=rules.categories| |=rules.branches| |=rules.versions|

`notify` is controlled by four configuration files, found at the root of a `NOTIFICATION-DIR`. These are:

```
=rules.archives
        Specifies actions to take whenever new categories
        are added to specific archives.

=rules.categories
        Specifies actions to take whenever new branches
        are added to specific categories.

=rules.branches
        Specifies actions to take whenever new versions
        are added to specific branches.

=rules.versions
        Specifies actions to take whenever new revisions
        are added to specific versions.
```

Within these files, blank lines are permitted and comments begin with '#' and continue to the end of line.

The file `=rules.versions` contains lines with the format:

```
VERSION         ACTION
```

where `VERSION` is a fully-qualified version name, and action a string (with no whitespace) of the form:

```
COMMAND:THUNK
```

Whenever new revisions are detected in `VERSION`, `notify` invokes:

```
larch COMMAND THUNK RVN ...
```

where `COMMAND` and `THUNK` are from the configuration file, and the trailing `RVN` arguments are a list of fully qualified revision names of newly detected revisions.

The file `=rules.branches` is similar. It contains lines of the form:

```
BRANCH COMMAND:THUNK VERSION-ACTION
```

Whenever new versions are created in `BRANCH`, `notify` takes two actions. First, it invokes:

```
larch COMMAND THUNK VSN ...
```

Second, it adds new rules to `=rules.versions` of the form:

```
VSN     VERSION-ACTION
```

The files `=rules.categories` and `=rules.archives` are similar. The former contains lines of the form:

```
CATEGORY COMMAND:THUNK BRANCH-ACTION VERSION-ACTION
```

and the latter lines of the form:

```
ARCHIVE COMMAND:THUNK CATEGORY-ACT BRANCH-ACT VERSION-ACT
```

In addition to the configuration files, `notify` maintains its private state in `NOTIFY-DIR`. In particular, it keeps track of what notifications have already been sent, in order to try to avoid sending redundant notifications.

23.2 Triggers for Everything in a Repository

Configuring triggers for every category, branch, and version in an archive is quite simple: you only need to write the `=rules.archives` file – other configuration files will be automatically genrated from that.

The initial `=rules.archives` file should contain lines of the form:

```
ARCHIVE COMMAND:THUNK CATEGORY-ACT BRANCH-ACT VERSION-ACT
```

for ach set of actions you want to trigger. After creating the config file, run `notify` in that directory. The first run will detect all categories in the repository, invoke `COMMAND` for those, and create a `rules.categories` file with lines:

```
CATEGORY CATEGORY-ACT BRANCH-ACT VERSION-ACT
```

Then `notify` will read that config file, detect all the branches in the repository, invoke `CATEGORY-ACT`, and create `=rules.branches`.

Finally, `=rules.versions` will be automatically created in a similar way.

Whenever a new category, branch, or version is added to an archive, the rules files will be automatically updated to reflect the addition (when `notify` is run, of course).

23.3 Robustness Issues with Triggers

Unfortunately, some fundamental physical properties of the universe make it impossible for `notify` to guarantee that actions will be invoked only once for each new category, branch, version, or revision. A (presumably rare) well timed interrupt or system failure can cause `notify` to invoke actions more than once for a given change to the archive.

Consequently, actions should be designed to be robust against that eventuality.

`notify` is designed to be safe in the face of concurrent invocations: if two processes invoke `notify` at the same time, everything should work wonderfully with no resultant redundant actions or `NOTIFY-DIR` corruptions.

23.4 Scheduling Triggers with cron

One strategy for using `notify` is to run it from a `cron` job.

23.5 Scheduling Triggers Directly

These commands (and only these commands) modify arch repositories in ways that are significant to `notify`:

```
make-category
make-branch
make-version
import
commit
```

Corresponding to those commands, are four others:

```
% larch my-category-action COMMAND ARGUMENTS...

% larch my-branch-action COMMAND ARGUMENTS...

% larch my-version-action COMMAND ARGUMENTS...

% larch my-revision-action COMMAND ARGUMENTS...
```

which in those invocations, set an action to be taken upon successful completion of an archive transaction, and in these invocations:

```
% larch my-category-action

% larch my-branch-action

% larch my-version-action
```

```
% larch my-revision-action
```

report the actions to be taken.

The actions have a fairly obvious meaning. After creating a new branch, for example, the command `make-branch` will invoke:

```
% COMMAND ARGUMENTS... BRANCH
```

where `BRANCH` is a fully qualified branch name.

A useful action to take is to invoke `notify` directly. Another useful action (though imposing a greater system administration burden) is to invoke a script which sends mail to some address, informing the recipient of the repository change. The recipient can be, for example, an automated process which invokes `notify`.

When the repository-modifying commands invoke one of these actions, they run the action as a `daemon`: an asynchronous process with no controlling terminal, with standard input redirected from `/dev/null`, and with `stdout` and `stderr` saved in a temporary file. If the command exits with non-0 status, the output from the command is sent to the user in an email notice.

Once again, the damn constraints of physics, particularly the prohibition against "action at a distance", impose a constraint: it is impossible to guarantee the `COMMAND` will be invoked and in (hopefully rare) circumstances, it might not be. `COMMANDs` should be chosen with that constraint in mind.

24 Using Triggers

This chapter describes two ways to use triggers: for sending email about changes to an archive, and for keeping a revision library up to date (see Chapter 22 [Revision Tree Libraries], page 99).

24.1 Sending Email Notices About Repository Changes

`arch` provides four commands designed for use as triggers that send email about changes to a repository. (See Chapter 23 [arch Triggers], page 103.) These are:

```
larch mail-new-categories
larch mail-new-branches
larch mail-new-versions
larch mail-new-revisions
```

You can create a triggers for sending email with a `notify =rules.archives` configuration rule such as this (here split over multiple lines to fit the printed page – in practice, this rule would be on one line):

```
joe.hacker--2002@gnu.org
        mail-new-categories:joe@gnu.org
        mail-new-branches:joe@gnu.org
        mail-new-versions:joe@gnu.org
        mail-new-revisions:joe@gnu.org
```

24.2 Updating a Revision Library

You can use triggers to keep a revision library automatically up-to-date with a rule like this:

```
joe.hacker--2002@gnu.org
        noop:
        noop:
        noop:
        library-add:--ignore-extra-args
```

25 Automatic Triggers

There are exactly six commands which modify repositories:

```
larch commit
larch import
larch make-category
larch make-branch
larch make-version
larch tag
```

Those commands can automatically invoke `larch notify` if you have a default `notify` direcotry set. You can set such a default with:

```
% larch my-notifier DIR
```

check your current default with:

```
% larch my-notifier
```

or delete your default setting with:

```
% larch my-notifier -d
```

Appendix A Getting Started with arch

Here is a whirlwind guide to getting started with `arch`. This appendix presumes that you are familiar with the material in previous chapters.

