Darcs 1.0.5 (release)

Darcs

David Roundy

December 7, 2005

Contents

Chapter 1

Introduction

Darcs is a revision control system, along the lines of CVS or arch. That means that it keeps track of various revisions and branches of your project, allows for changes to propagate from one branch to another. Darcs is intended to be an "advanced" revision control system. Darcs has two particularly distinctive features which differ from other revision control systems: 1) each copy of the source is a fully functional branch, and 2) underlying darcs is a consistent and powerful theory of patches.

Every source tree a branch The primary simplifying notion of darcs is that *every* copy of your source code is a full repository. This is dramatically different from CVS, in which the normal usage is for there to be one central repository from which source code will be checked out. It is closer to the notion of arch, since the 'normal' use of arch is for each developer to create his own repository. However, darcs makes it even easier, since simply checking out the code is all it takes to create a new repository. This has several advantages, since you can harness the full power of darcs in any scratch copy of your code, without committing your possibly destabilizing changes to a central repository.

Theory of patches The development of a simplified theory of patches is what originally motivated me to create darcs. This patch formalism means that darcs patches have a set of properties, which make possible manipulations that couldn't be done in other revision control systems. First, every patch is invertible. Secondly, sequential patches (i.e. patches that are created in sequence, one after the other) can be reordered, although this reordering can fail, which means the second patch is dependent on the first. Thirdly, patches which are in parallel (i.e. both patches were created by modifying identical trees) can be merged, and the result of a set of merges is independent of the order in which the merges are performed. This last property is critical to darcs' philosophy, as it means that a particular version of a source tree is fully defined by the list of patches that are in it, i.e. there is no issue regarding the order in which merges are performed. For a more thorough discussion of darcs' theory of patches, see Appendix A.

A simple advanced tool Besides being "advanced" as discussed above, darcs is actually also quite simple. Versioning tools can be seen as three layers. At the foundation is the ability to manipulate changes. On top of that must be placed some kind of database system to keep track of the changes. Finally, at the very top is some sort of distribution system for getting changes from one place to another.

Really, only the first of these three layers is of particular interest to me, so the other two are done as simply as possible. At the database layer, darcs just has an ordered list of patches along with the patches themselves, each stored as an individual file. Darcs' distribution system is strongly inspired by that of arch. Like arch, darcs uses a dumb server, typically apache or just a local or network file system when pulling patches. darcs has built-in support for using `ssh` to write to a remote file system. A darcs executable is called on the remote system to apply the patches. Arbitrary other transport protocols are supported, through an environment variable describing a command that will run darcs on the remote system. See the documentation for DARCS_APPLY_FOO in Chapter 4 for details.

The recommended method is to send patches through gpg-signed email messages, which has the advantage of being mostly asynchronous.

Keeping track of changes rather than versions In the last paragraph, I explained revision control systems in terms of three layers. One can also look at them as having two distinct uses. One is to provide a history of previous versions. The other is to keep track of changes that are made to the repository, and to allow these changes to be merged and moved from one repository to another. These two uses are distinct, and almost orthogonal, in the sense that a tool can support one of the two uses optimally while providing no support for the other. Darcs is not intended to maintain a history of versions, although it is possible to kludge together such a revision history, either by making each new patch depend on all previous patches, or by tagging regularly. In a sense, this is what the tag feature is for, but the intention is that tagging will be used only to mark particularly notable versions (e.g. released versions, or perhaps versions that pass a time consuming test suite).

Other revision control systems are centered upon the job of keeping track of a history of versions, with the ability to merge changes being added as it was seen that this would be desirable. But the fundamental object remained the versions themselves.

In such a system, a patch (I am using patch here to mean an encapsulated set of changes) is uniquely determined by two trees. Merging changes that are in two trees consists of finding a common parent tree, computing the diffs of each tree with their parent, and then cleverly combining those two diffs and applying the combined diff to the parent tree, possibly at some point in the process allowing human intervention, to allow for fixing up problems in the merge such as conflicts.

In the world of darcs, the source tree is *not* the fundamental object, but rather the patch is the fundamental object. Rather than a patch being defined in terms of the difference between two trees, a tree is defined as the result of applying a given set of patches to an empty tree. Moreover, these patches may be reordered (unless there are dependencies between the patches involved) without changing the tree. As a result, there is no need to find a common parent when performing a merge. Or, if you like, their common parent is defined by the set of common patches, and may not correspond to any version in the version history.

One useful consequence of darcs' patch-oriented philosophy is that since a patch need not be uniquely defined by a pair of trees (old and new), we can have several ways of representing the same change, which differ only in how they commute and what the result of merging them is. Of course, creating such a patch will require some sort of user input. This is a Good Thing, since the user *creating* the patch should be the one forced to think about what he really wants to change, rather than the users merging the patch. An example of this is the token replace patch (See Section A.3.2). This feature makes it possible to create a patch, for example, which changes every instance of the variable "stupidly_named_var" to "better_var_name", while leaving "other_stupidly_named_var" untouched. When this patch is merged with any other patch involving the "stupidly_named_var", that instance will also be modified to "better_var_name". This is in contrast to a more conventional merging method which would not only fail to change new instances of the variable, but would also involve conflicts when merging with any patch that modifies lines containing the variable. By more using additional information about the programmer's intent, darcs is thus able to make the process of changing a variable name the trivial task that it really is, which is really just a trivial search and replace, modulo tokenizing the code appropriately.

The patch formalism discussed in Appendix A is what makes darcs' approach possible. In order for a tree to consist of a set of patches, there must be a deterministic merge of any set of patches, regardless of the order in which they must be merged. This requires that one be able to reorder patches. While I don't know that the patches are required to be invertible as well, my implementation certainly requires invertibility. In particular, invertibility is required to make use of Theorem 2, which is used extensively in the manipulation of merges.

1.1 Features

Record changes locally In darcs, the equivalent of a cvs "commit" is called record, because it doesn't put the change into any remote or centralized repository. Changes are always recorded locally, meaning no net access is required in order to work on your project and record changes as you make them. Moreover, this means that there is no need for a separate "disconnected operation" mode.

Interactive records You can choose to perform an interactive record, in which case darcs will prompt you for each change you have made and ask if you wish to record it. Of course, you can tell darcs to record all the changes in a given file, or to skip all the changes in a given file, or go back to a previous change, or whatever. There is also an experimental graphical interface, which allows you to view and choose changes even more easily, and in whichever order you like.

Unrecord local changes As a corollary to the "local" nature of the record operation, if a change hasn't yet been published to the world—that is, if the local repository isn't accessible by others—you can safely unrecord a change (even if it wasn't the most recently recorded change) and then re-record it differently, for example if you forgot to add a file, introduced a bug or realized that what you recorded as a single change was really two separate changes.

Interactive everything else Most darcs commands support an interactive interface. The "revert" command, for example, which undoes unrecorded changes has the same interface as record, so you can easily revert just a single change. Pull, push, send and apply all allow you to view and interactively select which changes you wish to pull, push, send or apply.

Test suites Darcs has support for integrating a test suite with a repository. If you choose to use this, you can define a test command (e.g. "make check") and have darcs run that command on a clean copy of the project either prior to recording a change or prior to applying changes—and to reject changes that cause the test to fail.

Any old server Darcs does not require a specialized server in order to make a repository available for read access. You can use http, ftp, or even just a plain old ssh server to access your darcs repository.

You decide write permissions Darcs doesn't try to manage write access. That's your business. Supported push methods include direct ssh access (if you're willing to *give* direct ssh access away), using sudo to allow users who already have shell access to only apply changes to the repository, or verification of gpg-signed changes sent by email against a list of allowed keys. In addition, there is good support for submission of patches by email that are not automatically applied, but can easily be applied with a shell escape from a mail reader (this is how I deal with contributions to darcs).

Symmetric repositories Every darcs repository is created equal (well, with the exception of a "partial" repository, which doesn't contain a full history. . .), and every working directory has an associated repository. As a result, there is a symmetry between "uploading" and "downloading" changes—you can use the same commands (push or pull) for either purpose.

CGI script Darcs has a CGI script that allows browsing of the repositories.

Portable Darcs runs on UNIX (or UNIX-like) systems (which includes Mac OS X) as well as on Microsoft Windows.

File and directory moves Renames or moves of files and directories, of course are handled properly, so when you rename a file or move it to a different directory, its history is unbroken, and merges with repositories that don't have the file renamed will work as expected.

Token replace You can use the "darcs replace" command to modify all occurrences of a particular token (defined by a configurable set of characters that are allowed in "tokens") in a file. This has the advantage that merges with changes that introduce new copies of the old token will have the effect of changing it to the new token—which comes in handy when changing a variable or function name that is used throughout a project.

Configurable defaults You can easily configure the default flags passed to any command on either a per-repository or a per-user basis or a combination thereof.

1.2 Switching from CVS

Darcs is refreshingly different from CVS.

CVS keeps version controlled data in a central repository, and requires that users check out a working directory whenever they wish to access the version-controlled sources. In order to modify the central repository, a user needs to have write access to the central repository; if he doesn't, CVS merely becomes a tool to get the latest sources.

In darcs there is no distinction between working directories and repositories. In order to work on a project, a user makes a local copy of the repository he wants to work in; he may then harness the full power of version control locally. In order to distribute his changes, a user who has write access can *push* them to the remote repository; one who doesn't can simply send them by e-mail in a format that makes them easy to apply on the remote system.

Darcs commands for CVS users Because of the different models used by cvs and darcs, it is difficult to provide a complete equivalence between cvs and darcs. A rough correspondence for the everyday commands follows:

```
cvs checkout
     darcs get
cvs update
     darcs pull
```

`cvs -n update`
: `darcs pull --dry-run` (summarize remote changes)

`cvs -n update`
: `darcs whatsnew --summary` (summarize local changes)

`cvs -n update | grep '?'`
: `darcs whatsnew -ls | grep ^a` (list potential files to add)

`rm foo.txt; cvs update foo.txt`
: `darcs revert foo.txt` (revert to foo.txt from repo)

`cvs diff`
: `darcs whatsnew` (if checking local changes)

`cvs diff`
: `darcs diff` (if checking recorded changes)

`cvs commit`
: `darcs record` (if committing locally)

`cvs commit`
: `darcs tag` (if marking a version for later use)

`cvs commit`
: `darcs push` or `darcs send` (if committing remotely)

`cvs diff | mail`
: `darcs send`

`cvs add`
: `darcs add`

`cvs tag -b`
: `darcs get`

`cvs tag`
: `darcs tag`

Migrating CVS repositories to darcs Tools and instructions for migrating CVS repositories to darcs are provided on the darcs community website: http://darcs.net/DarcsWiki/ConvertingFromCvs

1.3 Switching from arch

Although arch, like darcs, is a distributed system, and the two systems have many similarities (both require no special server, for example), their essential organization is very different.

Like CVS, arch keeps data in two types of data structures: repositories (called "archives") and working directories. In order to modify a repository, one must first check out a corresponding working directory. This requires that users remember a number of different ways of pushing data around — `tla get`, `update`, `commit`, `archive-mirror` and so on.

[0] `http://darcs.net/DarcsWiki/ConvertingFromCvs`

In darcs, on the other hand, there is no distinction between working directories and repositories, and just checking out your sources creates a local copy of a repository. This allows you to harness the full power of version control in any scratch copy of your sources, and also means that there are just two ways to push data around: `darcs record`, which stores edits into your local repository, and `pull`, which moves data between repositories. (`darcs push` is merely the opposite of `pull`; `send` and `apply` are just the two halves of `push`).

Darcs commands for arch users Because of the different models used by arch and darcs, it is difficult to provide a complete equivalence between arch and darcs. A rough correspondence for the everyday commands follows:

`tla init-tree`
: `darcs initialize`

`tla get`
: `darcs get`

`tla update`
: `darcs pull`

`tla file-diffs f | patch -R`
: `darcs revert`

`tla changes --diffs`
: `darcs whatsnew`

`tla logs`
: `darcs changes`

`tla file-diffs`
: `darcs diff -u`

`tla add`
: `darcs add`

`tla mv`
: `darcs mv` (not `tla move`)

`tla commit`
: `darcs record` (if committing locally)

`tla commit`
: `darcs tag` (if marking a version for later use)

`tla commit`
: `darcs push` or `darcs send` (if committing remotely)

`tla archive-mirror`
: `darcs pull` or `darcs push`

`tla tag`
: `darcs get` (if creating a branch)

`tla tag`
: `darcs tag` (if creating a tag).

Migrating arch repositories to darcs Tools and instructions for migrating arch repositories to darcs are provided on the darcs community website: http://darcs.net/DarcsWiki/ConvertingFromArch

[0]`http://darcs.net/DarcsWiki/ConvertingFromArch`

Chapter 2

Building darcs

This chapter should walk you through the steps necessary to build darcs for yourself. There are in general two ways to build darcs. One is for building released versions from tarballs, and the other is to build the latest and greatest darcs, from the darcs repo itself.

Please let me know if you have any problems building darcs, or don't have problems described in this chapter and think there's something obsolete here, so I can keep this page up-to-date.

2.1 Prerequisites

To build darcs you will need to have `ghc`, the Glorious Glasgow Haskell Compiler. You should have at the very minimum version 6.2.

It is a good idea (but not required) to have a recent version of libcurl installed. If not, you will at least need to have either `wget` or `curl` installed if you want to be able to grab repos remotely over normal network protocols (ftp or http). You also might want to have scp available if you want to grab your repos over ssh…

To send patches, you will also need to have a working `/usr/sbin/sendmail` or `/usr/lib/sendmail`, which is provided by most mail transport agents, and is generally available on linux and BSD systems. It's also there on Mac OS X. However, if you don't have this, it won't stop you from building darcs.

To use the `diff` command of darcs, a `diff` program supporting options `-r` (recursive diff) and `-N` (show new files as differences against an empty file) is required. The `configure` script will look for `gdiff`, `gnudiff` and `diff` in this order. You can force the use of another program by setting the `DIFF` environment variable before running `configure`.

To rebuild the documentation (which should not be necessary since it is included in html form with the tarballs), you will need to have latex installed, as well as latex2html if you want to build it in html form.

2.2 Building on Mac OS X

To build on Mac OS X, you will need the Apple Developer Tools and the ghc 6.4 package installed.

2.3 Building on Microsoft Windows

To build on Microsoft Windows, you will need:

- MinGW which provides the GCC toolchain for win32.
- MSYS which provides a unix build environment for win32. Be sure to download the separate msysDTK, autoconf and automake.
- zlib-1.2.1+ library and headers.
- curl-7.12.2+ library and headers.
- If building with an SSL enabled curl you will need the OpenSSL libraries, unofficial builds are available at http://www.slproweb.com/products/Win32OpenSSL.html.

Copy the zlib and curl libraries and headers to both GHC and MinGW. GHC stores C headers in `<ghc-dir>/gcc-lib/include` and libraries in `<ghc-dir>/gcc-lib`. MinGW stores headers in `<mingw-dir>/include` and libraries in `<mingw-dir>/lib`.

Set PATH to include the `<msys-dir>/bin`, `<mingw-dir>/bin`, `<curl-dir>`, and a directory containing a pre-built darcs.exe if you want the build's patch context stored for '`darcs --exact-version`'.

```
C:\darcs> cd <darcs-source-dir>
C:\darcs> sh

$ export GHC=/c/<ghc-dir>/bin/ghc.exe
$ autoconf
$ ./configure --disable-mmap --target=mingw
$ make
```

2.4 Building from tarball

If you get darcs from a tarball, the procedure (after unpacking the tarball itself) is as follows:

[0] `http://www.mingw.org/`
[0] `http://www.mingw.org/msys.shtml`
[0] `http://www.gzip.org/zlib/`
[0] `http://curl.haxx.se/`

```
% ./configure
% make
# Optional, but recommended to test compatibility with your environment.
% make test
% make install
```

There are options to configure that you may want to check out with

```
% ./configure --help
```

If your header files are installed in a non-standard location, you may need to define the `CFLAGS` and `CPPFLAGS` environment variables to include the path to the headers. e.g. on NetBSD, you may need to run

```
% CFLAGS=-I/usr/pkg/include CPPFLAGS=-I/usr/pkg/include ./configure
```

2.5 Building darcs from the repository

To build the latest darcs from its repository, you will first need a working copy of darcs. You can get darcs using:

```
% darcs get -v http://abridgegame.org/repos/darcs
```

and once you have the darcs repository you can bring it up to date with a

```
% darcs pull
```

The repository doesn't hold automatically generated files, which include the configure script and the HTML documentation, so you need to run `autoconf` first.

You'll need `autoconf` 2.50 or higher. Some systems have more than one version of `autoconf` installed. For example, `autoconf` may point to version 2.13, while `autoconf259` runs version 2.59.

Also note that `make` is really "GNU make". On some systems, such as the *BSDs, you may need to type `gmake` instead of make for this to work.

If you want to create readable documentation you'll need to have latex installed.

```
% autoconf
% ./configure
% make
% make install
```

If you want to tweak the configure options, you'll need to run `./configure` yourself after the make, and then run make again.

2.6 Building darcs with git

To enable git support, you first need to grab a copy of the git source code; since darcs doesn't yet have the capability of accessing remote git repositories, you'll have to either download a tarball or use git itself to clone a git repository. Compile git (no need to install); this will create a file "`libgit.a`". Then create a symlink to the git source directory named "`git`" in your darcs source directory, configure darcs using the "`--enable-git`" option, and build darcs as usual.

2.7 Submitting patches to darcs

I know, this doesn't really belong in this chapter, but if you're using the repository version of darcs it's really easy to submit patches to me using darcs. In fact, even if you don't know any Haskell, you could submit fixes or additions to this document (by editing `building_darcs.tex`) based on your experience building darcs...

To do so, just record your changes (which you made in the darcs repository)

```
% darcs record --no-test
```

making sure to give the patch a nice descriptive name. The `--no-test` options keeps darcs from trying to run the unit tests, which can be rather time-consuming. Then you can send the patch to the darcs-devel mailing list by email by

```
% darcs send -u
```

The darcs repository stores the email address to which patches should be sent by default. The email address you see is actually my own, but when darcs notices that you haven't signed the patch with my GPG key, it will forward the message to darcs-devel.

Chapter 3

Getting started

This chapter will lead you through an example use of darcs, which hopefully will allow you to get started using darcs with your project.

3.1 Creating your repository

Creating your repository in the first place just involves telling darcs to create the special directory (called `_darcs`) in your project tree, which will hold the revision information. This is done by simply calling from the root directory of your project:

```
% cd my_project/
% darcs initialize
```

This creates the `_darcs` directory and populates it with whatever files and directories are needed to describe an empty project. You now need to tell darcs what files and directories in your project should be under revision control. You do this using the command `darcs add`[1]:

```
% darcs add *.c Makefile.am configure.ac
```

When you have added all your files (or at least, think you have), you will want to record your changes. "Recording" always includes adding a note as to why the change was made, or what it does. In this case, we'll just note that this is the initial version.

```
% darcs record --all
What is the patch name? Initial revision.
```

Note that since we didn't specify a patch name on the command line we were prompted for one. If the environment variable 'EMAIL' isn't set, you will also

[1]Note that darcs does not do wildcard expansion, instead relying on the command shell. The Windows port of darcs has a limited form of expansion provided by the C runtime

be prompted for your email address. Each patch that is recorded is given a unique identifier consisting of the patch name, its creator's email address, and the date when it was created.

3.2 Making changes

Now that we have created our repository, make a change to one or more of your files. After making the modification run:

```
% darcs whatsnew
```

This should show you the modifications that you just made, in the darcs patch format. If you prefer to see your changes in a different format, read Section 6.6.1, which describes the whatsnew command in detail.

Let's say you have now made a change to your project. The next thing to do is to record a patch. Recording a patch consists of grouping together a set of related changes, and giving them a name. It also tags the patch with the date it was recorded and your email address.

To record a patch simply type:

```
% darcs record
```

darcs will then prompt you with all the changes that you have made that have not yet been recorded, asking you which ones you want to include in the new patch. Finally, darcs will ask you for a name for the patch.

You can now rerun whatsnew, and see that indeed the changes you have recorded are no longer marked as new.

3.3 Making your repository visible to others

How do you let the world know about these wonderful changes? Obviously, they must be able to see your repository. Currently the easiest way to do this is typically by http using any web server. The recommended way to do this (using apache in a UNIX environment) is to create a directory called `/var/www/repos`, and then put a symlink to your repo there:

```
% cd /var/www/repos
% ln -s /home/username/myproject .
```

As long as you're running a web server and making your repo available to the world, you may as well make it easy for people to see what changes you've made. You can do this by running `make installserver`, which installs the program `darcs_cgi` at `/usr/lib/cgi-bin/darcs`. You also will need to create a cache directory named `/var/cache/darcs_cgi`, and make sure the owner of that directory is the same user that your web server runs its cgi scripts as. For me, this is www-data. Now your friends and enemies should be able to easily browse your repos by pointing their web browsers at `http://your.server.org/cgi-bin/darcs`.

3.4 Getting changes made to another repository

Ok, so I can now browse your repository using my web browser... so what? How do I get your changes into *my* repository, where they can do some good? It couldn't be easier. I just `cd` into my repository, and there type:

```
% darcs pull http://your.server.org/repos/yourproject
```

Darcs will check to see if you have recorded any changes that aren't in my current repository. If so, it'll prompt me for each one, to see which ones I want to add to my repository. Note that you may see a different series of prompts depending on your answers, since sometimes one patch depends on another, so if you answer yes to the first one, you won't be prompted for the second if the first depends on it.

Of course, maybe I don't even have a copy of your repository. In that case I'd want to do a

```
% darcs get http://your.server.org/repos/yourproject
```

which gets the whole repo.

I could instead create an empty repository and fetch all of your patches with pull. Get is just a more efficient way to clone a whole repository.

Get, pull and push also work over ssh. Ssh-paths are of the same form accepted by scp, namely `[username@]host:/path/to/repository`.

3.5 Moving patches from one repo to another

Darcs is flexible as to how you move patches from one repo to another. This section will introduce all the ways you can get patches from one place to another, starting with the simplest and moving to the most complicated.

3.5.1 All pulls

The simplest method is the "all-pull" method. This involves making each repository readable (by http, ftp, nfs-mounted disk, whatever), and you run `darcs pull` in the repo you want to move the patch to. This is nice, as it doesn't require you to give write access to anyone else, and is reasonably simple.

3.5.2 Send and apply manually

Sometimes you have a machine on which it is not convenient to set up a web server, perhaps because it's behind a firewall or perhaps for security reasons, or because it is often turned off. In this case you can use darcs send from that computer to generate a patch bundle destined for another repository. You can either let darcs email the patch for you, or save it as a file and transfer it by hand. Then in the destination repository you (or the owner of that repo) run darcs apply to apply the patches contained in the bundle. This is also quite a simple method since, like the all-pull method, it doesn't require that you give anyone write access to your repository. But it's less convenient, since you have to keep track of the patch bundle (in the email, or whatever).

If you use the send and apply method with email, you'll probably want to create a `_darcs/prefs/email` file containing your email address. This way anyone who sends to your repository will automatically send the patch bundle to your email address.

If you receive many patches by email, you probably will benefit by running darcs apply directly from your mail program. I have in my `.muttrc` the following

```
macro pager A "<pipe-entry>darcs apply --verbose --mark-conflicts \
        --reply droundy@abridgegame.org --repodir ~/darcs"
```

which allows me to apply patches directly from `mutt`, sending a confirmation email to the person who sent me the patch.

3.5.3 Push

If you use ssh (and preferably also ssh-agent, so you won't have to keep retyping your password), you can use the push method to transfer changes (using the scp protocol for communication). This method is again not very complicated, since you presumably already have the ssh permissions set up. Push can also be used when the target repository is local, in which case ssh isn't needed. On the other hand, in this situation you could as easily run a pull, so there isn't much benefit.

Note that you can use push to administer a multiple-user repository. You just need to create a user for the repository (or repositories), and give everyone with write access ssh access, perhaps using `.ssh/authorized_keys`. Then they run

```
% darcs push repouser@repo.server:repo/directory
```

3.5.4 Push —apply-as

Now we get more subtle. If you like the idea in the previous paragraph about creating a repository user to own a repository which is writable by a number of users, you have one other option.

Push `--apply-as` can run on either a local repository or one accessed with ssh, but uses `sudo` to run a darcs apply command (having created a patch bundle as in send) as another user. You can add the following line in your sudoers file to allow the users to apply their patches to a centralized repository:

```
ALL     ALL = (repo-user) NOPASSWD: /usr/bin/darcs apply --all --repodir /repo/path
```

This method is ideal for a centralized repository when all the users have accounts on the same computer, if you don't want your users to be able to run arbitrary commands as repo-user.

3.5.5 Sending signed patches by email

Most of the previous methods are a bit clumsy if you don't want to give each person with write access to a repo an account on your server. Darcs send can be configured to send a cryptographically signed patch by email. You can then set up your mail system to have darcs verify that patches were signed by an authorized user and apply them when a patch is received by email. The results of the apply can be returned to the user by email. Unsigned patches (or patches signed by unauthorized users) will be forwarded to the repository owner (or whoever you configure them to be forwarded to. . .).

This method is especially nice when combined with the `--test` option of darcs apply, since it allows you to run the test suite (assuming you have one) and reject patches that fail—and it's all done on the server, so you can happily go on working on your development machine without slowdown while the server runs the tests.

Setting up darcs to run automatically in response to email is by far the most complicated way to get patches from one repo to another. . . so it'll take a few sections to explain how to go about it.

Security considerations When you set up darcs to run apply on signed patches, you should assume that a user with write access can write to any file or directory that is writable by the user under which the apply process runs. Unless you specify the `--no-test` flag to darcs apply (and this is *not* the default), you are also allowing anyone with write access to that repository to run arbitrary code on your machine (since they can run a test suite—which they can modify however they like). This is quite a potential security hole.

For these reasons, if you don't implicitly trust your users, it is recommended that you create a user for each repository to limit the damage an attacker can do with access to your repository. When considering who to trust, keep in mind that a security breach on any developer's machine could give an attacker access to their private key and passphrase, and thus to your repository.

Installing necessary programs You also must install the following programs: gnupg, a mailer configured to receive mail (e.g. exim, sendmail or postfix), and a web server (usually apache). If you want to be able to browse your repository on the web you must also configure your web server to run cgi scripts and make sure the darcs cgi script was properly installed (by either a darcs-server package, or 'make install-server').

Setting up a repository with its own user To create a repository, as root run the '`darcs-createrepo`'. You will be prompted for the email address of the repository and the location of an existing copy of the repository. If your desired email is "myproject@my.url", this will create a user named "myproject" with a home directory of `/var/lib/darcs/repos/myproject`. FIXME: I have no idea if the darcs-createrepo program will even run on any system other than debian. Success reports would be appreciated (or of course bug reports if it fails).

The “myproject” user will be configured to run the darcs patcher on any emails it receives. However, the patcher will bounce any emails which aren’t signed by a key in the `/var/lib/darcs/repos/myproject/allowed_keys` gpg keyring (which is empty). To give yourself access to this repository you will need to create a gpg key. If you don’t know about public key cryptography, take a look at the gnupg manual.

Granting access to a repository You create your gpg key by running (as your normal user):

```
% gpg --gen-key
```

You will be prompted for your name and email address, among other options. To add your public key to the allowed keys keyring. Of course, you can skip this step if you already have a gpg key you wish to use.

You now need to export the public key so we can tell the patcher about it. You can do this with the following command (again as your normal user):

```
% gpg --export "email@address" > /tmp/exported_key
```

And now we can add your key to the `allowed_keys`:

```
(as root)> gpg --keyring /var/lib/darcs/repos/myproject/allowed_keys \
               --no-default-keyring --import /tmp/exported_key
```

You can repeat this process any number of times to authorize multiple users to send patches to the repository.

You should now be able to send a patch to the repository by running as your normal user, in a working copy of the repository:

```
% darcs send --sign http://your.computer/repos/myproject
```

You may want to add “send sign” to the file `_darcs/prefs/defaults` so that you won’t need to type `--sign` every time you want to send…

If your gpg key is protected by a passphrase, then executing `send` with the `--sign` option might give you the following error:

```
darcs failed:  Error running external program 'gpg'
```

The most likely cause of this error is that you have a misconfigured gpg that tries to automatically use a non-existent gpg-agent program. GnuPG will still work without gpg-agent when you try to sign or encrypt your data with a passphrase protected key. However, it will exit with an error code 2 (`ENOENT`) causing `darcs` to fail. To fix this, you will need to edit your `~/.gnupg/gpg.conf` file and comment out or remove the line that says:

```
use-agent
```

If after commenting out or removing the `use-agent` line in your gpg configuration file you still get the same error, then you probably have a modified GnuPG with use-agent as a hard-coded option. In that case, you should change `use-agent` to `no-use-agent` to disable it explicitly.

Setting up a sendable repository using procmail If you don't have root access on your machine, or perhaps simply don't want to bother creating a separate user, you can set up a darcs repository using procmail to filter your mail. I will assume that you already use procmail to filter your email. If not, you will need to read up on it, or perhaps should use a different method for routing the email to darcs.

To begin with, you must configure your repository so that a darcs send to your repository will know where to send the email. Do this by creating a file in `/path/to/your/repo/_darcs/prefs` called `email` containing your email address. As a trick (to be explained below), we will create the email address with "darcs repo" as your name, in an email address of the form "David Roundy <droundy@abridgegame.org>."

```
% echo 'my darcs repo <user@host.com>' > /path/to/your/repo/_darcs/prefs/email
```

The next step is to set up a gnupg keyring containing the public keys of people authorized to send to your repo. Here I'll give a second way of going about this (see above for the first). This time I'll assume you want to give me write access to your repository. You can do this by:

```
gpg --no-default-keyring \
    --keyring /path/to/the/allowed_keys --recv-keys D3D5BCEC
```

This works because "D3D5BCEC" is the ID of my gpg key, and I have uploaded my key to the gpg keyservers. Actually, this also requires that you have configured gpg to access a valid keyserver. You can, of course, repeat this command for all keys you want to allow access to.

Finally, we add a few lines to your `.procmailrc`:

```
:0:
* ^TOmy darcs repo
|(umask 022; darcs apply --reply user@host.com \
    --repodir /path/to/your/repo --verify /path/to/the/allowed_keys)
```

The purpose for the "my darcs repo" trick is partially to make it easier to recognize patches sent to the repository, but is even more crucial to avoid nasty bounce loops by making the `--reply` option have an email address that won't go back to the repository. This means that unsigned patches that are sent to your repository will be forwarded to your ordinary email.

I find that I need the "umask 022" in order to keep procmail from setting the umask incorrectly, which causes the repository to no longer be world-readable.

Checking if your e-mail patch was applied After sending a patch with `darcs send`, you may not receive any feedback, even if the patch is applied. You can confirm whether or not your patch was applied to the remote repo by pointing `darcs changes` at a remote repo:

```
darcs changes --last=10 --repo=http://abridgegame.org/repos/darcs
```

That shows you the last 10 changes in the remote repo. You can adjust the options given to `changes` if a more advanced query is needed.

3.6 Reducing disk space usage

A Darcs repository contains the patches that Darcs uses to store history, the working directory, and a *pristine tree* (a copy of the working directory files with no local modifications). For large repositories, this can add up to a fair amount of disk usage.

There are two techniques that can be used to reduce the amount of space used by Darcs repositories: linking and using no pristine tree. The former can be used on any repository; the latter is only suitable in special circumstances, as it makes some operations much slower.

3.6.1 Linking between repositories

A number of filesystems support *linking* files, sharing a single file data between different directories. Under some circumstances, when repositories are very similar (typically because they represent different branches of the same piece of software), Darcs will use linking to avoid storing the same file multiple times.

Whenever you invoke `darcs get` to copy a repository from a local filesystem onto the same filesystem, Darcs will link patches whenever possible.

In order to save time, `darcs get` does not link pristine trees even when individual files are identical. Additionally, as you pull patches into trees, patches will become unlinked. This will result in a lot of wasted space if two repositories have been living for a long time but are similar. In such a case, you should *relink* files between the two repositories.

Relinking is an asymmetric operation: you relink one repository (to which you must have write access) to another repository, called the *sibling*. This is done with `darcs optimize --relink`, with –the `--sibling` flag specifying the sibling.

```
$ cd /var/repos/darcs-unstable
$ darcs optimize --relink --sibling /var/repos/darcs
```

The `--sibling` flag can be repeated multiple times, in which case Darcs will try to find a file to link to in all of the siblings. If a default repository is defined, Darcs will try, as a last resort, to link against the default repository.

Additional space savings can be achieved by relinking files in the pristine tree (see below) by using the `--relink-pristine` flag. However, doing this prevents Darcs from having precise timestamps on the pristine files, which carries a moderate performance penalty.

3.6.2 Alternate formats for the pristine tree

By default, every Darcs repository contains a complete copy of the *pristine tree*, the working tree as it would be if there were no local edits. By avoiding the need to consult a possibly large number of patches just to find out if a file is modified, the pristine tree makes a lot of operations much faster than they would otherwise be.

Under some circumstances, keeping a whole pristine tree is not desirable. This is the case when preparing a repository to back up, when publishing a repository on a public web server with limited space, or when storing a repository on floppies or small USB keys. In such cases, it is possible to use a repository with no pristine tree.

Darcs automatically recognizes a repository with no pristine tree. In order to create such a tree, specify the `--no-pristine-tree` flag to `darcs initialize` or `darcs get`. There is currently no way to switch an existing repository to use no pristine tree.

The support for `--no-pristine-tree` repositories is fairly new, and has not been extensively optimized yet. Please let us know if you use this functionality, and which operations you find are too slow.