After building and installing `arch`, try these steps:

A.1 Set Your arch User Id

The first step is to set your id with a command like:

```
% larch my-id "Jane Hacker <jane.hacker@gnu.org>"
```

See Chapter 7 [The arch Global Name-space of Users], page 41.

A.2 Create Your First Archive

Create your own archive with:

```
% larch make-archive NAME DIRECTORY
```

Make that your default archive with:

```
% larch my-default-archive NAME
```

You might want to edit the top-level `=README` file in your archive to describe what you intend to store there. Editting `=README` files is one of the few exceptions to the rule that you should never modify files in the archive "by hand".

See Chapter 9 [Archives], page 47.

A.3 Create Your Revision Library

Revision libraries are strictly optional, but they are desirable if you want to obtain the best performance from `arch` or use all of the available features.

To create a library, simply create an empty directory:

```
% mkdir "{revisions}"
```

and tell `arch` about it:

```
% larch my-revision-library "`pwd`/{revisions}"
```

See Chapter 22 [Revision Tree Libraries], page 99.

A.4 Set Up Your Revision Browser

If you want to be able to browse change sets and past revisions with a web browser, you need to set up a browser directory.

Create an empty directory:

```
% mkdir "{browser}"
```

and mark that directory as a browser:

```
% touch "{browser}/=browser"
```

There is no need to inform `arch` of the location of your browser at this step – that will come later when you set up a "notifier" which updates the browser.

See Chapter 20 [arch Web Site Generation Tools], page 95.

A.5 Set Up Notifiers

If you want your library and browser to be updated automatically as you commit revisions to the repository, you need to set up a *notifier*:

```
% mkdir {notify}

% cd {notify}

[ edit "=rules.archives" ]
```

As an example of the format of *=rules.archives*, here is the one I use. In the documentation, long lines have been broken up with continuation markers (\), but in the real file, they are just very long lines:

```
lord@regexps.com--2002  \
   mail-new-categories:lord \
   mail-new-branches:lord \
   mail-new-versions:lord \
   mail-new-revisions:lord

lord@regexps.com--2002 \
  noop:notify-library:lord \
  noop:notify-library:lord \
  noop:notify-library:lord \
  notify-library:lord

lord@regexps.com--2002 \
  noop:notify-library:lord \
  noop:notify-library:lord \
  noop:notify-library:lord \
  notify-browser:lord
```

The first line defines rules for sending email when new revisions are committed, or when categories, branches, or versions are created.. The second line updates the revision library when new revisions are added to the library. The third line updates the browser whe new revisions are added.

Having set up your notifier directory, inform arch:

```
% larch my-notifier "‘pwd‘/{notify}
```

See Chapter 23 [arch Triggers], page 103 for more information.

A.6 Get to Work

Now you can create branches from other archives, import your own original projects, and otherwise use the various commands documented in this manual.

Additional step by step directions can be found in the *docs/examples* directory of distributions of arch.

Appendix B Implementing Development Policies

Different projects have different policies for managing the "main development path" as distinguished from various kinds of release (such as candidate releases, experimental releases, and stable releases).

`arch` is flexible enough to allow many such policies to be implemented in a direct way. Here are some examples and hints about using `arch`.

B.1 Milestone/Numbered Versions

One development policy is based on *milestones*. The team works on one milestone goal at a time. When they have just about reached that milestone, they make *candidate releases* of the milestone for people to test. After fixing bugs, a *milestone release* is made.

After several milestones, the milestone release gets a version number, and becomes a *version release*.

Beginning from scratch on the first milestone, the developers create the first development path – in which to work on reaching the first milestone:

```
mozilla--devo--1.0
------------------
base-0
patch-1
patch-2
patch-3
patch-4
```

After some time, they are ready to make some candidate releases and enter a bug-fixing cycle. They'll use tags for that – making the bug fixes in the `devo` branch:

```
mozilla--devo--1.0
------------------
base-0
patch-1
patch-2                       mozilla--milestone--1.0
patch-3                       -----------------------
patch-4 --------------------->base-0 (tag -- 1st release candidate)
patch-5              .------->patch-1 (tag -- 2nd candidate)
patch-6 --------------   ---->patch-3 (tag -- 3rd candidate)
patch-7                 |
patch-8 ----------------
```

In the diagram, patches `5..8` in the `devo` branch are fixes made based on reports from candidate releases. We'll suppose that after `patch-8`, the first milestone appears to be stable. The developers want to make the milestone release, and begin work on milestone 2:

```
mozilla--devo--1.0
------------------
base-0
patch-1
patch-2                       mozilla--milestone--1.0
```

```
   patch-3                       -----------------------
   patch-4 --------------------->base-0 (tag -- 1st release candidate)
   patch-5               .------>patch-1 (tag -- 2nd candidate)
   patch-6 --------------   ---->patch-3 (tag -- 2rd candidate)
   patch-7                 |     version-0 (milestone release)
 --patch-8 ----------------
|
|  mozilla--devo--2.0  (development path for the second milestone)
|  ------------------
 ->base-0 (continuation)
```

That pattern can continue indefinately, eventually resulting in a versioned release, (again using tags):

```
   mozilla--devo--1.0            mozilla-milestone--1.0
   ------------------            ----------------------
   ...
 --patch-N---------------------->...
|                                version-0 (milestone 1 release)
|
|
|  mozilla--devo--2.0            mozilla-milestone--2.0
|  ------------------            ----------------------
|  ...
 ->patch-N---------------------->...
|                                version-0 (milestone 2 release)
|
|
|  mozilla--devo--3.0            mozilla-milestone--3.0
|  ------------------            ----------------------
|  ...
 ->patch-N---------------------->...
|                                version-0 (milestone 3 release)
|          ...
|
|  mozilla--devo--10.0           mozilla-milestone--10.0
|  -------------------           -----------------------
|  ...
 ->patch-N---------------------->...
                               --version-0 (milestone 10 release)
                              |
                              |  mozilla-1.0
                              |  -----------
                               ->base-0 (tag)
                                 version-0 (versioned release)
```

That pattern of `arch` usage is very simple because the developers stay in sync, following a strict cycle of working on a milestone, making candidate releases and bug-fixing, making a milestone releases and starting on the next milestone. Less synchronized development can be much more complicated, as illustrated in the next example:

B.2 Even/Odd Versions

Another, more intricate policy is based on version numbers. Odd numbered versions are the "leading-edge" of development – often unstable, but having the very latest sources. Even numbered versions are "stable" – lagging behind the leading-edge, but containing only code known to work reasonably well.