Chapter 4

Configuring darcs

There are several ways you can adjust darcs' behavior to suit your needs. The first is to edit files in the `_darcs/prefs/` directory of a repository. Such configuration only applies when working with that repository. To configure darcs on a per-user rather than per-repository basis (but with essentially the same methods), you can edit (or create) files in the `~/.darcs/` directory. Finally, the behavior of some darcs commands can be modified by setting appropriate environment variables.

4.1 prefs

The `_darcs` directory contains a `prefs` directory. This directory exists simply to hold user configuration settings specific to this repository. The contents of this directory are intended to be modifiable by the user, although in some cases a mistake in such a modification may cause darcs to behave strangely.

defaults Default values for darcs commands can be configured on a per-repository basis by editing (and possibly creating) the `_darcs/prefs/defaults` file. Each line of this file has the following form:

```
COMMAND FLAG VALUE
```

where `COMMAND` is either the name of the command to which the default applies, or `ALL` to indicate that the default applies to all commands accepting that flag. The `FLAG` term is the name of the long argument option without the "`--`", i.e. `verbose` rather than `--verbose`. Finally, the `VALUE` option can be omitted if the flag is one such as `verbose` that doesn't involve a value. Each line only takes one flag. To set multiple defaults for the same command (or for `ALL` commands), use multiple lines.

```
    ~/.darcs/defaults              provides defaults for this user account
    project/_darcs/prefs/defaults  provides defaults for one project, overrules changes per u
```

For example, if your system clock is bizarre, you could instruct darcs to always ignore the file modification times by adding the following line to your `_darcs/prefs/defaults` file. (Note that this would have to be done for each repository!)

```
ALL ignore-times
```

If you never want to run a test when recording to a particular repository (but still want to do so when running check on that repo), and like to name all your patches "Stupid patch", you could use the following:

```
record no-test
record patch-name Stupid patch
```

If you would like a command to be run every time patches are recorded in a particular repository (for example if you have one central repository, that all developers contribute to), then you can set apply to always run a command when apply is successful. For example, if you need to make sure that the files in the repository have the correct access rights you might use the following. There are two things to note about using darcs this way:

- Without the second line you will get errors, because the sub process that runs apply cannot prompt interactively.
- Whatever script is run by the post apply command should not be be added to the repository with `darcs add`; doing so would allow people to modify that file and then run arbitrary scripts on your main repository, possibly damaging or violating security.

```
apply posthook chmod -R a+r *
apply run-posthook
```

There are some options which are meant specifically for use in `_darcs/prefs/defaults`. One of them is `--disable`. As the name suggests, this option will disable every command that got it as argument. So, if you are afraid that you could damage your repositories by inadvertent use of a command like amend-record, add the following line to `_darcs/prefs/defaults`:

```
amend-record disable
```

Also, a global preferences file can be created with the name `.darcs/defaults` in your home directory. Options present there will be added to the repository-specific preferences. If they conflict with repository-specific options, the repository-specific ones will take precedence.

repos The `_darcs/prefs/repos` file contains a list of repositories you have pulled from or pushed to, and is used for autocompletion of pull and push commands in bash. Feel free to delete any lines from this list that might get in there, or to delete the file as a whole.

author The `_darcs/prefs/author` file contains the email address (or name) to be used as the author when patches are recorded in this repository, e.g. `David Roundy <droundy@abridgegame.org>`. This file overrides the contents of the environment variables `$DARCS_EMAIL` and `$EMAIL`.

boring The `_darcs/prefs/boring` file may contain a list of regular expressions describing files, such as object files, that you do not expect to add to your project. As an example, the boring file that I use with my darcs repository is:

```
\.hi$
\.o$
^\.[^/]
^_
~$
(^|/)CVS($|/)
```

A newly created repository has a longer boring file that includes many common source control, backup, temporary, and compiled files.

You may want to have the boring file under version control. To do this you can use darcs setpref to set the value "boringfile" to the name of your desired boring file (e.g. "darcs setpref boringfile .boring", where .boring is the repository path of a file that has been darcs added to your repository). The boringfile pref overrides `_darcs/prefs/boring`, so be sure to copy that file to the boringfile.

You can also set up a "boring" regexps file in your home directory, named `~/.darcs/boring`, which will be used with all of your darcs repositories.

Any file whose repository path (such as `manual/index.html`) matches any of the boring regular expressions is considered boring. The boring file is used to filter the files provided to darcs add, to allow you to use a simple "darcs add newdir newdir/*" without accidentally adding a bunch of object files. It is also used when the `--look-for-adds` flag is given to whatsnew or record.

binaries The `_darcs/prefs/binaries` file may contain a list of regular expressions describing files that should be treated as binary files rather than text files. You probably will want to have the binaries file under version control. To do this you can use darcs setpref to set the value "binariesfile" to the name of your desired binaries file (e.g. "darcs setpref binariesfile ./.binaries", where .binaries is a file that has been darcs added to your repository). As with the boring file, you can also set up a `~/.darcs/binaries` file if you like.

email The `_darcs/prefs/email` file is used to provide the e-mail address for your repo that others will use when they `darcs send` a patch back to you. The contents of the file should simply be an e-mail address.

motd The `_darcs/prefs/motd` file may contain a "message of the day" which will be displayed to users who get or pull from the repo without the `--quiet` option.

4.2 Environment variables

There are a few environment variables whose contents affect darcs' behavior.

DARCS_EMAIL The DARCS_EMAIL environment variable determines the "author" name used by darcs when recording if no `_darcs/prefs/author` exists. If DARCS_EMAIL is undefined, the contents of the EMAIL environment variable are used.

DARCS_EDITOR When pulling up an editor (for example, when adding a long comment in record), darcs uses the contents of DARCS_EDITOR if it is defined. If not, it tries the contents of VISUAL, and if that isn't defined (or fails for some reason), it tries EDITOR. If none of those environment variables are defined, darcs tries `vi`, `emacs`, `emacs -nw` and `nano` in that order.

DARCS_TMPDIR If the environment variable DARCS_TMPDIR is defined, darcs will use that directory for its temporaries. Otherwise it will use TMPDIR, if that is defined, and if not that then `/tmp` and if `/tmp` doesn't exist, it'll put the temporaries in `_darcs`.

This is very helpful, for example, when recording with a test suite that uses MPI, in which case using `/tmp` to hold the test copy is no good, as `/tmp` isn't shared over NFS and thus the `mpirun` call will fail, since the binary isn't present on the compute nodes.

HOME HOME is used to find the per-user prefs directory, which is located at `$HOME/.darcs`.

TERM If darcs is compiled with libcurses support and support for color output, it uses the environment variable TERM to decide whether or not color is supported on the output terminal.

SSH_PORT When using ssh, if the SSH_PORT environment variable is defined, darcs will use that port rather than the default ssh port (which is 22).

DARCS_SSH The DARCS_SSH environment variable defines the command that darcs will use when asked to run ssh. This command is *not* interpreted by a shell, so you cannot use shell metacharacters, and the first word in the command must be the name of an executable located in your path.

DARCS_SCP and DARCS_SFTP The DARCS_SCP and DARCS_SFTP environment variables define the commands that darcs will use when asked to run scp or sftp. Note that scp and sftp is how darcs accesses repositories whose URL is of the form `user@foo.org:foo` or `foo.org:foo`. Darcs will use scp to copy single files (e.g. repository meta-information), and sftp to copy multiple files in batches (e.g. patches). These commands are *not* interpreted by a shell, so you cannot use shell metacharacters, and the first word in the command must be the name of an executable located in your path.

DARCS_PROXYUSERPWD This environment variable allows DARCS and libcurl to access remote repositories via a password-protected HTTP proxy. The proxy itself is specified with the standard environment variable for this purpose, namely 'http_proxy'. The DARCS_PROXYUSERPWD environment variable specifies the proxy username and password. It must be given in the form *username:password*.

DARCS_GET_FOO, DARCS_MGET_FOO and DARCS_APPLY_FOO When trying to access a repository with a url beginning foo://, darcs will invoke the program specified by the DARCS_GET_FOO environment variable (if defined) to download each file, and the command specified by the DARCS_APPLY_FOO environment variable (if defined) when pushing to a foo:// url.

This method overrides all other ways of getting `foo://xxx urls`.

Note that each command should be constructed so that it sends the downloaded content to STDOUT, and the next argument to it should be the URL. Here are some examples that should work for DARCS_GET_HTTP:

```
fetch -q -o -
curl -s -f
lynx -source
wget -q -O -
```

If set, DARCS_MGET_FOO will be used to fetch many files from a single repository simultaneously. Replace FOO and foo as appropriate to handle other URL schemes. These commands are *not* interpreted by a shell, so you cannot use shell metacharacters, and the first word in the command must be the name of an executable located in your path. The GET command will be called with a url for each file, the MGET command will be invoked with a number of urls and is expected to download the files to the current directory, preserving the filename but not the path, the APPLY command will be called with a darcs patchfile piped into its standard input. Example:

```
wget -q
```

DARCS_MGETMAX When invoking a DARCS_MGET_FOO command, darcs will limit the number of urls presented to the command to the value of this variable, if set, or 200.

DARCS_WGET This is a very old option that is only used if libcurl is not compiled in and one of the DARCS_GET_FOO is not used. Using one of those is recommended instead.

The DARCS_WGET environment variable defines the command that darcs will use to fetch all URLs for remote repositories. The first word in the command must be the name of an executable located in your path. Extra arguments can be included as well, such as:

```
wget -q
```

Darcs will append `-i` to the argument list, which it uses to provide a list of URLS to download. This allows wget to download multiple patches at the same time. It's possible to use another command besides `wget` with this environment variable, but it must support the `-i` option in the same way.

These commands are *not* interpreted by a shell, so you cannot use shell meta-characters.

4.3 Highlighted output

If the terminal understands ANSI color codes, darcs will highlight certain keywords and delimiters when printing patches. This can be turned off by setting the environment variable DARCS_DONT_COLOR to 1. If you use a pager that happens to understands ANSI colors, like `less -R`, darcs can be forced to always highlight the output by setting DARCS_ALWAYS_COLOR to 1. If you can't see colors you can set DARCS_ALTERNATIVE_COLOR to 1, and darcs will use ANSI codes for bold and reverse video instead of colors.

By default darcs will escape (by highlighting if possible) any kind of spaces at the end of lines when showing patch contents. If you don't want this you can turn it off by setting DARCS_DONT_ESCAPE_TRAILING_SPACES to 1. A special case exists for only carriage returns: DARCS_DONT_ESCAPE_TRAILING_CR.

4.4 Character escaping and non-ASCII character encodings

Darcs needs to escape certain characters when printing patch contents to a terminal. Characters like *backspace* can otherwise hide patch content from the user, and other character sequences can even in some cases redirect commands to the shell if the terminal allows it.

By default darcs will only allow printable 7-bit ASCII characters (including space), and the two control characters *tab* and *newline*. (See the last paragraph in this section for a way to tailor this behavior.) All other octets are printed in quoted form (as `^<control letter>` or `\<hex code>`).

Darcs has some limited support for locales. If the systems locale is a single-byte character encoding, like the Latin encodings, you can set the environment variable DARCS_DONT_ESCAPE_ISPRINT to 1 and darcs will display all the printables in the current system locale instead of just the ASCII ones. NOTE: This does curently not work on some architectures if darcs is compiled with

GHC 6.4. Some non-ASCII control characters might be printed and can possibly spoof the terminal.

For multi-byte character encodings things are less smooth. UTF-8 will work if you set DARCS_DONT_ESCAPE_8BIT to 1, but non-printables outside the 7-bit ASCII range are no longer escaped. E.g., the extra control characters from Latin1 might leave your terminal at the mercy of the patch contents. Space characters outside the 7-bit ASCII range are no longer recognized and will not be properly escaped at line endings.

As a last resort you can set DARCS_DONT_ESCAPE_ANYTHING to 1. Then everything that doesn't flip code sets should work, and so will all the bells and whistles in your terminal. This environment variable can also be handy if you pipe the output to a pager or external filter that knows better than darcs how to handle your encoding. Note that *all* escaping, including the special escaping of any line ending spaces, will be turned off by this setting.

There are two environment variables you can set to explicitly tell darcs to not escape or escape octets. They are DARCS_DONT_ESCAPE_EXTRA and DARCS_ESCAPE_EXTRA. Their values should be strings consisting of the verbatim octets in question. The do-escapes take precedence over the dont-escapes. Space characters are still escaped at line endings though. The special environment variable DARCS_DONT_ESCAPE_TRAILING_CR turns off escaping of carriage return last on the line (DOS style).

Chapter 5

Best practices

5.1 Introduction

This chapter is intended to review various scenarios and describe in each case effective ways of using darcs. There is no one "best practice", and darcs is a sufficiently low-level tool that there are many high-level ways one can use it, which can be confusing to new users. The plan (and hope) is that various users will contribute here describing how they use darcs in different environments. However, this is not a wiki, and contributions will be edited and reviewed for consistency and wisdom.

5.2 Creating patches

This section will lay down the concepts around patch creation. The aim is to develop a way of thinking that corresponds well to how darcs is behaving — even in complicated situations.

In a single darcs repository you can think of two "versions" of the source tree. They are called the *working* and *pristine* trees. *Working* is your normal source tree, with or without darcs alongside. The only thing that makes it part of a darcs repository is the `_darcs` directory in its root. *Pristine* is the recorded state of the source tree. The pristine tree is constructed from groups of changes, called *patches* (some other version control use the term *changeset* instead of *patch*).[1] Darcs will create and store these patches based on the changes you make in *working*.

[1]If you look inside _darcs you will find files or directories named `patches` and `inventories`, which store all the patches ever recorded. If the repository holds a cached pristine tree, it is stored in a directory called `pristine` or `current`; otherwise, the fact that there is no pristine tree is marked by the presence of a file called `pristine.none` or `current.none`.

5.2.1 Changes

If *working* and *pristine* are the same, there are “no changes” in the repository. Changes can be introduced (or removed) by editing the files in *working*. They can also be caused by darcs commands, which can modify *both working* and *pristine*. It is important to understand for each darcs command how it modifies *working*, *pristine* or both of them.

`whatsnew` (as well as `diff`) can show the difference between *working* and *pristine* to you. It will be shown as a difference in *working*. In advanced cases it need *not* be *working* that has changed; it can just as well have been *pristine*, or both. The important thing is the difference and what darcs can do with it.

5.2.2 Keeping or discarding changes

If you have a difference in *working*, you do two things with it: `record` it to keep it, or `revert` it to lose the changes.[2]

If you have a difference between *working* and *pristine*—for example after editing some files in *working*—`whatsnew` will show some “unrecorded changes”. To save these changes, use `record`. It will create a new patch in *pristine* with the same changes, so *working* and *pristine* are no longer different. To instead undo the changes in *working*, use `revert`. It will modify the files in *working* to be the same as in *pristine* (where the changes do not exist).

5.2.3 Unrecording changes

`unrecord` is a command meant to be run only in private repositories. Its intended purpose is to allow developers the flexibility to undo patches that haven't been distributed yet.

However, darcs does not prevent you from unrecording a patch that has been copied to another repository. Be aware of this danger!

If you `unrecord` a patch, that patch will be deleted from *pristine*. This will cause *working* to be different from *pristine*, and `whatsnew` to report unrecorded changes. The difference will be the same as just before that patch was `record`ed. Think about it. `record` examines what's different with *working* and constructs a patch with the same changes in *pristine* so they are no longer different. `unrecord` deletes this patch; the changes in *pristine* disappear and the difference is back.

If the recorded changes included an error, the resulting flawed patch can be unrecorded. When the changes have been fixed, they can be recorded again as a new—hopefully flawless—patch.

If the whole change was wrong it can be discarded from *working* too, with `revert`. `revert` will update *working* to the state of *pristine*, in which the changes do no longer exist after the patch was deleted.

[2]Revert can undo precious work in a blink. To protect you from great grief, the discarded changes are saved temporarily so the latest revert can be undone with unrevert.

Keep in mind that the patches are your history, so deleting them with `unrecord` makes it impossible to track what changes you *really* made. Redoing the patches is how you "cover the tracks". On the other hand, it can be a very convenient way to manage and organize changes while you try them out in your private repository. When all is ready for shipping, the changes can be reorganized in what seems as useful and impressive patches. Use it with care.

All patches are global, so don't *ever* replace an already "shipped" patch in this way! If an erroneous patch is deleted and replaced with a better one, you have to replace it in *all* repositories that have a copy of it. This may not be feasible, unless it's all private repositories. If other developers have already made patches or tags in their repositories that depend on the old patch, things will get complicated.

5.2.4 Special patches and pending

The patches described in the previous sections have mostly been hunks. A *hunk* is one of darcs' primitive patch types, and it is used to remove old lines and/or insert new lines. There are other types of primitive patches, such as *adddir* and *addfile* which add new directories and files, and *replace* which does a search-and-replace on tokens in files.

Hunks are always calculated in place with a diff algorithm just before `whatsnew` or `record`. But other types of primitive patches need to be explicitly created with a darcs command. They are kept in *pending*[3] until they are either recorded or reverted.