The `arch` concept of a "continuation version" is ideal for this, because there is no requirement that a continuation version be a continuation of the immediately preceeding version.

A series of diagrams can help to illustrate this usage of `arch` and some of the subtleties that can arise. We'll start with just the initial leading-edge development path

```
linux--0.1
----------
base-0
patch-1
patch-2
patch-3
patch-4
version-0
```

At that point, the developers decide to make the first stable release:

```
linux--0.1              linux--0.2
----------              ----------
base-0            ----> base-0 (continuation)
patch-1          |      patch-1
patch-2          |      version-0
patch-3          |
patch-4          |
version-0  ------
```

We can suppose that `patch-1` of `linux--0.2` is just a quick change to the top level `README` file, and that after that – the new stable version is sealed (creating `version-0`) and released. We never particularly want to merge `patch-1` of `linux-0.2` back on to the odd-numbered versions.

Naturally, some bugs are detected in the stable release – three in rapid succession. The developers fix these on the *leading edge* branch, planning to merge them with the stable tree later:

```
linux--0.1              linux--0.2
----------              ----------
base-0            ----> base-0 (continuation)
patch-1          |      patch-1
patch-2          |      version-0
patch-3          |
```

```
        patch-4          |
        version-0  ------
        versionfix-1
        versionfix-2
        versionfix-3
```

They wait a week for some testing to occur. The bug fixes are looking ok, so the developers decide to start work on the next leading-edge version:

```
        linux--0.1              linux--0.2
        ----------              ----------
        base-0            ----> base-0 (continuation)
        patch-1          |      patch-1
        patch-2          |      version-0
        patch-3          |
        patch-4          |
        version-0  ------
        versionfix-1
        versionfix-2
      --versionfix-3
     |
     |
     |  linux--0.3
     |  ---------
      ->base-0 (continuation)
        patch-1
```

Just as the developers are about to merge the bug fixes with the stable version, one more bug report trickles in. Fortunately, it's a trivial bug – so the developers are confident about making the fix in the leading edge, but immediately releasing it in the stable version.

Here's a catch, though – the latest leading edge version (`linux--0.3`) has already diverged from the stable version because of `patch-1`. The developers definately don't want `patch-1` of `0.3` in the the stable `0.2` yet, so they fix the newly reported bug in `0.1`, then merge all four bug fixes with the stable tree in the usual way. Meanwhile, separate work also continues on `0.3`:

```
        linux--0.1              linux--0.2
        ----------              ----------
        base-0            ----> base-0 (continuation)
        patch-1          |      patch-1
        patch-2          |      version-0
        patch-3          |    ->versionfix-1 (merge)
        patch-4          |   |
        version-0  ------    |
        versionfix-1\        |
        versionfix-2 |-------
```

```
 --versionfix-3 |
|  versionfix-4/
|
|
|
|  linux--0.3
|  ---------
 ->base-0 (continuation)
   patch-1
   patch-2
   patch-3
```

It's worth noting at that point that `linux--0.3` is missing a bug-fix (`versionfix-4`) from `linux--0.1`:

```
% cd ~/wd/linux--0.3

% larch whats-missing linux--0.1
versionfix-4
```

We'll eventually do an update to pick up that bug fix, but first, let's make the situation more complicated.

Suppose, some mailing lists get wind of `patch-3` in `linux--0.3`. Soon `Slashdot` and `Newsforge` pick up the story. It turns out that `patch-3` is a very desirable feature and the implementation appears to be clean and stable. People are clammering for its appearence in a stable release, and the developers happen to think it is a good idea.

So, they start the next stable revision:

```
   linux--0.1              linux--0.2
   ----------              ----------
   base-0            ----> base-0 (continuation)
   patch-1          |      patch-1
   patch-2          |      version-0
   patch-3          |    ->versionfix-1 (merge)
   patch-4          |   |     |
   version-0  ------    |     |
   versionfix-1\        |     |
   versionfix-2 |-------      |    linux--0.4
 --versionfix-3 |             |    ----------
|  versionfix-4/               --->base-0 (continuation)
|
|
|
|  linux--0.3
|  ---------
 ->base-0 (continuation)
   patch-1
   patch-2
   patch-3
```

They only want `patch-3` of `0.3` for `0.4` – not anything else. That's a job for `replay --exact`:

```
   linux--0.1              linux--0.2
   ----------              ----------
   base-0            ----> base-0 (continuation)
   patch-1          |      patch-1
   patch-2          |      version-0
   patch-3          |    ->versionfix-1 (merge)
   patch-4          |   |     |
   version-0  ------    |     |
   versionfix-1\        |     |
   versionfix-2 |-------      |    linux--0.4
 --versionfix-3 |             |    ----------
|  versionfix-4/               --->base-0 (continuation)
|                               =->patch-1 (replay --exact merge)
|                              |   version-0
|                              .
|  linux--0.3                  |
|  ---------                   .
 ->base-0 (continuation)       |
   patch-1                     .
   patch-2                     |
   patch-3-=--=--=-=-=-=-=-=-=-
   ...
```

After merging the much-desired `patch-3` of `0.3` into `0.4`, the developers seal `0.4` and make the stable release.

Let's suppose that development on `0.3` continues for a while. After some time, the developers decide that `0.3` has aquired enough new features. They want to do two things: start `0.5`, and start getting the stable `0.6` release ready:

```
   linux--0.1              linux--0.2
   ----------              ----------
   base-0            ----> base-0 (continuation)
   patch-1          |      patch-1
   patch-2          |      version-0
   patch-3          |    ->versionfix-1 (merge)
   patch-4          |   |     |
   version-0  ------    |     |
   versionfix-1\        |     |
   versionfix-2 |-------      |    linux--0.4
 --versionfix-3 |             |    ----------
|  versionfix-4/               --->base-0 (continuation)
|                               =->patch-1 (replay --exact merge)
|                              |   version-0
|                              .
```

```
     |  linux--0.3                 |
     |  ---------                  .
      ->base-0 (continuation)      |
        patch-1                    .
        patch-2                    |
        patch-3-=--=-=-=-=-=-=-=-=-
        ...                          linux--0.6
      -version-0                     ----------
     |
     |
     |
     |  linux--0.5
     |  ----------
      ->base-0
```

Notice that they haven't created the base revision for `0.6` yet. There's a choice here. They could make `0.6` a continuation of `0.4`, then merge in all the patches they're missing from `0.3`. On the other hand, they could make `0.6` a continuation of some `0.3` and pick up all those missing patches "the easy way".

But, oops, when someone checks out `version-0` from `0.3` and runs `whats-missing`, they find out that the `0.3` branch never picked up `versionfix-4` from `0.1`. That's easily fixed by updating a `0.3` tree against `0.1`, and checking in the resulting merge to `0.3`. The resulting merge is also the revision that will become the base revision for `0.6`:

```
        linux--0.1              linux--0.2
        ----------              ----------
        base-0            ----> base-0 (continuation)
        patch-1          |      patch-1
        patch-2          |      version-0
        patch-3          |    ->versionfix-1 (merge)
        patch-4          |   |     |
        version-0  ------    |     |
        versionfix-1\        |     |
        versionfix-2 |-------      |    linux--0.4
      --versionfix-3 |             |    ----------
     |  versionfix-4/               --->base-0 (continuation)
     |           |                   =->patch-1 (replay --exact merge)
     |            -----            |   version-0
     |                |            .
     |  linux--0.3    |            |
     |  ---------     |            .
      ->base-0 (continuation)      |
        patch-1       |            .
        patch-2       |            |
        patch-3-=--=-=-=-=-=-=-=-=-
        ...           |              linux--0.6
      -version-0      V              ----------
     | versionfix-1 (0.1 update)------>base-0 (continuation)
     |                                 version-0
```

```
|
|  linux--0.5
|  ----------
 ->base-0
   patch-1
   patch-2
```

Meanwhile, new work continues on `0.5`.

But now, the `0.5` tree is in an interesting state. If a developer checks out the latest `0.5` and asks:

```
% larch whats-missing linux--0.1
linux--0.1--versionfix-4
```

If the developer asks `whats-missing` from `0.3`, the answer is:

```
% larch whats-missing linux--0.3
linux--0.3--versionfix-1
```

If those two patches were unrelated – there would be no problem: simply update from both branches and check the result into `0.5`.

In fact, though, `versionfix-1` from `0.3` is really the same change as `versionfix-4` from `0.1` (look back at how `versionfix-1` was created). Let's also suppose that when the fix was merged into `0.3`, a slight change had to be made – to resolve a merge conflict.

So if the developer just blindly updates from `0.1`, then from `0.3`, the second update will result in new conflicts. That might not be so bad if we're only talking about two patches – but if we were talking about `20` or `200`, a lot of needless work would be called for.

Fortunately, `arch` can help. First, the developer gets the latest `0.5` revision:

```
% larch get linux--0.5 ~/wd/linux--0.5
```

Then gets a list of all patches missing from `0.1` and `0.3`:

```
% cd ~/wd/linux--0.5

% larch whats-missing --full linux--0.1 linux--0.3
archive@kernel.org--primary/linux--0.1--versionfix-4
archive@kernel.org--primary/linux--0.3--versionfix-1
```

That list can be piped into the `reconcile` tool:

```
% ... | larch reconcile
archive@kernel.org--primary/linux--0.3--versionfix-1
```

What happened? `reconcile` figured out that `versionfix-1` from `0.3` already includes `versionfix-4` from `0.1` – there's no need to apply both patches. So `patch-plan` reported the list of patches that *do* need to be applied, and in this case, there's only one.

In a more complicated situation, `patch-plan` would print a list of patches in the order they should be applied. In general, it will be a subset of the patches in its input, but applying that subset will have the same effect as applying all of the patches (but hopefully with fewer conflicts).

The developer uses `larch replay --list` to process that list, finally winding up with:

```
    linux--0.1              linux--0.2
    ----------              ----------
    base-0            ----> base-0 (continuation)
    patch-1          |      patch-1
    patch-2          |      version-0
    patch-3          |    ->versionfix-1 (merge)
    patch-4          |   |     |
    version-0  ------    |     |
    versionfix-1\        |     |
    versionfix-2 |-------      |    linux--0.4
  --versionfix-3 |             |    ----------
 |  versionfix-4/               --->base-0 (continuation)
 |            |                  =->patch-1 (replay --exact merge)
 |             -----            |   version-0
 |                  |           .
 |  linux--0.3      |           |
 |  ---------       |           .
  ->base-0 (continuation)       |
    patch-1         |           .
    patch-2         |           |
    patch-3-=--=--=--=--=--=--=-
    ...             |              linux--0.6
  -version-0        V              ----------
 | versionfix-1 (0.1 update)------>base-0 (continuation)
 |                    |            version-0
 |                    |
 |  linux--0.5        |
 |  ----------        |
  ->base-0            |
    patch-1           |
    patch-2           V
    patch-3 (0.1/0.3 reconciliation)
```

Now if someone gets the latest revision of `0.5` and asks:

```
% larch whats-missing --full linux--0.1 linux--0.3
[no output]
```

Isn't `reconcile` handy?

Appendix C The Theory of Patches and Revisions

This appendix briefly explains "patch sets" and "revision control" in the abstract.

C.1 The Theory of Patches

A *patch set* is an expression of the differences between two revisions of a tree of files (usually, primarily, text files). A patch set tells you what files and directories have been added or removed between the two revisions, what files have been renamed, what files have changed. For files added or removed, a patch set tells you the complete contents of those files. For files modified, a patch set contains a description of the changes in the form of a *context diff* (see the man page for `diff(1)`). If a file is a symbolic link, and the link target has changed, the patch set records that fact. If a regular file or directory is replaced by a symbolic link (or vice versa) the patch records that fact. Finally, if any files have had their permissions or modification times changed – a patch set records that too.

Some notation will be helpful. A shorter name for "patch set" is *delta*. Let's suppose that `A` and `B` are two revisions of a source tree. Then:

```
delta (A, B)
```

is the name for a patch set describing the differences between `A` and `B`.

Anytime you make a series of changes to a tree, perhaps using shell utilities and text editors, the entire series of changes can be summarized as a single patch set. In a sense, a patch set (or "delta") is *the fundamental editting operation*, in terms of which all others, and all combinations of others, can be described.

You can *apply a patch* – which means to make the changes it describes. In our notation:

```
delta (A, B) [A] == B
```

says "the patch set describing the differences between A and B, when applied to A, gives B". A patch set can also be applied "in reverse":

```
delta (A, B) {B} == A
```

"the delta from A to B, applied in reverse to B, gives A".

A patch set can also be applied (or reverse-applied) to a tree which is not the same as either `A` or `B`. For example, suppose that we have a tree `A_prime` which is similar to `A`, but has some slight differences. Then:

```
delta (A, B) [A_prime] ~== B_prime
```

where `~==` means "approximately equals". When a patch set is applied to a tree which is not one of the trees used to compute the patch set, the edits might or might not go well. For example, if the patch set wants to modify a file `F`, but `A_prime` doesn't contain `F`, the patch can't be applied perfectly. Similarly, if the patch set wants to modify `F`, but the version of `F` in `A_prime` is already very different from the version in `A`, then those edits can't be done automatically.