Pending can be thought of as a special extension of *working.* When you issue, e.g., a darcs `replace` command, the replace is performed on the files in *working* and at the same time a replace patch is put in *pending.* Patches in *pending* describe special changes made in *working.* The diff algorithm will fictively apply these changes to *pristine* before it compares it to *working*, so all lines in *working* that are changed by a `replace` command will also be changed in *pending+pristine* when the hunks are calculated. That's why no hunks with the replaced lines will be shown by `whatsnew`; it only shows the replace patch in *pending* responsible for the change.

If a special patch is recorded, it will simply be moved to *pristine.* If it is instead reverted, it will be deleted from *pending* and the accompanying change will be removed from *working.*

Note that reverting a patch in pending is *not* the same as simply removing it from pending. It actually applies the inverse of the change to *working.* Most notable is that reverting an addfile patch will delete the file in *working* (the inverse of adding it). So if you add the wrong file to darcs by mistake, *don't* `revert` the addfile. Instead first rename the file, revert, and then rename it back.

[3]In the file `_darcs/patches/pending`.

5.3 Using patches

This section will lay down the concepts around patch distribution and branches. The aim is to develop a way of thinking that corresponds well to how darcs is behaving — even in complicated situations.

A repository is a collection of patches. Patches have no defined order, but patches can have dependencies on other patches. Patches can be added to a repository in any order as long as all depended upon patches are there. Patches can be removed from a repository in any order, as long as no remaining patches depend on them.

Repositories can be cloned to create branches. Patches created in different branches may conflict. A conflict is a valid state of a repository. A conflict makes the working tree ambiguous until the conflict is resolved.

5.3.1 Dependencies

There are two kinds of dependencies: implicit dependencies and explicit dependencies.

Implicit dependencies is the far most common kind. These are calculated automatically by darcs. If a patch removes a file or a line of code, it will have to depend on the patch that added that file or line of code.[4] If a patch adds a line of code, it will usually have to depend on the patch or patches that added the adjacent lines.

Explicit dependencies can be created if you give the `--ask-deps` option to `darcs record`. This is good for assuring that logical dependencies hold between patches. It can also be used to group patches—a patch with explicit dependencies doesn't need to change anything—and pulling the patch also pulls all patches it was made to depend on.

5.3.2 Branches: just normal repositories

Darcs does not have branches—it doesn't need to. Every repository can be used as a branch. This means that any two repositories are "branches" in darcs, but it is not of much use unless they have a large portion of patches in common. If they are different projects they will have nothing in common, but darcs may still very well be able to merge them, although the result probably is nonsense. Therefore the word "branch" isn't a technical term in darcs; it's just the way we think of one repository in relation to another.

Branches are *very* useful in darcs. They are in fact *necessary* if you want to do more than only simple work. When you `get` someone's repository from the Internet, you are actually creating a branch of it. It may first seem inefficient (or if you come from CVS—frightening), not to say plain awkward. But darcs is designed this way, and it has means to make it efficient. The answer to many questions about how to do a thing with darcs is: "use a branch". It is a simple

[4]Actually it doesn't have to—in theory—, but in practice it's hard to create "negative" files or lines in the working tree. See the chapter about Theory of patches for other constraints.

and elegant solution with great power and flexibility, which contributes to darcs' uncomplicated user interface.

You create new branches (i.e., clone repositories) with the `get` and `put` commands.

5.3.3 Moving patches around—no versions

Patches are global, and a copy of a patch either is or is not present in a branch. This way you can rig a branch almost any way you like, as long as dependencies are fulfilled—darcs *won't* let you break dependencies. If you suspect a certain feature from some time ago introduced a bug, you can remove the patch/patches that adds the feature, and try without it.[5]

Patches are added to a repository with `pull` and removed from the repositories with `unpull`. Don't confuse these two commands with `record` and `unrecord`, which constructs and deconstructs patches.

It is important not to lose patches when (re)moving them around. `pull` needs a source repository to copy the patch from, whereas `unpull` just erases the patch. Beware that if you unpull *all* copies of a patch it is completely lost—forever. Therefore you should work with branches when you unpull patches. The `unpull` command can wisely be disabled in a dedicated main repository by adding `unpull disable` to the repository's defaults file.

For convenience, there is a `push` command. It works like `pull` but in the other direction. It also differs from `pull` in an important way: it starts a second instance of darcs to apply the patch in the target repository, even if it's on the same computer. It can cause surprises if you have a "wrong" darcs in your PATH.

5.3.4 Tags—versions

While `pull` and `unpull` can be used to construct different "versions" in a repository, it is often desirable to name specific configurations of patches so they can be identified and retrieved easily later. This is how darcs implements what is usually known as versions. The command for this is `tag`, and it records a tag in the current repository.

A tag is just a patch, but it only contains explicit dependencies. It will depend on all the patches in the current repository.[6] Darcs can recognize if a patch is as a tag; tags are sometimes treated specially by darcs commands.

While traditional revision control systems tag versions in the time line history, darcs lets you tag any configuration of patches at any time, and pass the tags around between branches.

With the option `--tag` to `get` you can easily get a named version in the repository as a new branch.

[5]darcs even has a special command, `trackdown` that automatically removes patches until a specified test no longer fails.

[6]It will omit patches already depended upon by other patches, since they will be indirectly depended upon anyway.

5.3.5 Conflicts

This part of darcs becomes a bit complicated, and the description given here is slightly simplified.

Conflicting patches are created when you record changes to the same line in two different repositories. Same line does *not* mean the same line number and file name, but the same line added by a common depended upon patch.

Contrary to many other merging tools, darcs considers two patches making the *same* change to be a conflict. In fact, darcs doesn't even look at the contents of the conflicting lines. If you think this is wrong, think about two different patches each adding a new keyword and also changing the line "`#define NUM_OF_KEYWORDS 17`" to "`#define NUM_OF_KEYWORDS 18`".

A conflict *happens* when two conflicting patches meet in the same repository. This is no problem for darcs; it can happily pull together just any patches. But it is a problem for the files in *working* (and *pristine*). The conflict can be thought of as two patches telling darcs different things about what a file should look like.

Darcs escapes this problem by ignoring those parts[7] of the patches that conflict. They are ignored in *both* patches. If patch A changes the line "FIXME" to "FIXED", and patch B changes the same line to "DONE", the two patches together will produce the line "FIXME". Darcs doesn't care which one you pulled into the repository first, you still get the same result when the conflicting patches meet. All other changes made by A and B are performed as normal.

Darcs can mark a conflict for you in *working*. This is done with `resolve` (which isn't a very good name). Conflicts are marked such that both conflicting changes are inserted with special delimiter lines around them. Then you can merge the two changes by hand, and remove the delimiters.

When you pull patches, darcs automatically performs a `resolve` for you if a conflict happens. You can remove the markup with `revert`, Remember that the result will be the lines from the previous version common to both conflicting patches. The conflict marking can be redone again with `resolve`.

A special case is when a pulled patch conflicts with unrecorded changes in the repository. The conflict will be automatically marked as usual, but since the markup is *also* an unrecorded change, it will get mixed in with your unrecorded changes. There is no guarantee you can revert `only` the markup after this, and `resolve` will not be able to redo this markup later if you remove it. It is good practice to record important changes before pulling.

`resolve` can't mark complicated conflicts. In that case you'll have to use `darcs diff` and other commands to understand what the conflict is all about. If for example two conflicting patches create the same file, `resolve` will pick just one of them, and no delimiters are inserted. So watch out if darcs tells you about a conflict.

`resolve` can also be used to check for unresolved conflicts. If there are none, darcs replies "No conflicts to resolve". While `pull` reports when a conflict happens, `unpull` and `get` don't.

[7]The primitive patches making up the total patch.

5.3.6 Resolving conflicts

A conflict is resolved (not marked, as with the command `resolve`) as soon as some new patch depends on the conflicting patches. This will usually be the resolve patch you record after manually putting together the pieces from the conflict markup produced by `resolve` (or `pull`). But it can just as well be a tag. So don't forget to fix conflicts before you accidently "resolve" them by recording other patches.

If the conflict is with one of your not-yet-published patches, you may choose to amend that patch rather than creating a resolve patch.

If you want to back out and wait with the conflict, you can `unpull` the conflicting patch you just pulled. Before you can do that you have to `revert` the conflict markups that `pull` inserted when the conflict happened.

5.4 Distributed development with one primary developer

This is how darcs itself is developed. There are many contributors to darcs, but every contribution is reviewed and manually applied by myself. For this sort of a situation, `darcs send` is ideal, since the barrier for contributions is very low, which helps encourage contributors.

One could simply set the `_darcs/prefs/email` value to the project mailing list, but I also use darcs send to send my changes to the main server, so instead the email address is set to "`Davids Darcs Repo <droundy@abridgegame.org>`". My .procmailrc file on the server has the following rule:

```
:0:
* ^TODavids Darcs Repo
|(umask 022; darcs apply --reply darcs-devel@abridgegame.org \
             --repodir /path/to/repo --verify /path/to/allowed_keys)
```

This causes darcs apply to be run on any emails sent to "Davids Darcs Repo". Apply actually applies them only if they are signed by an authorized key. Currently, the only authorized key is mine, but of course this could be extended easily enough.

The central darcs repository contains the following values in its `_darcs/prefs/defaults`:

```
apply test
apply verbose
apply happy-forwarding
```

The first line tells apply to always run the test suite. The test suite is in fact the main reason I use send rather than push, since it allows me to easily continue working (or put my computer to sleep) while the tests are being run on the main server. The second line is just there to improve the email response that I get when a patch has either been applied or failed the tests. The third line makes darcs not complain about unsigned patches, but just to forward them to darcs-devel.

On my development computer, I have in my .muttrc the following alias, which allows me to easily apply patches that I get via email directly to my darcs working directory:

```
macro pager A "<pipe-entry>(umask 022; darcs apply --no-test -v --repodir ~/darcs)"
```

5.5 Development by a small group of developers in one office

This section describes the development method used for the density functional theory code DFT++, which is available at http://dft.physics.cornell.edu/dft.

We have a number of workstations which all mount the same /home via NFS. We created a special "dft" user, with the central repository living in that user's home directory. The ssh public keys of authorized persons are added to the "dft" user's .ssh/allowed_keys, and we commit patches to this repository using darcs push. As in Section 5.4, we have the central repository set to run the test suite before the push goes through.

Note that no one ever runs as the dft user.

A subtlety that we ran into showed up in the running of the test suite. Since our test suite includes the running of MPI programs, it must be run in a directory that is mounted across our cluster. To achieve this, we set the $DARCS_TMPDIR environment variable to ~/tmp.

Note that even though there are only four active developers at the moment, the distributed nature of darcs still plays a large role. Each developer works on a feature until it is stable, a process that often takes quite a few patches, and only once it is stable pushes to the central repo.

Chapter 6

Darcs commands

The general format of a darcs command is

```
% darcs COMMAND OPTIONS ARGUMENTS ...
```

Here `COMMAND` is a command such as `add` or `record`, which of course may have one or more arguments. Options have the form `--option` or `-o`, while arguments vary from command to command. There are many options which are common to a number of different commands, which will be summarized here.

If you wish, you may use any unambiguous beginning of a command name as a shortcut: for `darcs record`, you could type `darcs recor` or `darcs rec`, but not `darcs re` since that could be confused with `darcs replace`, `darcs revert` and `darcs remove`.

In some cases, `COMMAND` actually consists of two words, a super-command and a subcommand. For example, the "display the manifest" command has the form `darcs query manifest`.

Command overview Not all commands modify the "patches" of your repository (that is, the named patches which other users can pull); some commands only affect the copy of the source tree you're working on (your "working directory"), and some affect both. This table summarizes what you should expect from each one and will hopefully serve as guide when you're having doubts about which command to use.

[1]But it affects the repository and working directory targeted by the push

[2]As for the other end, see apply

affects	patches	working directory
record	yes	no
unrecord	yes	no
rollback	yes	no
revert	no	yes
unrevert	no	yes
pull	yes	yes
unpull	yes	yes
apply	yes	yes
push[1]	no	no
send[2]	no	no
put[3]	no	no

6.1 Common options to darcs commands

`--help`

Every `COMMAND` accepts `--help` as an argument, which tells it to provide a bit of help. Among other things, this help always provides an accurate listing of the options available with that command, and is guaranteed never to be out of sync with the version of darcs you actually have installed (unlike this manual, which could be for an entirely different version of darcs).

```
% darcs COMMAND --help
```

`--disable`

Every `COMMAND` accepts the `--disable` option, which can be used in `_darcs/prefs/defaults` to disable some commands in the repository. This can be helpful if you want to protect the repository from accidental use of advanced commands like unpull, unrecord or amend-record.

`--verbose`

Most commands also accept the `--verbose` option, which tells darcs to provide additional output. The amount of verbosity varies from command to command.

`--repodir`

Another common option is the `--repodir` option, which allows you to specify the directory of the repository in which to perform the command. This option is used with commands, such as whatsnew, that ordinarily would be performed within a repository directory, and allows you to use those commands without actually being in the repo directory when calling the command. This is useful when running darcs as a pipe, as might be the case when running apply from a mailer.

Selecting patches Many commands operate on a patch or patches that have already been recorded. There are a number of options that specify which patches are selected for these operations: `--patch`, `--match`, `--tag`, and variants on these, which for `--patch` are `--patches`, `--from-patch`, and `--to-patch`. The `--patch` and `--tag` forms simply take (POSIX extended, aka `egrep`) regular expressions and match them against tag and patch names. `--match`, described below, allows more powerful patterns.

The plural forms of these options select all matching patches. The singular forms select the last matching patch. The range (from and to) forms select patches after or up to (both inclusive) the last matching patch.

These options use the current order of patches in the repository. darcs may reorder patches, so this is not necessarily the order of creation or the order in which patches were applied. However, as long as you are just recording patches in your own repository, they will remain in order.

Match Currently `--match` accepts five primitive match types, although there are plans to expand it to match more patterns. Also, note that the syntax is still preliminary and subject to change.

The first match type accepts a literal string which is checked against the patch name. The syntax is

```
darcs annotate --summary --match 'exact foo+bar'
```

This is useful for situations where a patch name contains characters that could be considered special for regular expressions.

The second match type accepts a regular expression which is checked against the patch name. The syntax is

```
darcs annotate --summary --match 'name foo'
```

If you want to include spaces in the regular expression, it must be enclosed in double quotes ("), and currently there is no provision for escaping a double quote, so you have to choose between matching double quotes and matching spaces.

The third match type matches the darcs hash for each patch:

```
darcs annotate --summary --match \
  'hash 20040403105958-53a90-c719567e92c3b0ab9eddd5290b705712b8b918ef'
```

This is intended to be used, for example, by programs allowing you to view darcs repositories (e.g. CGI scripts like viewCVS).

The fourth match type accepts a regular expression which is checked against the patch author. The syntax is

```
darcs annotate --summary --match 'author foo'
```

There is also now rudimentary support for matching by date. This is done using commands such as

```
darcs annotate --summary --match 'date "last week"'
darcs annotate --summary --match 'date yesterday'
darcs annotate --summary --match 'date today'
darcs changes --from-match 'date "Sat Jun  30 11:31:30 EDT 2004"'
```

currently date matching always matches only the day itself. FIXME: It should be extended to match the time as well if the time is specified. In general, a lot of cleanup is needed in the date matching code.

Because the date matching is only by day, you may prefer to combine it with a more specific pattern.

```
darcs annotate --summary --match 'date "last week" && name foo'
```

The `--match` pattern can include the logical operators `&&`, `||` and `not`, as well as grouping of patterns with parentheses. For example

```
darcs annotate --summary --match 'name record && not name overrode'
```

`--ignore-times`

Darcs optimizes its operations by keeping track of the modification times of your files. This dramatically speeds up commands such as `whatsnew` and `record` which would otherwise require reading every file in the repo and comparing it with a reference version. However, there are times when this can cause problems, such as when running a series of darcs commands from a script, in which case often a file will be modified twice in the same second, which can lead to the second modification going unnoticed. The solution to such predicaments is the `--ignore-times` option, which instructs darcs not to trust the file modification times, but instead to check each file's contents explicitly.

`--author`

Several commands need to be able to identify you. Conventionally, you provide an email address for this purpose, which can include comments, e.g. `David Roundy <droundy@abridg`
The easiest way to do this is to define an environment variable `EMAIL` or `DARCS_EMAIL` (with the latter overriding the former). You can also override this using the `--author` flag to any command. Alternatively, you could set your email address on a per-repository basis using the "defaults" mechanism for "ALL" commands, as described in Appendix B. Or, you could specify the author on a per-repository basis using the `_darcs/prefs/author` file as described in section 4.1.

Also, a global author file can be created in your home directory with the name `.darcs/author`. This file overrides the contents of the environment variables, but a repository-specific author file overrides the global author file.

`--dont-compress, --compress`

By default, darcs commands that write patches to disk will compress the patch files. If you don't want this, you can choose the `--dont-compress` option, which causes darcs not to compress the patch file.

`--gui`

Certain commands may have an optional graphical user interface. If such commands are supported, you can activate the graphical user interface by calling darcs with the `--gui` flag.