Nevertheless, for the kinds of changes people typically make to trees of source code, approximately applied patch sets are very useful. For example, suppose we start with tree A, and create two revisions `B_one` and `B_two`. Then:

```
delta (A, B_one) [B_two] ~== delta (A, B_two) [B_one]
```

To make that more concrete: if programmer Alice makes a set of changes to give us `B_one`, and programmer Bob makes a set of changes to give us `B_two`, then the patch sets between A and those two revisions of `B` give us a way to get `B_three` – a tree that contains *both* Alice's and Bob's changes. Even when a patch set can't be perfectly applied in that way, it can often be applied to do "80%" of the work, making it much easier to finish merging the two sets of changes by hand.

Incidently, patch sets have a useful algebraic property if we think of them as functions that can be composed. Using the notation `F o G` to mean "the function F composed with the function G":

```
For all trees, A, B, and C:

delta (B, C) o delta (A, B) == delta (A, C)
```

so to build `C` from `A`, we can use:

```
delta (A, C) [ A ] == C
```

but that is the same as:

```
(delta (B, C) o delta (A, B)) [ A ] == C
```

or, in other words:

```
delta (B, C) [ delta (A, B) [ A ]] == C
```

The algebraic property suggests that if I want to apply:

```
delta (B, C) o delta (A, B)
```

I can save time by instead applying:

```
delta (A, C)
```

and if I want to have a record of all three:

```
delta (A, B)
delta (B, C)
delta (A, C)
```

I can save space by storing only:

```
delta (A, B)
delta (B, C)
```

And if we're applying patch sets to trees that might need to be touched up by hand, and I want to apply `delta (A, C)`, then I have a choice between applying:

```
delta (A, C)
```

and have just one, possibly large set of errors to clean-up by hand, or applying:

```
delta (A, B) then delta (B, C)
```

and having two, but possibly smaller sets of errors to clean-up by hand.

Patch sets have many uses, but three important ones are:

Compression One use is *compression*. If `A` and `B` are large trees, and the differences between them small, then the patch set between them will be much smaller than either tree. You can save disk space by not storing both `A` and `B`, but instead, storing only `A` (or only `B`) along with `delta (A, B)`. That makes patch sets an ideal form for space-efficient archival of multiple revisions of a tree.

Similarly, if someone has downloaded a copy of `A` (or `B`) and they want a copy of `B` (or `A`), they can save download time and bandwidth by downloading only `delta (A, B)`. That makes patch sets a good way to distribute multiple revisions of a tree.

Inspection Another use for patch sets is *inspection*. If a patch set is stored in a human-readable format, it provides a useful way to quickly see precisely what has changed between two revisions of a tree. For example, a patch set is handy for reviewing the changes made by programmers to a large source tree.

Combining Separate Efforts For some kinds of trees, patch sets are good at merging (combining) changes made by people working separately (as in the example of Alice and Bob, above). This is especially true of program source code. That makes patch sets a very handy tool for making a team of programmers more effective – allowing the work separately up to a point, then combine their efforts by creating and applying patch sets.

C.2 The Theory of Revisions

Suppose that we start with a tree, `A0`, and make a set of changes resulting in the tree `A1`:

```
A0
A1
```

We can repeat that process several times:

```
A0
A1
A2
A3
...
```

Each instance of the tree is called a *revision*. Between each revision and its successor, we can compute a patch set:

```
delta (A0,A1) [A0] == A1
delta (A1,A2) [A1] == A2
delta (A2,A3) [A2] == A3
...
```

Something we can usefully do is create an *archive* of revisions. We might store the first tree verbatim, and every successive tree as a delta:

```
A0: "complete copy of tree"
A1: delta (A0,A1)
A2: delta (A1,A2)
A3: delta (A2,A3)
...
```

If we want to retrieve an `An`, we start with `A0` and apply the first `n` deltas:

```
delta (An-1, An) [delta (An-2, An-1) [....[A0]...]] = An
```

or, making our notation more concise:

```
An [ An-1 [ An-2 [ ... [A0] ...]]] = An
```

Each revision in a series like this is called a *patch level*. The entire series is called a *development path*.

At any point along the way, we might make a copy of some `An`, and start a new development path. For example, we might copy `A2` to form `B0`:

```
A0
A1
A2 ----------------> B0
A3                   B1
...                  B2
                     ...
```

When we have multiple, related development paths, each is called a *branch*. The tree we copied to start a new branch (e.g. `A2`) is called a *branch point*.

If we're building an archive, we can store `B0` as a pointer to the `A0` development path, and every successive revision of the `B0` path as an ordinary delta:

```
A0                      B0: "equal to A2"
A1: delta (A0,A1)       B1: delta (B0, B1)
A2: delta (A1,A2)       B2: delta (B1, B2)
A3: delta (A2,A3)       B3: delta (B2, B3)
...                     ...
```

To make all this more concrete, imagine that the `A0` development path is successive revisions of a program we're working on. Alice wants to add a very complicated feature. Rather than make many small changes to the `A0` development path, she makes a branch, the `B0` development path – and works on the complicated feature there. Each new revision of the `B0` branch is a small step on the way to the complicated feature.

Eventually, Alice is done with the feature – but meanwhile, the `A0` development path has added several changes of its own. What we'd like to do next is to make a revision of the tree that has both sets of changes – from both development paths.

```
A0                          B0: "equal to A2"
A1: delta (A0,A1)           B1: delta (B0, B1)
A2: delta (A1,A2)           B2: delta (B1, B2)
A3: delta (A2,A3)           B3: delta (B2, B3)
A4: delta (A3,A4)           B4: delta (B3, B4)

        How can we make revision A5, which
         includes all the changes made on
                 both branches?
```

The "theory of patches" gives us several possible solutions.

One solution is to make this tree:

```
B4 [ B3 [ B2 [ B1 [ A4 ]]]]
```

That solution is called *replaying* the patches from the `B0` branch against the `A0` branch.

That *might* work reasonably, but patch set `B1` wasn't formed from `A4` – it was formed from `B0` which is the same as `A2`. So when we apply `B1` to `A4`, there might be problems that have to be resolved "by hand". The same will happen again when we apply `B2`, `B3`, and `B4`. In some situations, the risk and complexity of doing all that work by hand is worth it – but not in other situations. What other options do we have?

Another solution is to make this tree:

```
A4 [ A3 [ B4 ] ]
```

That solution is also replaying: replaying the patches from the `A0` branch against the `B0` branch. The same kind of problem might occur (having to fix things up by hand), but in this case, we're only applying two patches instead of four – so this might be a simpler solution.

Here's a third solution:

```
delta (A2, B4) [A4]
```

That solution is based on the fact that:

```
B0 == A2
```

and the algebraic property that:

```
B4 o B3 o B2 o B1 == delta (B0, B4)
```

That solution is called *updating* the `B0` development path with respect to the `A0` path. The difference between "replaying" and "updating" is a little bit subtle. When we "replay" from another development path, that means that we take all patches we're missing from that other path, and apply them in order. When we "update"from another development path, that means we take the latest revision on that other path, and apply to it a delta between the branch point and our own most-recent revision.

Applying the patch during an update certainly can fail to work perfectly – it might require fixing up by-hand. On the other hand, an update only ever applies one patch; often, therefore, the amount of by-hand repairs is minimized. "Nine times out of ten," updating is the preferred technique for joining two previously branched revisions.

There are other, more obscure solutions too. To choose one arbitrarilly, we might try building:

```
B4 [ A3 [ B2 [ A4 [ B1 [ B3 [ A2 ]]]]]]
```

A bizarre solution like that is so rare it doesn't really have name – but "one time in ten thousand" – it's the solution that works best.

No matter what solution we choose, if we store the resulting revision back on the `B0` path, we'll wind up with:

```
A0                        B0: "equal to A2"
A1: delta (A0,A1)         B1: delta (B0, B1)
A2: delta (A1,A2)         B2: delta (B1, B2)
A3: delta (A2,A3)         B3: delta (B2, B3)
A4: delta (A3,A4)         B4: delta (B3, B4)
                          B5: delta (B4, B5) "has changes A3, A4"
```

We can store that same revision back on the `A0` development path:

```
A0                        B0: "equal to A2"
A1: delta (A0,A1)         B1: delta (B0, B1)
A2: delta (A1,A2)         B2: delta (B1, B2)
A3: delta (A2,A3)         B3: delta (B2, B3)
A4: delta (A3,A4)         B4: delta (B3, B4)
A5: delta (A5, A4) ==     B5: delta (B4, B5) "has changes A3, A4"
```

`A5` and `B5` are called a *merge point*. For all practical purposes, a merge point is also a branch point – since, using the example, `B5` and `A5` are equal, just the the two revisions of the original branch point (`A2` and `B0`) were equal. If additional development happens on the two branches, we no longer have to worry about merging all changes since `A2` and `B0`; we can instead just merge only the changes since `A5` and `B5`.

C.3 What is a Revision Control System?

So what is a revision control system?

A revision control system is a set of tools for computing and applying patch sets, for archiving patch sets, for distributing patch sets, and for helping to merge changes on the basis of patch sets.

A revision control system has to come up with a reasonable way of naming and cataloging revisions. It has to be able to represent branch points and help with merges. When merges occur, a good revision control system should help figure out what patches to apply to which revisions in order to minimize hand-editting.

Appendix D The arch Patch Set Format

An arch patch set is a directory containing a number of files and subdirectories. Each is described below.

Files:

```
orig-dirs-index
mod-dirs-index
orig-files-index
mod-files-index
```

Format:

```
<file><tab><tag>
```

Sorting:

```
sort -k 2
```

These contain indexes for all files and directories added, removed, or modified between the two trees.

Files:

```
original-only-dir-metadata
modified-only-dir-metadata
```

Format:

```
<metadata><tab><name>
```

Sorting:

```
sort -t '<tab>' -k 2
```

The field `<metadata>` contains literal output from the program `file-metadata` given the options `--permissions`. Some example output is:

```
--permissions 777
```

That output is also suitable for use as options and option arguments to the program `set-file-metadata`. Future releases `arch` might add additional flags (beside just `permissions`).

The list records the file permissions for all directories persent in only one of the two trees.

Directories:

```
removed-files-archive
new-files-archive
```

Each of these directories contains complete copies of all files that occur in only the original tree (`removed-files-archive`) or modified tree (`new-files-archive`). Each saved file is archived at the same relative location it had in its source tree, with permissions (at least) preserved.

Directory:

```
patches
```

This directory contains a tree whose directory structure is a subset of the directory structure of the modified tree. It contains modification data for directories and files common to both trees.

For a file stored in the modified tree at the path `new_name`, the `patches` directory may contain:

```
new_name.link-orig


        The original file is a symbolic link.
        ‘new_name.link-orig’ is a text file containing the
        target of that link plus a final newline.

        This file is only present if link target has changed,
        or if the link was replaced by a regular file.

new_name.link-mod


        The modified file is a symbolic link and this file
        is a text file containing the target for the link plus
        a final newline.

        This file is only present if the link target has
        changed, or if the link replaces a regular file.

new_name.original


        This is a complete copy of the file from the original
        tree, preserving (at least) permissions.

        This file is only present if the file was replaced by
        a symbolic link, or if the file contents can not be
        handled by ‘diff(1)’.

new_name.modified


        This is a complete copy of the file from the modified
        tree, preserving (at least) permissions.

        This file is only present if the file replaces a
        symbolic link, or if the file contents can not be
        handled by ‘diff(1)’.

new_name.patch
```

```
This is a standard context diff between the original
file and modified file.  One popular version of diff
(`GNU diff') generates non-standard context diffs by
omitting one copy of lines of context that are
identical between the original and modified file, so
for now, `.patch' files may have the same bug.
Fortunately, the only popular version of `patch'
(``GNU patch'') is tolerant of receiving such input.


new_name.meta-orig
new_name.meta-mod


        File metadata (currenly only permissions) changed
        between the two versions of the file.  These files
        contain output from the `file-metadata' program with
        the flags `--symlink --permissions', suitable for
        comparison to similar output, and for use as options
        and option arguments to `set-file-metadata'.


        These files are also included if a regular file has
        replaced a symbolic link or vice versa.


new_name/=dir-meta-orig
new_name/=dir-meta-mod


        Directory metadata (currenly only permissions) changed
        between the two versions of the directory containing
        these files.  These files contain output from the
        `file-metadata' program with the flags `--symlink
        --permissions', suitable for comparison to similar
        output, and for use as options and option arguments to
        `set-file-metadata'.
```

Note: If a regular file (or symbolic link) replaces a directory, or vice versa, this is recorded as a file (or link) removed (or added) in one tree and added (or removed) in the other.

Appendix E The arch Archive Format

An arch archive is a directory containing a number of files and subdirectories. Its structure is described in this appendix.