NOTE: The GUI is not currently functional, but is expected to re-appear in a future release.

`--dry-run`

The `--dry-run` option will cause darcs not to actually take the specified action, but only print what would have happened. Not all commands accept `--dry-run`, but those that do should accept the `--summary` option.

`--summary, --no-summary`

The `--summary` option shows a summary of the patches that would have been pulled/pushed/whatever. The format is similar to the output format of `cvs update` and looks like this:

```
A  ./added_but_not_recorded.c
A! ./added_but_not_recorded_conflicts.c
a  ./would_be_added_if_look_for_adds_option_was_used.h

M  ./modified.t -1 +1
M! ./modified_conflicts.t -1 +1

R  ./removed_but_not_recorded.c
R! ./removed_but_not_recorded_conflicts.c
```

You can probably guess what the flags mean from the clever file names.

`A` is for files that have been added but not recorded yet.

`a` is for files found using the `--look-for-adds` option available for `whatsnew` and `record`. They have not been added yet, but would be added automatically if `--look-for-adds` were used with the next `record` command.

`M` is for files that have been modified in the working directory but not recorded yet. The number of added and subtracted lines is also shown.

`R` is for files that have been removed, but the removal is not recorded yet.

An exclamation mark appears next to any option that has a conflict.

Resolution of conflicts To resolve conflicts using an external tool, you need to specify a command to use, e.g.

```
--external-merge 'opendiff %1 %2 -ancestor %a -merge %o'.
```

The `%1` and `%2` are replaced with the two versions to be merged, `%a` is replaced with the common ancestor of the two versions. Most importantly, `%o` is replaced with the name of the output file that darcs will require to be created holding the merged version. The above example works with the FileMerge.app tool that comes with Apple's developer tools. To use xxdiff, you would use

```
--external-merge 'xxdiff -m -O -M %o %1 %a %2'
```

To use `kdiff3`, you can use

```
--external-merge 'kdiff3 --output %o %a %1 %2'
```

Note that the command is split into space-separated words and the first one is `exec`ed with the rest as arguments—it is not a shell command. Also the substitution of the `%` escapes is only done on complete words. This means that to use Emacs' Ediff package for merging, for example, you need a helper script as follows; call it `emerge3`, say:

```
#! /bin/sh
# External merge command for darcs, using Emacs Ediff, via server if possible.
# It needs args %1 %2 %a %o, i.e. the external merge command is, say,
# `emerge3 %1 %2 %a %o'.
test $# -eq 4 || exit 1
form="(ediff-merge-files-with-ancestor"
while test $# -gt 0; do
    count=$count.
    if [ $count = .... ]; then
        form=$form\ nil          # Lisp STARTUP-HOOKS arg
    fi
    case $1 in                   # Worry about quoting -- escape " and \
        *[\"\\]* ) form=$form\ \"$(echo $1 | sed -e's/["\\]/\\\0/g')\" ;;
        *) form=$form\ \"$1\" ;;
    esac
    shift
done
form=$form')'
( emacsclient --eval "$form" || # Emacs 22 server
   gnudoit "$form" ||            # XEmacs/Emacs 21 server
   emacs --eval "$form" ||       # Relatively slow to start up
   xemacs -eval "$form"          # Horribly slow to start up
 ) 2>/dev/null
```

It would be invoked like:

```
--external-merge 'emerge3 %1 %2 %a %o'
```

If you figure out how to use darcs with another merge tool, please let me know what flags you used so I can mention it here.

Note that if you do use an external merge tool, most likely you will want to add to your defaults file (`_darcs/prefs/defaults` or `~/.darcs/prefs`, see 4.1) a line such as

```
ALL external-merge kdiff3 --output %o %a %1 %2
```

Note that the defaults file does not want quotes around the command.

`--posthook=COMMAND, --no-posthook`

To provide a command that should be run whenever a darcs command completes successfully, use `--posthook` to specify the command. This is useful for people who want to have a command run whenever a patch is applied. Using `--no-posthook` will disable running the command.

`--prompt-posthook, --run-posthook`

These options control prompting before running the posthook. Use `--prompt-posthook` to force prompting before running the posthook command. For security reasons, this is the default. When defining a posthook for apply, you will need to use `--run-posthook` or else you will get an error, because the subprocess which runs the apply command cannot prompt the user.

6.2 Options apart from darcs commands

`--help, --extended-help`

Calling darcs with just `--help` as an argument gives a brief summary of what commands are available. The `--extended-help` option gives a more technical summary of what the commands actually *do*.

`--version, --exact-version`

Calling darcs with the flag `--version` tells you the version of darcs you are using. Calling darcs with the flag `--exact-version` gives the precise version of darcs, even if that version doesn't correspond to a released version number. This is helpful with bug reports, especially when running with a "latest" version of darcs.

```
--commands
```

Similarly calling darcs with only `--commands` gives a simple list of available commands. This latter arrangement is primarily intended for the use of command-line autocompletion facilities, as are available in bash.

6.3 Creating repositories

6.3.1 darcs initialize

`Usage: darcs initialize [OPTION]...`

Options:

`--plain-pristine-tree`	Use a plain pristine tree [DEFAULT]
`--no-pristine-tree`	Use no pristine tree

Initialize a new source tree as a darcs repository.

Generally you will only call initialize once for each project you work on, and calling it is just about the first thing you do. Just make sure you are in the main directory of the project, and initialize will set up all the directories and files darcs needs in order to start keeping track of revisions for your project.

The `initialize` command actually follows a very simple procedure. It creates the directories `_darcs`, `_darcs/current` (or `_darcs/pristine`) and `_darcs/patches`, and then creates an empty file, `_darcs/inventory`. However, it is strongly recommended that you use `darcs initialize` to do this, as this procedure may change in a future version of darcs.

```
--no-pristine-tree
```

In order to save disk space, you can use `initialize` with the `--no-pristine-tree` flag to create a repository with no pristine tree. Please see Section 3.6 for more information.

6.3.2 darcs get

`Usage: darcs get [OPTION]... <REPOSITORY> [<DIRECTORY>]`

Options:

	`--repo-name DIRECTORY`	path of output directory
	`--partial`	get partial repository using checkpoint
	`--complete`	get a complete copy of the repository
	`--to-match PATTERN`	select changes up to a patch matching PATTERN
	`--to-patch REGEXP`	select changes up to a patch matching REGEXP
	`--tag REGEXP`	select tag matching REGEXP
	`--context FILENAME`	version specified by the context in FILENAME
`-v`	`--verbose`	give verbose output
`-q`	`--quiet`	suppress informational output
	`--standard-verbosity`	neither verbose nor quiet output
	`--set-default`	set default repository [DEFAULT]
	`--no-set-default`	don't set default repository
	`--set-scripts-executable`	make scripts executable
	`--dont-set-scripts-executable`	don't make scripts executable
	`--plain-pristine-tree`	Use a plain pristine tree [DEFAULT]
	`--no-pristine-tree`	Use no pristine tree

If the remote repo and the current directory are in the same filesystem and that filesystem supports hard links, get will create hard links for the patch files, which means that the additional storage space needed will be minimal. This is *very* good for your disk usage (and for the speed of running get), so if you want multiple copies of a repository, I strongly recommend first running `darcs get` to get yourself one copy, and then running `darcs get` on that copy to make any more you like. The only catch is that the first time you run `darcs push` or `darcs pull` from any of these second copies, by default they will access your first copy—which may not be what you want.

You may specify the name of the repository created by providing a second argument to get, which is a directory name.

`--context, --tag, --to-patch, --to-match`

If you want to get a specific version of a repository, you have a few options. You can either use the `--tag`, `--to-patch` or `--to-match` options, or you can use the `--context=FILENAME` option, which specifies a file containing a context generated with `darcs changes --context`. This allows you (for example) to include in your compiled program an option to output the precise version of the repository from which it was generated, and then perhaps ask users to include this information in bug reports.

Note that when specifying `--to-patch` or `--to-match`, you may get a version of your code that has never before been seen, if the patches have gotten themselves reordered. If you ever want to be able to precisely reproduce a given version, you need either to tag it or create a context file.

`--partial`

Only get the patches since the last checkpoint. This will save time, bandwidth and disk space, at the expense of losing the history before the checkpoint.

```
--no-pristine-tree
```

In order to save disk space, you can use `get` with the `--no-pristine-tree` flag to create a repository with no pristine tree. Please see Section 3.6 for more information.

6.3.3 darcs put

`Usage: darcs put [OPTION]... <NEW REPOSITORY>`

Options:

`-v`	`--verbose`	give verbose output
`-q`	`--quiet`	suppress informational output
	`--standard-verbosity`	neither verbose nor quiet output
	`--repodir DIRECTORY`	specify the repository directory in which to run
	`--to-match PATTERN`	select changes up to a patch matching PATTERN
	`--to-patch REGEXP`	select changes up to a patch matching REGEXP
	`--tag REGEXP`	select tag matching REGEXP
	`--context FILENAME`	version specified by the context in FILENAME
	`--apply-as USERNAME`	apply patch as another user using sudo
	`--apply-as-myself`	don't use sudo to apply as another user [DEFAULT]
	`--plain-pristine-tree`	Use a plain pristine tree [DEFAULT]
	`--no-pristine-tree`	Use no pristine tree

Put is the opposite of get. Put copies the content of the current repository and puts it in a newly created repository.

WARNING: Put is far less optimized than get, especially for local repositories. We recommend avoiding use of put except for small repositories.

Put is used when you already have a repository and want to make a copy of it. A typical use-case is when you want to branch your project.

Put works by first initializing a repository. If the new repository is not on the local file system then darcs will login to the remote host and run `darcs init` there. After the new repository is created all selected patches will be pushed just as with the command `push`.

```
--apply-as
```

If you give the `--apply-as` flag, darcs will use sudo to apply the changes as a different user. This can be useful if you want to set up a system where several users can modify the same repository, but you don't want to allow them full write access. This isn't secure against skilled malicious attackers, but at least can protect your repository from clumsy, inept or lazy users.

`--context, --tag, --to-patch, --to-match`

If you want to put a specific version of a repository, you have a few options. You can either use the `--tag`, `--to-patch` or `--to-match` options, or you can use the `--context=FILENAME` option, which specifies a file containing a context generated with `darcs changes --context`. This allows you (for example) to include in your compiled program an option to output the precise version of the repository from which it was generated, and then perhaps ask users to include this information in bug reports.

Note that when specifying `--to-patch` or `--to-match`, you may get a version of your code that has never before been seen, if the patches have gotten themselves reordered. If you ever want to be able to precisely reproduce a given version, you need either to tag it or create a context file.

6.4 Modifying the contents of a repo

6.4.1 darcs add

`Usage: darcs add [OPTION]... <FILE or DIRECTORY> ...`

Options:

	`--boring`	don't skip boring files
	`--case-ok`	don't refuse to add files differing only in case
`-r`	`--recursive`	add contents of subdirectories
	`--not-recursive`	don't add contents of subdirectories
	`--date-trick`	add files with date appended to avoid conflict. [EXPERIMENTAL]
	`--no-date-trick`	don't use experimental date appending trick. [DEFAULT]
`-v`	`--verbose`	give verbose output
`-q`	`--quiet`	suppress informational output
	`--standard-verbosity`	neither verbose nor quiet output
	`--repodir DIRECTORY`	specify the repository directory in which to run
	`--dry-run`	don't actually take the action

Add needs to be called whenever you add a new file or directory to your project. Of course, it also needs to be called when you first create the project, to let darcs know which files should be kept track of.

Darcs will refuse to add a file or directory that differs from an existing one only in case. This is because the HFS+ file system used on MacOS treats such files as being one and the same.

You can not add symbolic links to darcs. If you try to do that, darcs will refuse and print an error message. Perhaps you want to make symbolic links *to* the files in darcs instead?

`--boring`

By default darcs will ignore all files that match any of the boring patterns. If you want to add such a file anyway you must use the `--boring` option.

`--date-trick`

The `--date-trick` option allows you to enable an experimental trick to make add conflicts, in which two users each add a file or directory with the same name, less problematic. While this trick is completely safe, it is not clear to what extent it is beneficial.

6.4.2 darcs remove

`Usage: darcs remove [OPTION]... <FILE or DIRECTORY> ...`

Options:

`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--repodir DIRECTORY`	specify the repository directory in which to run

Remove should be called when you want to remove a file from your project, but don't actually want to delete the file. Otherwise just delete the file or directory, and darcs will notice that it has been removed. Be aware that the file WILL be deleted from any other copy of the repo to which you later apply the patch.

6.4.3 darcs mv

`Usage: darcs mv [OPTION]... [FILE or DIRECTORY]...`

Options:

	`--case-ok`	don't refuse to add files differing only in case
`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--repodir DIRECTORY`	specify the repository directory in which to run

Darcs mv needs to be called whenever you want to move files or directories. Unlike remove, mv actually performs the move itself in your working copy. This is why "mv" isn't called "move", since it is really almost equivalent to the unix command "mv". I could add an equivalent command named "move" for those who like vowels.

`--case-ok`

Darcs mv will by default refuse to rename a file if there already exists a file having the same name apart from case. This is because doing so could create a repository that could not be used on file systems that are case insensitive (such as Apples HFS+). You can override this by with the flag `--case-ok`.

6.4.4 darcs replace

`Usage: darcs replace [OPTION]... <OLD> <NEW> <FILE> ...`

Options:

	`--token-chars "[CHARS]"`	define token to contain these characters
	`--force`	proceed with replace even if 'new' token already exists
	`--no-force`	don't force the replace if it looks scary
`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output

Replace allows you to change a specified token wherever it occurs in the specified files. The replace is encoded in a special patch and will merge as expected with other patches. Tokens here are defined by a regexp specifying the characters which are allowed. By default a token corresponds to a C identifier.

The default regexp is `[A-Za-z_0-9]`), and if one of your tokens contains a '`-`' or '`.`', you will then (by default) get the "filename" regexp, which is `[A-Za-z_0-9\-\.]`.

`--token-chars`

If you prefer to choose a different set of characters to define your token (perhaps because you are programming in some other language), you may do so with the `--token-chars` option. You may prefer to define tokens in terms of delimiting characters instead of allowed characters using a flag such as `--token-chars '[^ \n\t]'`, which would define a token as being white-space delimited.

If you do choose a non-default token definition, I recommend using `_darcs/prefs/defaults` to always specify the same `--token-chars`, since your replace patches will be better behaved (in terms of commutation and merges) if they have tokens defined in the same way.

When using darcs replace, the "new" token may not already appear in the file—if that is the case, the replace change would not be invertible. This limitation holds only on the already-recorded version of the file.

There is a potentially confusing difference, however, when a replace is used to make another replace possible:

```
% darcs replace newtoken aaack ./foo.c
% darcs replace oldtoken newtoken ./foo.c
% darcs record
```

will be valid, even if `newtoken` and `oldtoken` are both present in the recorded version of foo.c, while the sequence

```
% [manually edit foo.c replacing newtoken with aaack]
% darcs replace oldtoken newtoken ./foo.c
```

will fail because "newtoken" still exists in the recorded version of `foo.c`. The reason for the difference is that when recording, a "replace" patch always is recorded *before* any manual changes, which is usually what you want, since often you will introduce new occurrences of the "newtoken" in your manual changes. In contrast, multiple "replace" changes are recorded in the order in which they were made.

6.5 Working with changes

6.5.1 darcs record

`Usage: darcs record [OPTION]... [FILE or DIRECTORY]...`

Options:

`-m`	`--patch-name PATCHNAME`	name of patch
`-A`	`--author EMAIL`	specify author id
	`--logfile FILE`	give patch name and comment in file
	`--delete-logfile`	delete the logfile when done
`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--no-test`	don't run the test script
	`--test`	run the test script
	`--leave-test-directory`	don't remove the test directory
	`--remove-test-directory`	remove the test directory
	`--compress`	create compressed patches
	`--dont-compress`	don't create compressed patches
`-a`	`--all`	answer yes to all patches
	`--pipe`	expect to receive input from a pipe
	`--interactive`	prompt user interactively
	`--ask-deps`	ask for extra dependencies
	`--no-ask-deps`	don't ask for extra dependencies
	`--edit-long-comment`	Edit the long comment by default
	`--skip-long-comment`	Don't give a long comment
	`--prompt-long-comment`	Prompt for whether to edit the long comment
	`--ignore-times`	don't trust the file modification times
`-l`	`--look-for-adds`	In addition to modifications, look for files that are not boring, and thus are potentially pending addition
	`--dont-look-for-adds`	Don't look for any files or directories that could be added, and don't add them automatically
	`--repodir DIRECTORY`	specify the repository directory in which to run

If you provide one or more files or directories as additional arguments to record, you will only be prompted to changes in those files or directories. Each patch is given a name, which typically would consist of a brief description of the changes. This name is later used to describe the patch. The name must fit on one line (i.e. cannot have any embedded newlines). If you have more to say, stick it in the log. The patch is also flagged with the author of the change, taken by default from the `DARCS_EMAIL` environment variable, and if that doesn't exist, from the `EMAIL` environment variable. The date on which the patch was recorded is also included. Currently there is no provision for keeping track of when a patch enters a given repository. Finally, each changeset should have a full log (which may be empty). This log is for detailed notes which are too lengthy to fit in the name. If you answer that you do want to create a comment file, darcs will open an editor so that you can enter the comment in. The choice of editor proceeds as follows. If one of the `$DARCS_EDITOR`, `$VISUAL` or `$EDITOR` environment variables is defined, its value is used (with precedence proceeding in the order listed). If not, "vi", "emacs", "emacs -nw" and "nano" are tried in that order.

```
--logfile
```

If you wish, you may specify the patch name and log using the `--logfile` flag. If you do so, the first line of the specified file will be taken to be the patch name, and the remainder will be the "long comment". This feature can be especially handy if you have a test that fails several times on the record (thus aborting the record), so you don't have to type in the long comment multiple times. The file's contents will override the `--patch-name` option.

```
--ask-deps
```

Each patch may depend on any number of previous patches. If you choose to make your patch depend on a previous patch, that patch is required to be applied before your patch can be applied to a repo. This can be used, for example, if a piece of code requires a function to be defined, which was defined in an earlier patch.

If you want to manually define any dependencies for your patch, you can use the `--ask-deps` flag, and darcs will ask you for the patch's dependencies.

```
--no-test,  --test
```

If you configure darcs to run a test suite, darcs will run this test on the recorded repo to make sure it is valid. Darcs first creates a pristine copy of the source tree (in a temporary directory), then it runs the test, using its return value to decide if the record is valid. If it is not valid, the record will be aborted. This is a handy way to avoid making stupid mistakes like forgetting to 'darcs add' a new file. It also can be tediously slow, so there is an option (`--no-test`) to skip the test.

```
--pipe
```

If you run record with the `--pipe` option, you will be prompted for the patch name, patch date and the long comment. The long comment will extend until the end of file of stdin is reached (ctrl-D on Unixy systems). This interface is intended for scripting darcs, in particular for writing repository conversion scripts. The prompts are intended mostly as a useful guide (since scripts won't need them), to help you understand the format in which to provide the input.

```
--interactive
```

By default, `record` works interactively. Probably the only thing you need to know about using this is that you can press `?` at the prompt to be shown a list of the rest of the options and what they do. The rest should be clear from there. Here's a "screenshot" to demonstrate:

```
hunk ./hello.pl +2
+#!/usr/bin/perl
+print "Hello World!\n";
Shall I record this patch? (2/2) [ynWsfqadjk], or ? for help: ?
How to use record...
y: record this patch
n: don't record it
w: wait and decide later, defaulting to no

s: don't record the rest of the changes to this file
f: record the rest of the changes to this file

d: record selected patches
a: record all the remaining patches
q: cancel record

j: skip to next patch
k: back up to previous patch
h or ?: show this help

<Space>: accept the current default (which is capitalized)
```

What you can't see in that "screenshot" is that `darcs` will also try to use color in your terminal to make the output even easier to read.

6.5.2 darcs pull

```
Usage: darcs pull [OPTION]... [REPOSITORY]
```

Options:

	`--matches PATTERN`	select patches matching PATTERN
`-p`	`--patches REGEXP`	select patches matching REGEXP
`-t`	`--tags REGEXP`	select tags matching REGEXP
`-a`	`--all`	answer yes to all patches
	`--interactive`	prompt user interactively
	`--external-merge COMMAND`	Use external tool to merge conflicts
	`--compress`	create compressed patches
	`--dont-compress`	don't create compressed patches
	`--test`	run the test script
	`--no-test`	don't run the test script
	`--dry-run`	don't actually take the action
`-s`	`--summary`	summarize changes
	`--no-summary`	don't summarize changes
`-v`	`--verbose`	give verbose output
`-q`	`--quiet`	suppress informational output
	`--standard-verbosity`	neither verbose nor quiet output
	`--ignore-times`	don't trust the file modification times
	`--no-deps`	don't automatically fulfill dependencies
	`--set-default`	set default repository [DEFAULT]
	`--no-set-default`	don't set default repository
	`--repodir DIRECTORY`	specify the repository directory in which to run
	`--set-scripts-executable`	make scripts executable
	`--dont-set-scripts-executable`	don't make scripts executable

Pull is used to bring changes made in another repo into the current repo (that is, either the one in the current directory, or the one specified with the –repodir option). Pull allows you to bring over all or some of the patches that are in that repo but not in this one. Pull accepts an argument, which is the URL from which to pull, and when called without an argument, pull will use the repository from which you have most recently either pushed or pulled.

`--external-merge`

You can use an external interactive merge tool to resolve conflicts with the flag `--external-merge`. For more details see subsection 6.1.

`--matches, --no-deps, --patches, --tags`

The `--patches`, `--matches`, and `--tags` options can be used to select which patches to pull, as described in subsection 6.1. darcs will silently pull along any other patches upon which the selected patches depend. So `--patches bugfix` means "pull all the patches with 'bugfix' in their name, along with any patches they require." If you really only want the patches with 'bugfix' in their name, you should use the `--no-deps` option, which makes darcs pull in only the selected patches which have no dependencies (apart from other selected patches).

`--no-test, --test`

If you specify the `--test` option, pull will run the test (if a test exists) on a scratch copy of the repo contents prior to actually performing the pull. If the test fails, the pull will be aborted.

`--verbose`

Adding the `--verbose` option causes another section to appear in the output which also displays a summary of patches that you have and the remote repo lacks. Thus, the following syntax can be used to show you all the patch differences between two repos:

```
darcs pull --dry-run --verbose
```

6.5.3 darcs push

`Usage: darcs push [OPTION]... [REPOSITORY]`

Options:

`-v`	`--verbose`	give verbose output
`-q`	`--quiet`	suppress informational output
	`--standard-verbosity`	neither verbose nor quiet output
	`--matches PATTERN`	select patches matching PATTERN
`-p`	`--patches REGEXP`	select patches matching REGEXP
`-t`	`--tags REGEXP`	select tags matching REGEXP
`-a`	`--all`	answer yes to all patches
	`--interactive`	prompt user interactively
	`--apply-as USERNAME`	apply patch as another user using sudo
	`--apply-as-myself`	don't use sudo to apply as another user [DEFAULT]
	`--dry-run`	don't actually take the action
`-s`	`--summary`	summarize changes
	`--no-summary`	don't summarize changes
	`--repodir DIRECTORY`	specify the repository directory in which to run
	`--set-default`	set default repository [DEFAULT]
	`--no-set-default`	don't set default repository

Push is the opposite of pull. Push allows you to copy changes from the current repository into another repository.

For obvious reasons, you can only push to repositories to which you have write access. In addition, you can only push to repos that you access either on the local file system or with ssh. In order to apply with ssh, darcs must also be installed on the remote computer. The command invoked to run ssh may be configured by the `DARCS_SSH` environment variable (see subsection 4.2). The command invoked via ssh is always `darcs`, i.e. the darcs executable must be in the default path on the remote machine.

Push works by creating a patch bundle, and then running darcs apply in the target repository using that patch bundle. This means that the default options for *apply* in the *target* repository (such as, for example, `--test`) will affect the behavior of push. This also means that push is somewhat less efficient than pull.

When you receive an error message such as

```
bash: darcs: command not found
```

then this means that the darcs on the remote machine could not be started. Make sure that the darcs executable is called `darcs` and is found in the default path. The default path can be different in interactive and in non-interactive shells. Say

```
ssh login@remote.machine darcs
```

to try whether the remote darcs can be found, or

```
ssh login@remote.machine 'echo $PATH'
```

(note the single quotes) to check the default path.

```
--apply-as
```

If you give the `--apply-as` flag, darcs will use sudo to apply the changes as a different user. This can be useful if you want to set up a system where several users can modify the same repository, but you don't want to allow them full write access. This isn't secure against skilled malicious attackers, but at least can protect your repository from clumsy, inept or lazy users.

```
--matches, --patches, --tags
```

The `--patches`, `--matches`, and `--tags` options can be used to select which patches to push, as described in subsection 6.1. darcs will silently push along any other patches upon which the selected patches depend.

When there are conflicts, the behavior of push is determined by the default flags to `apply` in the *target* repository. Most commonly, for pushed-to repositories, you'd like to have `--dont-allow-conflicts` as a default option to apply (by default, it is already the default...). If this is the case, when there are conflicts on push, darcs will fail with an error message. You can then resolve by pulling the conflicting patch, recording a resolution and then pushing the resolution together with the conflicting patch.

Darcs does not have an explicit way to tell you which patch conflicted, only the file name. You may want to pull all the patches from the remote repo just to be sure. If you don't want to do this in your working directory, you can create another darcs working directory for this purpose.

If you want, you could set the target repo to use `--allow-conflicts`. In this case conflicting patches will be applied, but the conflicts will not be marked in the working directory.

If, on the other hand, you have `--mark-conflicts` specified as a default flag for apply in the target repository, when there is a conflict, it will be marked in the working directory of the target repository. In this case, you should resolve the conflict in the target repository itself.

6.5.4 darcs send

```
Usage: darcs send [OPTION]... [REPOSITORY]
```

Options:

`-v`	`--verbose`	give verbose output
`-q`	`--quiet`	suppress informational output
	`--standard-verbosity`	neither verbose nor quiet output
	`--matches PATTERN`	select patches matching PATTERN
`-p`	`--patches REGEXP`	select patches matching REGEXP
`-t`	`--tags REGEXP`	select tags matching REGEXP
`-a`	`--all`	answer yes to all patches
	`--interactive`	prompt user interactively
	`--from EMAIL`	specify email address
`-A`	`--author EMAIL`	specify author id
	`--to EMAIL`	specify destination email
	`--cc EMAIL`	mail results to additional EMAIL(s). Requires –reply
`-o`	`--output FILE`	specify output filename
	`--sign`	sign the patch with your gpg key
	`--sign-as KEYID`	sign the patch with a given keyid
	`--sign-ssl IDFILE`	sign the patch using openssl with a given private key
	`--dont-sign`	do not sign the patch
`-u`	`--unified`	output patch in a darcs-specific format similar to diff -u
	`--dry-run`	don't actually take the action
`-s`	`--summary`	summarize changes
	`--no-summary`	don't summarize changes
	`--context FILENAME`	send to context stored in FILENAME
	`--edit-description`	edit the patch bundle description
	`--dont-edit-description`	don't edit the patch bundle description
	`--set-default`	set default repository [DEFAULT]
	`--no-set-default`	don't set default repository
	`--repodir DIRECTORY`	specify the repository directory in which to run
	`--sendmail-command COMMAND`	specify sendmail command

Send is used to prepare a bundle of patches that can be applied to a target repository. Send accepts the URL of the repository as an argument. When called without an argument, send will use the most recent repository that was either pushed to, pulled from or sent to. By default, the patch bundle is sent by email, although you may save it to a file.

`--unified`

If you want to create patches having context, you can use the `--unified` option, which create output vaguely reminiscent of `diff -u`. This format is still darcs-specific and should not be expected to apply cleanly by `patch`.

`--output, --to, --cc`

The `--output` and `--to` flags determine what darcs does with the patch bundle after creating it. If you provide an `--output` argument, the patch bundle is saved to that file. If you give one or more `--to` arguments, the bundle of patches is emailed to those addresses.

If you don't provide either a `--output` or a `--to` flag, darcs will look at the contents of the `_darcs/prefs/email` file in the target repository (if it exists), and send the patch by email to that address. In this case, you may use the `--cc` option to specify additional recipients without overriding the default repository email address.

If there is no email address associated with the repository, darcs will prompt you for an email address.

```
--matches, --patches, --tags
```

The `--patches`, `--matches`, and `--tags` options can be used to select which patches to send, as described in subsection 6.1. darcs will silently send along any other patches upon which the selected patches depend.

```
--edit-description
```

If you want to include a description or explanation along with the bundle of patches, you need to specify the `--edit-description` flag, which will cause darcs to open up an editor with which you can compose an email to go along with your patches.

```
--sendmail-command
```

If you want to use a command different from the default one for sending mails, you need to specify a commandline with the `--sendmail-command` option. The commandline can contain some format specifiers which are replaced by the actual values. Accepted format specifiers are `%s` for subject, `%t` for to, `%c` for cc, `%b` for the body of the mail, `%f` for from, `%a` for the patch bundle and the same specifiers in uppercase for the urlencoded values. Additionally you can add `%<` to the end of the commandline if the command expects the complete mail on standard input. E.g. the commandlines for evolution and msmtp look like this:

```
evolution "mailto:%T?subject=%S&attach=%A&cc=%C&body=%B"
msmtp %t %<
```

6.5.5 darcs apply

```
Usage: darcs apply [OPTION]... <PATCHFILE>
```

Options:

	`--verify PUBRING`	verify that the patch was signed by a key in PUBRING
	`--verify-ssl KEYS`	verify using openSSL with authorized keys from file 'KEYS'
	`--no-verify`	don't verify patch signature
	`--reply FROM`	reply to email-based patch using FROM address
	`--cc EMAIL`	mail results to additional EMAIL(s). Requires –reply
`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--ignore-times`	don't trust the file modification times
	`--compress`	create compressed patches
	`--dont-compress`	don't create compressed patches
`-a`	`--all`	answer yes to all patches
	`--interactive`	prompt user interactively
	`--mark-conflicts`	mark conflicts
	`--allow-conflicts`	allow conflicts, but don't mark them
	`--external-merge COMMAND`	Use external tool to merge conflicts
	`--no-resolve-conflicts`	equivalent to –dont-allow-conflicts, for backwards compatibility
	`--dont-allow-conflicts`	fail on patches that create conflicts [DEFAULT]
	`--no-test`	don't run the test script
	`--test`	run the test script
	`--happy-forwarding`	forward unsigned messages without extra header
	`--leave-test-directory`	don't remove the test directory
	`--remove-test-directory`	remove the test directory
	`--repodir DIRECTORY`	specify the repository directory in which to run
	`--sendmail-command COMMAND`	specify sendmail command
	`--set-scripts-executable`	make scripts executable
	`--dont-set-scripts-executable`	don't make scripts executable

Apply is used to apply a bundle of patches to this repository. Such a bundle may be created using send.

Darcs apply accepts a single argument, which is the name of the patch file to be applied. If you omit this argument, the patch is read from standard input.[4] This allows you to use apply with a pipe from your email program, for example.

`--verify`

If you specify the `--verify PUBRING` option, darcs will check that the patch was GPG-signed by a key which is in `PUBRING`, and will refuse to apply the patch otherwise.

[4]One caveat: don't name your patch file "magic darcs standard input", or darcs will read from standard input instead!

```
--cc, --reply
```

If you give the `--reply FROM` option to `darcs apply`, it will send the results of the application to the sender of the patch. This only works if the patch is in the form of an email with its headers intact, so that darcs can actually know the origin of the patch. The reply email will indicate whether or not the patch was successfully applied. The `FROM` flag is the email address that will be used as the "from" address when replying. If the darcs apply is being done automatically, it is important that this address not be the same as the address at which the patch was received, in order to avoid automatic email loops.

If you want to also send the apply email to another address (for example, to create something like a "commits" mailing list), you can use the `--cc` option to specify additional recipients. Note that the `--cc` option *requires* the `--reply` option, which provides the "From" address.

The `--reply` feature of apply is intended primarily for two uses. When used by itself, it is handy for when you want to apply patches sent to you by other developers so that they will know when their patch has been applied. For example, in my `.muttrc` (the config file for my mailer) I have:

```
macro pager A "<pipe-entry>darcs apply --verbose \
        --reply droundy@abridgegame.org --repodir ~/darcs
```

which allows me to apply a patch to darcs directly from my mailer, with the originator of that patch being sent a confirmation when the patch is successfully applied. NOTE: For some reason mutt seems to set the umask such that patches created with the above macro are not world-readable. I'm not sure why this is, but use it with care.

When used in combination with the `--verify` option, the `--reply` option allows for a nice pushable repository. When these two options are used together, any patches that don't pass the verify will be forwarded to the `FROM` address of the `--reply` option. This allows you to set up a repository so that anyone who is authorized can push to it and have it automatically applied, but if a stranger pushes to it, the patch will be forwarded to you. Please (for your own sake!) be certain that the `--reply FROM` address is different from the one used to send patches to a pushable repository, since otherwise an unsigned patch will be forwarded to the repository in an infinite loop.

If you use '`darcs apply --verify PUBRING --reply`' to create a pushable repo by applying patches automatically as they are received by email, you will also want to use the `--dont-allow-conflicts` option.

```
--dont-allow-conflicts
```

The `--dont-allow-conflicts` flag causes apply to fail when applying a patch would cause conflicts. This flag is recommended on repositories which will be pushed to or sent to.

`--allow-conflicts`

`--allow-conflicts` will allow conflicts, but will keep the local and recorded versions in sync on the repo. This means the conflict will exist in both locations until it is resolved.

`--mark-conflicts`

`--mark-conflicts` will add conflict markers to illustrate the the conflict.

`--external-merge`

You can use an external interactive merge tool to resolve conflicts with the flag `--external-merge`. For more details see subsection 6.1.