E.1 Directory Structure

Each category, branch, version, and revision are given a separate directory. These are nested. From the root of the archive:

```
CATEGORY/
  CATEGORY/BRANCH/
    CATEGORY/BRANCH/VERSION/
      CATEGORY/BRANCH/VERSION/REVISION/
```

For example, version `1.0` of the `devo` branch of the `arch` category might have these directories:

```
arch/
  arch/arch--devo/
    arch/arch--devo--1.0/
      arch/arch--devo--1.0/base-0/  # the base revision
      arch/arch--devo--1.0/patch-1/ # pre-patch revisions
      arch/arch--devo--1.0/patch-2/ # ...
      ...
      arch/arch--devo--1.0/version-0/ # the version revision
      arch/arch--devo--1.0/versionfix-1/ # post-patch revisions
      arch/arch--devo--1.0/versionfix-2/ # ...
      ...
```

E.2 Within a Revision Directory

Every revision directory contains the log message for that revision as a plain-text file:

File: log

Format: RFC822-style headers plus body

If a revision directory is a full-source revision (typically a `base-0` revision) it will contain a compressed tar file of the entire revision tree. The name of the tar file is the full name (sans archive name) of the revision:

File: REVISION.tar.gz

Format: gzip-compressed tar file containing a complete source tree, rooted in a single top-level directory named REVISION.

Example: arch–devo–1.0–base-0.tar.gz

```
contains a full-source tree rooted at
'arch--devo--1.0--base-0'
```

If a revision directory is a patch (not a full-source revision), then the revision directory contains a compressed tar file of the patch set (see Appendix D [The arch Patch Set Format], page 133):

File: REVISION.patches.tar.gz

Format: gzip-compressed tar file containing a patch set, rooted in a single top-level directory named REVISION.

Example: arch–devo–1.0–patch-1.tar.gz

```
contains a patch-set tree rooted at ‘arch--devo--1.0--patch-1’
```

Note: Every revision is either a full source revision or a patch revision. Thus, every revision directory contains exactly one of the two files:

```
REVISION.tar.gz
REVISION.patches.tar.gz
```

If a revision is a continuation revision (a tag of some other revision)

Appendix F The GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

```
A "Secondary Section" is a named appendix or a front-matter section of
the Document that deals exclusively with the relationship of the
publishers or authors of the Document to the Document's overall subject
(or to related matters) and contains nothing that could fall directly
within that overall subject.  (For example, if the Document is in part a
textbook of mathematics, a Secondary Section may not explain any
mathematics.)  The relationship could be a matter of historical
connection with the subject or with related matters, or of legal,
commercial, philosophical, ethical or political position regarding
them.

The "Invariant Sections" are certain Secondary Sections whose titles
are designated, as being those of Invariant Sections, in the notice
that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed,
as Front-Cover Texts or Back-Cover Texts, in the notice that says that
the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy,
represented in a format whose specification is available to the
general public, whose contents can be viewed and edited directly and
straightforwardly with generic text editors or (for images composed of
pixels) generic paint programs or (for drawings) some widely available
drawing editor, and that is suitable for input to text formatters or
for automatic translation to a variety of formats suitable for input
to text formatters.  A copy made in an otherwise Transparent file
format whose markup has been designed to thwart or discourage
subsequent modification by readers is not Transparent.  A copy that is
not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain
ASCII without markup, Texinfo input format, LaTeX input format, SGML
or XML using a publicly available DTD, and standard-conforming simple
HTML designed for human modification.  Opaque formats include
PostScript, PDF, proprietary formats that can be read and edited only
by proprietary word processors, SGML or XML for which the DTD and/or
processing tools are not generally available, and the
machine-generated HTML produced by some word processors for output
purposes only.

The "Title Page" means, for a printed book, the title page itself,
plus such following pages as are needed to hold, legibly, the material
this License requires to appear in the title page.  For works in
formats which do not have any title page as such, "Title Page" means
the text near the most prominent appearance of the work's title,
preceding the beginning of the body of the text.
```

distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as

```
   given on its Title Page, then add an item describing the Modified
   Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for
   public access to a Transparent copy of the Document, and likewise
   the network locations given in the Document for previous versions
   it was based on.  These may be placed in the "History" section.
   You may omit a network location for a work that was published at
   least four years before the Document itself, or if the original
   publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications",
   preserve the section's title, and preserve in the section all the
   substance and tone of each of the contributor acknowledgements
   and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document,
   unaltered in their text and in their titles.  Section numbers
   or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements".  Such a section
   may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements"
   or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or
appendices that qualify as Secondary Sections and contain no material
copied from the Document, you may at your option designate some or all
of these sections as invariant.  To do this, add their titles to the
list of Invariant Sections in the Modified Version's license notice.
These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains
nothing but endorsements of your Modified Version by various
parties--for example, statements of peer review or that the text has
been approved by an organization as the authoritative definition of a
standard.

You may add a passage of up to five words as a Front-Cover Text, and a
passage of up to 25 words as a Back-Cover Text, to the end of the list
of Cover Texts in the Modified Version.  Only one passage of
Front-Cover Text and one of Back-Cover Text may be added by (or
through arrangements made by) any one entity.  If the Document already
includes a cover text for the same cover, previously added by you or
by arrangement made by the same entity you are acting on behalf of,
you may not add another; but you may replace the old one, on explicit
permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License
give permission to use their names for publicity for or to assert or
imply endorsement of any Modified Version.
```

```
5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this
License, under the terms defined in section 4 above for modified
versions, provided that you include in the combination all of the
Invariant Sections of all of the original documents, unmodified, and
list them all as Invariant Sections of your combined work in its
license notice.

The combined work need only contain one copy of this License, and
multiple identical Invariant Sections may be replaced with a single
copy.  If there are multiple Invariant Sections with the same name but
different contents, make the title of each such section unique by
adding at the end of it, in parentheses, the name of the original
author or publisher of that section if known, or else a unique number.
Make the same adjustment to the section titles in the list of
Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History"
in the various original documents, forming one section entitled
"History"; likewise combine any sections entitled "Acknowledgements",
and any sections entitled "Dedications".  You must delete all sections
entitled "Endorsements."


6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents
released under this License, and replace the individual copies of this
License in the various documents with a single copy that is included in
the collection, provided that you follow the rules of this License for
verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute
it individually under this License, provided you insert a copy of this
License into the extracted document, and follow this License in all
other respects regarding verbatim copying of that document.


7. AGGREGATION WITH INDEPENDENT WORKS
```

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

```
Each version of the License is given a distinguishing version number.
If the Document specifies that a particular numbered version of this
License "or any later version" applies to it, you have the option of
following the terms and conditions either of that specified version or
of any later version that has been published (not as a draft) by the
Free Software Foundation.  If the Document does not specify a version
number of this License, you may choose any version ever published (not
as a draft) by the Free Software Foundation.


ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of
the License in the document and put the following copyright and
license notices just after the title page:

      Copyright (c)  YEAR  YOUR NAME.
      Permission is granted to copy, distribute and/or modify this
      document under the terms of the GNU Free Documentation License,
      Version 1.1 or any later version published by the Free Software
      Foundation; with the Invariant Sections being LIST THEIR TITLES,
      with the Front-Cover Texts being LIST, and with the Back-Cover
      Texts being LIST.  A copy of the license is included in the
      section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections"
instead of saying which ones are invariant.  If you have no
Front-Cover Texts, write "no Front-Cover Texts" instead of
"Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we
recommend releasing these examples in parallel under your choice of
free software license, such as the GNU General Public License,
to permit their use in free software.
```