`--all, --gui, --interactive`

If you provide the `--interactive` or `--gui` flag, darcs will ask you for each change in the patch bundle whether or not you wish to apply that change. The opposite is the `--all` flag, which can be used to override an `interactive` or `gui` which might be set in your "defaults" file.

NOTE: The GUI is not currently functional, but is expected to re-appear in a future release.

`--sendmail-command`

If you want to use a command different from the default one for sending mail, you need to specify a command line with the `--sendmail-command` option. The command line can contain the format specifier `%t` for to and you can add `%<` to the end of the commandline if the command expects the complete mail on standard input. E.g. the commandline for msmtp looks like this:

```
msmtp %t %<
```

`--no-test, --test`

If you specify the `--test` option, apply will run the test (if a test exists) prior to applying the patch. If the test fails, the patch is not applied. In this case, if the `--reply` option was used, the results of the test are sent in the reply email. You can also specify the `--no-test` option, which will override the `--test` option, and prevent the test from being run. This is helpful when setting up a pushable repository, to keep users from running code.

6.6 Seeing what you've done

6.6.1 darcs whatsnew

```
Usage: darcs whatsnew [OPTION]... [FILE or DIRECTORY]...
```

Options:

`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
`-s`	`--summary`	summarize changes
	`--no-summary`	don't summarize changes
`-u`	`--unified`	output patch in a darcs-specific format similar to diff -u
	`--ignore-times`	don't trust the file modification times
`-l`	`--look-for-adds`	In addition to modifications, look for files that are not boring, and thus are potentially pending addition
	`--dont-look-for-adds`	Don't look for any files or directories that could be added, and don't add them automatically
	`--boring`	don't skip boring files
	`--repodir DIRECTORY`	specify the repository directory in which to run

Display unrecorded changes in the working copy. whatsnew gives you a view of what changes you've made in your working copy that haven't yet been recorded. The changes are displayed in darcs patch format. Note that –look-for-adds implies –summary usage. `darcs whatsnew` will return a non-zero value if there are no changes, which can be useful if you just want to see in a script if anything has been modified. If you want to see some context around your changes, you can use the `-u` option, to get output similar to the unidiff format.

If you give one or more file or directory names as an argument to `whatsnew`, darcs will output only changes to those files or to files in those directories.

6.6.2 darcs changes

```
Usage: darcs changes [OPTION]... [FILE or DIRECTORY]...
```

Options:

	`--to-match PATTERN`	select changes up to a patch matching PATTERN
	`--to-patch REGEXP`	select changes up to a patch matching REGEXP
	`--to-tag REGEXP`	select changes up to a tag matching REGEXP
	`--from-match PATTERN`	select changes starting with a patch matching PATTERN
	`--from-patch REGEXP`	select changes starting with a patch matching REGEXP
	`--from-tag REGEXP`	select changes starting with a tag matching REGEXP
	`--last NUMBER`	select the last NUMBER patches
	`--matches PATTERN`	select patches matching PATTERN
`-p`	`--patches REGEXP`	select patches matching REGEXP
`-t`	`--tags REGEXP`	select tags matching REGEXP
	`--context`	give output suitable for get –context
	`--xml-output`	generate XML formatted output
	`--human-readable`	give human-readable output
`-s`	`--summary`	summarize changes
	`--no-summary`	don't summarize changes
`-v`	`--verbose`	give verbose output
`-q`	`--quiet`	suppress informational output
	`--standard-verbosity`	neither verbose nor quiet output
	`--reverse`	show changes in reverse order
	`--repo URL`	specify the repository URL

Changes gives a changelog-style summary of the repo history, including options for altering how the patches are selected and displayed.

When given one or more files or directories as an argument, changes lists only those patches which affect those files or the contents of those directories or, of course, the directories themselves. This includes changes that happened to files before they were moved or renamed.

`--from-match, --from-patch, --from-tag`

If changes is given a `--from-patch`, `--from-match`, or `--from-tag` option, it outputs only those changes since that tag or patch.

Without any options to limit the scope of the changes, history will be displayed going back as far as possible.

`--context, --human-readable, --xml-output`

When given the `--context` flag, darcs changes outputs sufficient information to allow the current state of the repository to be recreated at a later date. This information should generally be piped to a file, and then can be used later in conjunction with `darcs get --context` to recreate the current version. Note that while the `--context` flag may be used in conjunction with `--xml-output` or `--human-readable`, in neither case will darcs get be able to read the output. On the other hand, sufficient information *will* be output for a knowledgeable human to recreate the current state of the repository.

6.6.3 darcs query manifest

`Usage: darcs query manifest [OPTION]...`

Options:

	`--files`	include files in output [DEFAULT]
	`--no-files`	do not include files in output
	`--directories`	include directories in output
	`--no-directories`	do not include directories in output [DEFAULT]
	`--pending`	reflect pending patches in output [DEFAULT]
	`--no-pending`	only included recorded patches in output
`-0`	`--null`	separate file names by NUL characters
	`--repodir DIRECTORY`	specify the repository directory in which to run

The manifest command lists the version-controlled files in the working copy.

By default (and if the `--pending` option is specified), the effect of pending patches on the repository is taken into account. In other words, if you add a file using `darcs add`, it immediately appears in the output of `query manifest`, even if it is not yet recorded. If you specify the `--no-pending` option, `query manifest` will only list recorded files (and directories).

The `--files` and `--directories` options control whether files and directories are included in the output. The `--no-files` and `--no-directories` options have the reverse effect. The default is to include files, but not directories.

If you specify the `--null` option, the file names are written to standard output in unescaped form, and separated by ASCII NUL bytes. This format is suitable for further automatic processing (for example, using `xargs -0`).

6.7 More advanced commands

6.7.1 darcs tag

`Usage: darcs tag [OPTION]... [TAGNAME]`

Options:

`-m`	`--patch-name PATCHNAME`	name of patch
`-A`	`--author EMAIL`	specify author id
	`--checkpoint`	create a checkpoint file
	`--compress`	create compressed patches
	`--dont-compress`	don't create compressed patches
	`--pipe`	expect to receive input from a pipe
	`--interactive`	prompt user interactively
`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output

Tag is used to name a version of this repository (i.e. the whole tree). Tag differs from record in that it doesn't record any new changes, and it always depends on all patches residing in the repository when it is tagged. This means that one can later reproduce this version of the repository by calling, for example:

```
% darcs get --tag "darcs 3.14" REPOLOCATION
```

Each tagged version has a version name. The version is also flagged with the person who tagged it (taken by default from the 'DARCS_EMAIL' or 'EMAIL' environment variable). The date is also included in the version information.

A tagged version automatically depends on all patches in the repo. This allows you to later reproduce precisely that version. The tag does this by depending on all patches in the repo, except for those which are depended upon by other tags already in the repo. In the common case of a sequential series of tags, this means that the tag depends on all patches since the last tag, plus that tag itself.

`--checkpoint`

The `--checkpoint` option allows the tag be used later with the `--partial` flag to `get` or `check`.

A partial repository only contains patches from after the checkpoint. A partial repository works just like a normal repository, but any command that needs to look at the contents of a missing patch will complain and abort.

`--pipe`

If you run tag with the `--pipe` option, you will be prompted for the tag name and date. This interface is intended for scripting darcs, in particular for writing repository conversion scripts. The prompts are intended mostly as useful guide (since scripts won't need them), to help you understand the format in which to provide the input. Here's an example of what the `--pipe` prompts looks like:

```
 What is the date? Mon Nov 15 13:38:01 EST 2004
 Who is the author? David Roundy
 What is the version name? 3.0
 Finished tagging patch 'TAG 3.0'
```

Using `tag` creates an entry in the repo history just like `record`. It will show up with `darcs changes` appearing in the format:

```
tagged My Tag Name
```

Because the word 'tagged' is always prepended to the tag name, you can search for tag names by simply passing the output of `darcs changes` through `grep`:

```
darcs changes | grep tagged
```

The above example would display all the tag names in use in the repo.

6.7.2 darcs setpref

`Usage: darcs setpref [OPTION]... <PREF> <VALUE>`
Options:

Usage example:

```
% darcs setpref test "echo I am not really testing anything."
```

Setpref allows you to set a preference value in a way that will propagate to other repositories.

Valid preferences are: test predist boringfile binariesfile. If you just want to set the pref value in your repository only, you can just edit "`_darcs/prefs/prefs`". Changes you make in that file will be preserved.

The "`_darcs/prefs/prefs`" holds the only preferences information that can propagate between repositories by pushes and pulls, and the only way this happens is when the setprefs command is used. Note that although prefs settings are included in patches, they are *not* fully version controlled. In particular, depending on the order in which a series of merges is performed, you may end up with a different final prefs configuration. In practice I don't expect this to be a problem, as the prefs usually won't be changed very often.

The following values are valid preferences options which can be configured using setpref:

- "test" — the command to run as a test script.
- "predist" — a command to run prior to tarring up a distribution tarball. Typically this would consist of autoconf and/or automake.
- "boringfile" — the name of a file to read instead of the "boring" prefs file.
- "binariesfile" — the name of a file to read instead of the "binaries" prefs file.

6.7.3 darcs check

`Usage: darcs check [OPTION]...`

Options:

	`--complete`	check the entire repository
	`--partial`	check patches since latest checkpoint
`-v`	`--verbose`	give verbose output
`-q`	`--quiet`	suppress informational output
	`--standard-verbosity`	neither verbose nor quiet output
	`--no-test`	don't run the test script
	`--test`	run the test script
	`--leave-test-directory`	don't remove the test directory
	`--remove-test-directory`	remove the test directory
	`--repodir DIRECTORY`	specify the repository directory in which to run

Check the repository for consistency. Check verifies that the patches stored in the repository, when successively applied to an empty tree, properly recreate the stored pristine tree.

`--complete, --partial`

If you have a checkpoint of the repository (as is the case if you got the repo originally using `darcs get --partial`), by default darcs check will only verify the contents since the most recent checkpoint. You can change this behavior using the `--complete` flag.

If you like, you can configure your repository to be able to run a test suite of some sort. You can do this by using “setpref” to set the “test” value to be a command to run, e.g.

```
% darcs setpref test "sh configure && make && make test"
```

Or, if you want to define a test specific to one copy of the repository, you could do this by editing the file `_darcs/prefs/prefs`.

`--leave-test-directory, --remove-test-directory`

Normally darcs deletes the directory in which the test was run afterwards. Sometimes (especially when the test fails) you'd prefer to be able to be able to examine the test directory after the test is run. You can do this by specifying the `--leave-test-directory` flag. Alas, there is no way to make darcs leave the test directory only if the test fails. The opposite of `--leave-test-directory` is `--remove-test-directory`, which could come in handy if you choose to make `--leave-test-directory` the default (see section 4.1).

`--no-test`

If you just want to check the consistency of your repository without running the test, you can call darcs check with the `--no-test` option.

6.7.4 darcs optimize

`Usage: darcs optimize [OPTION]...`

Options:

	`--checkpoint`	create a checkpoint file
	`--compress`	create compressed patches
	`--dont-compress`	don't create compressed patches
	`--uncompress`	uncompress patches
`-t`	`--tag TAGNAME`	name of version to checkpoint
`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--modernize-patches`	rewrite all patches in current darcs format
	`--reorder-patches`	reorder the patches in the repository
	`--sibling URL`	specify a sibling directory
	`--relink`	relink random internal data to a sibling
	`--relink-pristine`	relink pristine tree (not recommended)

Optimize can help to improve the performance of your repository in a number of cases.

Optimize always writes out a fresh copy of the inventory that minimizes the amount of inventory that need be downloaded when people pull from the repo.

Specifically, it breaks up the inventory on the most recent tag. This speeds up most commands when run remotely, both because a smaller file needs to be transfered (only the most recent inventory). It also gives a guarantee that all the patches prior to a given tag are included in that tag, so less commutation and history traversal is needed. This latter issue can become very important in large repositories.

`--checkpoint, --tag`

If you use the `--checkpoint` option, optimize creates a checkpoint patch for a tag. You can specify the tag with the `--tag` option, or just let darcs choose the most recent tag. Note that optimize `--checkpoint` will fail when used on a "partial" repository. Also, the tag that is to be checkpointed must not be preceded by any patches that are not included in that tag. If that is the case, no checkpointing is done.

The created checkpoint is used by the `--partial` flag to `get` and `check`. This allows for users to retrieve a working repository with limited history with a savings of disk space and bandwidth.

`--compress, --dont-compress, --uncompress`

Some compression options are available, and are independent of the `--checkpoint` option.

By default the patches in the repository are compressed. These use less disk space, which translates into less bandwidth if the repository is accessed remotely. Note that patches will always have the ".gz" extension whether they are compressed or not.

You may want to uncompress the patches when you've got enough disk space but are running out of physical memory.

If you give the `--compress` option, optimize will compress all the patches in the repository. Similarly, if you give the `--uncompress`, optimize will decompress all the patches in the repository. `--dont-compress` means "don't compress, but don't uncompress either". It would be useful if one of the compression options was provided as a default and you wanted to override it.

```
--modernize-patches
```

If you provide the `--modernize-patches` argument, darcs will convert obsolete patches into the current darcs format. This affects both the patch contents and the patch formatting.

Older versions of darcs formatted the long comments slightly differently, which can cause trouble with third-party tools that wish to parse the darcs patches, although darcs itself still reads the older patches fine. `--modernize-patches` standardizes the formatting of all patches.

In addition, *very* old versions of darcs created the "merger 0.9" patch type when there were conflicts. This patch type inherently had bugs which could lead to corruption, which is why it was phased out. `--modernize-patches` will convert old "merger 0.9" patches into an equivalent change (which will, however, commute differently).

```
--relink
```

The `--relink` and `--relink-pristine` options cause Darcs to relink files from a sibling. See Section 3.6.

```
--reorder-patches
```

The `--reorder-patches` option causes Darcs to create an optimal ordering of its internal patch inventory. This may help to produce shorter 'context' lists when sending patches, and may improve performance for some other operations as well. You should not run `--reorder-patches` on a repository from which someone may be simultaneously pulling or getting, as this could lead to repository corruption.

6.8 Undoing, redoing and running in circles

6.8.1 darcs amend-record

```
Usage: darcs amend-record [OPTION]... [FILE or DIRECTORY]...
```

Options:

	`--match PATTERN`	select patch matching PATTERN
`-p`	`--patch REGEXP`	select patch matching REGEXP
`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--no-test`	don't run the test script
	`--test`	run the test script
	`--leave-test-directory`	don't remove the test directory
	`--remove-test-directory`	remove the test directory
	`--compress`	create compressed patches
	`--dont-compress`	don't create compressed patches
`-a`	`--all`	answer yes to all patches
	`--interactive`	prompt user interactively
	`--ignore-times`	don't trust the file modification times
`-l`	`--look-for-adds`	In addition to modifications, look for files that are not boring, and thus are potentially pending addition
	`--dont-look-for-adds`	Don't look for any files or directories that could be added, and don't add them automatically
	`--repodir DIRECTORY`	specify the repository directory in which to run

Amend-record is used to replace a patch with a newer version with additional changes.

If you provide one or more files or directories as additional arguments to amend-record, you will only be prompted to changes in those files or directories.

The old version of the patch is lost and the new patch will include both the old and the new changes. This is mostly the same as unrecording the old patch, fixing the changes and recording a new patch with the same name and description.

`amend-record` will modify the date of the recorded patch. **WARNINGS:** You should *ONLY* use `amend-record` on patches which only exist in a single repository! Also, running amend-record while another user is pulling from the same repo may cause repository corruption.

If you configure darcs to run a test suite, darcs will run this test on the amended repo to make sure it is valid. Darcs first creates a pristine copy of the source tree (in a temporary directory), then it runs the test, using its return value to decide if the amended change is valid.

6.8.2 darcs rollback

```
Usage: darcs rollback [OPTION]...
```

Options:

	`--match PATTERN`	select patch matching PATTERN
`-p`	`--patch REGEXP`	select patch matching REGEXP
	`--compress`	create compressed patches
	`--dont-compress`	don't create compressed patches
`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--repodir DIRECTORY`	specify the repository directory in which to run

Rollback is used to undo the effects of a single patch without actually deleting that patch. Instead, it applies the inverse patch as a new patch. Unlike unpull and unrecord (which accomplish a similar goal) rollback is perfectly safe, since it leaves in the repository a record of the patch it is removing. If you decide you didn't want to roll back a patch after all, you probably should use unrecord to undo the rollback, since like rollback, unrecord doesn't affect the working directory.

6.8.3 darcs unrecord

`Usage: darcs unrecord [OPTION]...`

Options:

	`--from-match PATTERN`	select changes starting with a patch matching PATTERN
	`--from-patch REGEXP`	select changes starting with a patch matching REGEXP
	`--from-tag REGEXP`	select changes starting with a tag matching REGEXP
	`--last NUMBER`	select the last NUMBER patches
	`--matches PATTERN`	select patches matching PATTERN
`-p`	`--patches REGEXP`	select patches matching REGEXP
`-t`	`--tags REGEXP`	select tags matching REGEXP
`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--compress`	create compressed patches
	`--dont-compress`	don't create compressed patches
	`--repodir DIRECTORY`	specify the repository directory in which to run

Unrecord does the opposite of record in that it makes the changes from patches active changes again which you may record or revert later. The working copy itself will not change.