Appendix G The GNU General Public License

```
The Hackerlab C Library is free software: you can redistribute it
and/or modify it under the terms of the GNU General Public License as
published by the Free Software Foundation (and reproduced below).

This software is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License (reproduced below) for more details.


                    GNU GENERAL PUBLIC LICENSE
                       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
     59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change
free software--to make sure the software is free for all its users.
This General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit
to using it.  (Some other Free Software Foundation software is
covered by the GNU Library General Public License instead.)  You can
apply it to your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that
you have the freedom to distribute copies of free software (and
charge for this service if you wish), that you receive source code
or can get it if you want it, that you can change the software or
use pieces of it in new free programs; and that you know you can do
these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the
rights.  These restrictions translate to certain responsibilities
for you if you distribute copies of the software, or if you modify
it.
```

```
  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights
that you have.  You must make sure that they, too, receive or can
get the source code.  And you must show them these terms so they
know their rights.

  We protect your rights with two steps: (1) copyright the software,
and (2) offer you this license which gives you legal permission to
copy, distribute and/or modify the software.

  Also, for each author's protection and ours, we want to make
certain that everyone understands that there is no warranty for this
free software.  If the software is modified by someone else and
passed on, we want its recipients to know that what they have is not
the original, so that any problems introduced by others will not
reflect on the original authors' reputations.

  Finally, any free program is threatened constantly by software
patents.  We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making
the program proprietary.  To prevent this, we have made it clear
that any patent must be licensed for everyone's free use or not
licensed at all.

  The precise terms and conditions for copying, distribution and
modification follow.


                    GNU GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License applies to any program or other work which
contains a notice placed by the copyright holder saying it may be
distributed under the terms of this General Public License.  The
"Program", below, refers to any such program or work, and a "work
based on the Program" means either the Program or any derivative
work under copyright law: that is to say, a work containing the
Program or a portion of it, either verbatim or with modifications
and/or translated into another language.  (Hereinafter, translation
is included without limitation in the term "modification".)  Each
licensee is addressed as "you".

Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running the Program is not restricted, and the output from the
Program is covered only if its contents constitute a work based on
the Program (independent of having been made by running the
Program).  Whether that is true depends on what the Program does.
```

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all.

For example, if a patent license would not permit royalty-free
redistribution of the Program by all those who receive copies
directly or indirectly through you, then the only way you could
satisfy both it and this License would be to refrain entirely from
distribution of the Program.

If any portion of this section is held invalid or unenforceable
under any particular circumstance, the balance of the section is
intended to apply and the section as a whole is intended to apply in
other circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices. Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is
willing to distribute software through any other system and a
licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed
to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces,
the original copyright holder who places the Program under this
License may add an explicit geographical distribution limitation
excluding those countries, so that distribution is permitted only in
or among countries not thus excluded. In such case, this License
incorporates the limitation as if written in the body of this
License.

9. The Free Software Foundation may publish revised and/or new
versions of the General Public License from time to time. Such new
versions will be similar in spirit to the present version, but may
differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the
Program specifies a version number of this License which applies to
it and "any later version", you have the option of following the
terms and conditions either of that version or of any later version
published by the Free Software Foundation. If the Program does not
specify a version number of this License, you may choose any version
ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it
does.>
Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License as
published by the Free Software Foundation; either version 2 of
the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public
License along with this program; if not, write to the Free
Software Foundation, Inc., 59 Temple Place, Suite 330, Boston,
MA 02111-1307 USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type
'show w'.  This is free software, and you are welcome to
redistribute it under certain conditions; type 'show c' for
details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
program 'Gnomovision' (which makes passes at compilers) written by
James Hacker.
```

```
  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice

This General Public License does not permit incorporating your
program into proprietary programs.  If your program is a subroutine
library, you may consider it more useful to permit linking
proprietary applications with the library.  If this is what you want
to do, use the GNU Library General Public License instead of this
License.
```

Appendix H Preliminary Data Sheet for the Hackerlab arch

```
Package:
        Hackerlab arch (a revision control system)

Supplier:
        regexps.com

Function:

        Hackerlab arch performs revision control offering fancy
        features for branching and merging.  It provides distributed
        repositories (repositories spread over multiple hosts) and a
        global (world-wide) namespace for branches and revisions.


Key Features:

        arch is simple, small, fast, and featureful.

        Distributed repositories are especially appropriate for
        projects developed in the "open source" style -- with
        geographically distributed developers and sub-teams, loosely
        and flexibly cooperating.

        Arch's fancy merging features are ideal for projects
        supporting multiple concurrent releases and/or performing a
        lot of development in separate branches.

        Postscript and HTML documentation is included.


Licensing:

        Hackerlab arch is part of is distributed under the terms of
        the GNU General Public License, Version 2, as published by the
        Free Software Foundation.


Prerequisites:

        standard C compiler
        Posix libc.a (standard C library)
        GNU Make
        GNU tar
        GNU patch
        most of the standard Posix shell, text, and file utilities
```

```
Recommended and Disrecommended Applications:

        arch is brand new.  The recommended application at this stage
        is evaluation, help with porting, and help with testing.

        Given resources for robust testing, arch could be made ready
        for heavy-duty, mission critical application within months --
        but absent those resources, arch suffers from the risks
        associated with brand-new software.

        If your projects are characterized by distributed development,
        consider evaluating arch and finding ways to contribute to a
        polished, commercial-quality release.


Limitations:

        The primary limitation is that arch needs porting and testing.

        A secondary limitation is that arch likes to operate on
        whole-trees, not individual files.  This limitation can be
        overcome with further development, though it remains to be
        seen if user's truly miss such features.


Size:

        The core of arch is around 40K lines of code, mostly shell and
        awk scripts.

        Small parts of arch rely on the Hackerlab C library, which
        adds considerably to the overall code size.  It is possible to
        eliminate this dependency, but not necessarily desirable.


Performance:

        Arch has some nice performance characteristics.

        Repositories are stored in compressed format and tree-deltas
        are efficiently represented.  This both saves disk space and
        reduces network traffic.

        On the client side, arch makes heavy use of caching to speed
        up some operations and make detached operation possible.
```

```
Support:

        To purchase an alternative license, request additional
        features, or for any kind of support assistance, you can
        contact us at "hackerlab@regexps.com" or via our web site
        "www.regexps.com".  We can also be reached at (412) 401-5204.
```

Indexes

J

L

M

N

P

R

S

T

U

V

W

Short Contents

Table of Contents