Unrecord can be thought of as undo-record. If a record is followed by an unrecord, everything looks like before the record; all the previously unrecorded changes are back, and can be recorded again in a new patch. The unrecorded patch however is actually removed from your repository, so there is no way to record it again to get it back.[5].

If you want to remove the changes from the working copy too (where they

[5]The patch file itself is not actually deleted, but its context is lost, so it cannot be reliably read—your only choice would be to go in by hand and read its contents.

otherwise will show up as unrecorded changes again), you'll also need to `darcs revert`. To do unrecord and revert in one go, you can use `darcs unpull`.

If you don't revert after unrecording, then the changes made by the unrecorded patches are left in your working tree. If these patches are actually from another repository, interaction (either pushes or pulls) with that repository may be massively slowed down, as darcs tries to cope with the fact that you appear to have made a large number of changes that conflict with those present in the other repository. So if you really want to undo the result of a *pull* operation, use unpull! Unrecord is primarily intended for when you record a patch, realize it needs just one more change, but would rather not have a separate patch for just that one change.

WARNING: Unrecord should not be run when there is a possibility that another user may be pulling from the same repo. Attempting to do so may cause repository corruption.

```
--from-match, --from-patch, --from-tag, --last
```

Usually you only want to unrecord the latest changes, and almost never would you want to unrecord changes before a tag—you would have to have unrecorded the tag as well to do that. Therefore, and for efficiency, darcs only prompts you for the latest patches, after some optimal tag.

If you do want to unrecord more patches in one go, there are the `--from` and `--last` options to set the earliest patch selectable to unrecord.

```
--matches, --patches, --tags
```

With these options you can specify what patch or patches to be prompted for by unrecord. This is especially useful when you want to unrecord patches with dependencies, since all the dependent patches (but no others) will be included in the choices.

These options can be slow if the list of patches to match is long, which can happen if `--from` or `--last` is used. The latter options can of course be used to *shorten* the list too, if it is long by default.

6.8.4 darcs unpull

`Usage: darcs unpull [OPTION]...`

Options:

	`--from-match PATTERN`	select changes starting with a patch matching PATTERN
	`--from-patch REGEXP`	select changes starting with a patch matching REGEXP
	`--from-tag REGEXP`	select changes starting with a tag matching REGEXP
	`--last NUMBER`	select the last NUMBER patches
	`--matches PATTERN`	select patches matching PATTERN
`-p`	`--patches REGEXP`	select patches matching REGEXP
`-t`	`--tags REGEXP`	select tags matching REGEXP
`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--compress`	create compressed patches
	`--dont-compress`	don't create compressed patches
	`--ignore-times`	don't trust the file modification times
	`--repodir DIRECTORY`	specify the repository directory in which to run

Unpull completely removes recorded patches from your local repository. The changes will be undone in your working copy and the patches will not be shown in your changes list anymore. Beware that if the patches are not still present in another repo you will lose precious code by unpulling!

Unlike unrecord, unpull does not just delete the patch from the repository, it actually applies an inverse patch to the repository. This makes unpull a particularly dangerous command, as it not only deletes the patch from the repo, but also removes the changes from the working directory. It is equivalent to an unrecord followed by a revert, except that revert can be unreverted.

WARNING: Unpull should not be run when there is a possibility that another user may be pulling from the same repo. Attempting to do so may cause repository corruption.

Contrary to what its name suggests, there is nothing in unpull that requires that the "unpulled" patch originate from a different repository. The name was chosen simply to suggest a situation in which it is "safe" to use unpull. If the patch was originally from another repo, then unpulling is safe, because you can always pull the patch again if you decide you want it after all. If you unpull a locally recorded patch, all record of that change is lost, which is what makes this a "dangerous" command, and thus deserving of an obscure name which is more suggestive of when it is safe to use than precisely what it does.

`--from-match, --from-patch, --from-tag, --last`

For efficiency, darcs only prompts you for the latest patches, after some optimal tag. If you do want to unpull more patches in one go, there are the `--from` and `--last` options to set the earliest patch selectable to unpull.

`--matches, --patches, --tags`

With these options you can specify what patch or patches to be prompted for by unpull. This is especially useful when you want to unpull patches with dependencies, since all the dependent patches (but no others) will be included in the choices.

In the case of tags, what you are unpulling is the tag itself, not any other patches.

These options can be slow if the list of patches to match with is long, which can happen if `--from` or `--last` is used. The latter options can of course be used to *shorten* the list too, if it is long by default.

6.8.5 darcs obliterate

`Usage: darcs obliterate [OPTION]...`

Options:

	`--from-match PATTERN`	select changes starting with a patch matching PATTERN
	`--from-patch REGEXP`	select changes starting with a patch matching REGEXP
	`--from-tag REGEXP`	select changes starting with a tag matching REGEXP
	`--last NUMBER`	select the last NUMBER patches
	`--matches PATTERN`	select patches matching PATTERN
`-p`	`--patches REGEXP`	select patches matching REGEXP
`-t`	`--tags REGEXP`	select tags matching REGEXP
`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--compress`	create compressed patches
	`--dont-compress`	don't create compressed patches
	`--ignore-times`	don't trust the file modification times
	`--repodir DIRECTORY`	specify the repository directory in which to run

Obliterate completely removes recorded patches from your local repository. The changes will be undone in your working copy and the patches will not be shown in your changes list anymore. Beware that you can lose precious code by obliterating!

Obliterate deletes a patch from the repository *and* removes those changes from the working directory. It is therefore a *very dangerous* command. When there are no local changes, obliterate is equivalent to an unrecord followed by a revert, except that revert can be unreverted. In the case of tags, obliterate removes the tag itself, not any other patches.

Note that obliterate is currently an alias for unpull.

WARNING: Obliterate should not be run when there is a possibility that another user may be pulling from the same repo. Attempting to do so may cause repository corruption.

`--from-match, --from-patch, --from-tag, --last`

For efficiency, darcs only prompts you for the latest patches, after some optimal tag. If you do want to unpull more patches in one go, there are the `--from` and `--last` options to set the earliest patch selectable to unpull.

`--matches, --patches, --tags`

With these options you can specify what patch or patches to be prompted for by unpull. This is especially useful when you want to unpull patches with dependencies, since all the dependent patches (but no others) will be included in the choices.

6.8.6 darcs revert

`Usage: darcs revert [OPTION]... [FILE or DIRECTORY]...`

Options:

`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--ignore-times`	don't trust the file modification times
`-a`	`--all`	answer yes to all patches
	`--interactive`	prompt user interactively
	`--repodir DIRECTORY`	specify the repository directory in which to run

Revert is used to undo changes made to the working copy which have not yet been recorded. You will be prompted for which changes you wish to undo. The last revert can be undone safely using the unrevert command if the working copy was not modified in the meantime. The actions of a revert may be reversed using the unrevert command (see subsection 6.8.7). However, if you've made changes since the revert your mileage may vary, so please be careful.

You can give revert optional arguments indicating files or directories. If you do so it will only prompt you to revert changes in those files or in files in those directories.

6.8.7 darcs unrevert

`Usage: darcs unrevert [OPTION]...`

Options:

`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--ignore-times`	don't trust the file modification times
`-a`	`--all`	answer yes to all patches
	`--interactive`	prompt user interactively
	`--repodir DIRECTORY`	specify the repository directory in which to run

Unrevert is used to undo the results of a revert command. It is only guaranteed to work properly if you haven't made any changes since the revert was performed.

The command makes a best effort to merge the unreversion with any changes you have since made. In fact, unrevert should even work if you've recorded changes since reverting.

6.9 Advanced examination of the repository

6.9.1 darcs diff

`Usage: darcs diff [OPTION]... [FILE or DIRECTORY]...`

Options:

	`--to-match PATTERN`	select changes up to a patch matching PATTERN
	`--to-patch REGEXP`	select changes up to a patch matching REGEXP
	`--to-tag REGEXP`	select changes up to a tag matching REGEXP
	`--from-match PATTERN`	select changes starting with a patch matching PATTERN
	`--from-patch REGEXP`	select changes starting with a patch matching REGEXP
	`--from-tag REGEXP`	select changes starting with a tag matching REGEXP
	`--match PATTERN`	select a single patch matching PATTERN
`-p`	`--patch REGEXP`	select a single patch matching REGEXP
	`--last NUMBER`	select the last NUMBER patches
	`--diff-opts OPTIONS`	options to pass to diff
`-u`	`--unified`	pass -u option to diff
	`--repodir DIRECTORY`	specify the repository directory in which to run

Diff can be used to create a diff between two versions which are in your repository. Specifying just –from-patch will get you a diff against your working copy. If you give diff no version arguments, it gives you the same information as whatsnew except that the patch is formatted as the output of a diff command

```
--diff-opts
```

Diff calls an external "diff" command to do the actual work, and passes any unrecognized flags to this diff command. Thus you can call

```
% darcs diff -t 0.9.8 -t 0.9.10 -- -u
```

to get a diff in the unified format. Actually, thanks to the wonders of getopt you need the "`--`" shown above before any arguments to diff. You can also specify additional arguments to diff using the `--diff-opts` flag. The above command would look like this:

```
% darcs diff --diff-opts -u -t 0.9.8 -t 0.9.10
```

This may not seem like an improvement, but it really pays off when you want to always give diff the same options. You can do this by adding

```
% diff diff-opts -udp
```

to your `_darcs/prefs/defaults` file.

If you want to view only the differences to one or more files, you can do so with a command such as

```
% darcs diff foo.c bar.c baz/
```

FIXME: I should allow the user to specify the external diff command. Currently it is hardwired to “diff”.

6.9.2 darcs annotate

`Usage: darcs annotate [OPTION]... [FILE or DIRECTORY]...`

Options:

`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don’t give verbose output
`-s`	`--summary`	summarize changes
	`--no-summary`	don’t summarize changes
`-u`	`--unified`	output patch in a darcs-specific format similar to diff -u
	`--human-readable`	give human-readable output
	`--xml-output`	generate XML formatted output
	`--match PATTERN`	select patch matching PATTERN
`-p`	`--patch REGEXP`	select patch matching REGEXP
`-t`	`--tag REGEXP`	select tag matching REGEXP
	`--creator-hash HASH`	specify hash of creator patch (see docs)
	`--repodir DIRECTORY`	specify the repository directory in which to run

Display which patch last modified something. Annotate displays which patches created or last modified a directory file or line. It can also display the contents of a particular patch in darcs format.

`--human-readable, --summary, --unified, --xml--output`

When called with just a patch name, annotate outputs the patch in darcs format, which is the same as `--human-readable`.

`--xml-output` is the alternative to `--human-readable`.

`--summary` can be used with either the `--xml-output` or the `--human-readable` options to alter the results. It is documented fully in the ‘common options’ portion of the manual.

Giving the `--unified` flag implies `--human-readable`, and causes the output to remain in a darcs-specific format that is similar to that produced by `diff --unified`.

If a directory name is given, annotate will output details of the last modifying patch for each file in the directory and the directory itself. The details look like this:

```
# Created by [bounce handling patch
# mark**20040526202216]  as ./test/m7/bounce_handling.pl
   bounce_handling.pl
```

If a patch name and a directory are given, these details are output for the time after that patch was applied. If a directory and a tag name are given, the details of the patches involved in the specified tagged version will be output.

If a file name is given, the last modifying patch details of that file will be output, along with markup indicating patch details when each line was last (and perhaps next) modified.

```
--creator-hash HASH
```

The `--creator-hash` option should only be used in combination with a file or directory to be annotated. In this case, the name of that file or directory is interpreted to be its name *at the time it was created*, and the hash given along with `--creator-hash` indicates the patch that created the file or directory. This allows you to (relatively) easily examine a file even if it has been renamed multiple times.

6.9.3 darcs query manifest

`Usage: darcs query manifest [OPTION]...`

Options:

	`--files`	include files in output [DEFAULT]
	`--no-files`	do not include files in output
	`--directories`	include directories in output
	`--no-directories`	do not include directories in output [DEFAULT]
	`--pending`	reflect pending patches in output [DEFAULT]
	`--no-pending`	only included recorded patches in output
`-0`	`--null`	separate file names by NUL characters
	`--repodir DIRECTORY`	specify the repository directory in which to run

The manifest command lists the version-controlled files in the working copy.

By default (and if the `--pending` option is specified), the effect of pending patches on the repository is taken into account. In other words, if you add a file using `darcs add`, it immediately appears in the output of `query manifest`, even if it is not yet recorded. If you specify the `--no-pending` option, `query manifest` will only list recorded files (and directories).

The `--files` and `--directories` options control whether files and directories are included in the output. The `--no-files` and `--no-directories` options have the reverse effect. The default is to include files, but not directories.

If you specify the `--null` option, the file names are written to standard output in unescaped form, and separated by ASCII NUL bytes. This format is suitable for further automatic processing (for example, using `xargs -0`).

6.10 Rarely needed and obscure commands

6.10.1 darcs resolve

`Usage: darcs resolve [OPTION]...`

Options:

`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output
	`--ignore-times`	don't trust the file modification times
	`--repodir DIRECTORY`	specify the repository directory in which to run

Resolve is used to mark and resolve any conflicts that may exist in a repository. Note that this trashes any unrecorded changes in the working copy.

6.10.2 darcs dist

`Usage: darcs dist [OPTION]...`

Options:

`-d`	`--dist-name DISTNAME`	name of version
`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output

Create a distribution tarball.

Dist is a handy tool for implementing a "make dist" target in your makefile. It creates a tarball of the recorded edition of your tree. Basically, you will typically use it in a makefile rule such as

```
dist:
    darcs dist --dist-name darcs-`./darcs --version`
```

`darcs dist` then simply creates a clean copy of the source tree, which it then tars and gzips. If you use programs such as autoconf or automake, you really should run them on the clean tree before tarring it up and distributing it. You can do this using the pref value "predist", which is a shell command that is run prior to tarring up the distribution:

```
% darcs setpref predist "autoconf && automake"
```

6.10.3 darcs trackdown

`Usage: darcs trackdown [OPTION]... [[INITIALIZATION] COMMAND]`

Options:

`-v`	`--verbose`	give verbose output
	`--standard-verbosity`	don't give verbose output

Trackdown tries to find the most recent version in the repository which passes a test. Given no arguments, it uses the default repository test. Given one argument, it treats it as a test command. Given two arguments, the first is an initialization command with is run only once, and the second is the test command.

Trackdown is helpful for locating when something was broken. It creates a temporary directory with the latest repo content in it and cd to it. First, and only once, it runs the initialization command if any, for example

```
'autoconf; ./configure >/dev/null'
```

Then it runs the test command, for example

```
'make && cd tests && sh /tmp/test.sh'
```

While the test command exits with an error return code, darcs "unapplies" one patch from the version controlled files to retrieve an earlier version, and repeats the test command. If the test command finally succeeds, the name of the hunted down patch is found in the output before the last test run.

FIXME: It is still rather primitive. Currently it just goes back over the history in reverse order trying each version. I'd like for it to explore different patch combinations, to try to find the minimum number of patches that you would need to unpull in order to make the test succeed.

FIXME: I also would like to add an interface by which you can tell it which patches it should consider not including. Without such a feature, the following command:

```
% darcs trackdown 'make && false'
```

would result in compiling every version in the repository–which is a rather tedious prospect.

Example usage

If you want to find the last version of darcs that had a FIXME note in the file Record.lhs, you could run

```
% darcs trackdown 'grep FIXME Record.lhs'
```

To find the latest version that compiles, you can run

```
% darcs trackdown 'autoconf' './configure && make'
```

Trackdown can also be used to see how other features of the code changed with time. For example

```
% darcs trackdown 'autoconf; ./configure' \
   "make darcs &> /dev/null && cd ~/darcs && time darcs check && false"
```

would let you see how long 'darcs check' takes to run on each previous version of darcs that will actually compile. The "`&& false`" ensures that trackdown keeps going.

6.10.4 darcs repair

`Usage: darcs repair [OPTION]...`

Options:

`-v`	`--verbose`	give verbose output
`-q`	`--quiet`	suppress informational output
	`--standard-verbosity`	neither verbose nor quiet output

Repair attempts to fix corruption that may have entered your repository.

Repair currently will only repair damage to the pristine tree. Fortunately this is just the sort of corruption that is most likely to happen.

Appendix A

Theory of patches

A.1 Background

I think a little background on the author is in order. I am a physicist, and think like a physicist. The proofs and theorems given here are what I would call "physicist" proofs and theorems, which is to say that while the proofs may not be rigorous, they are practical, and the theorems are intended to give physical insight. It would be great to have a mathematician work on this, but I am not a mathematician, and don't care for math.

From the beginning of this theory, which originated as the result of a series of email discussions with Tom Lord, I have looked at patches as being analogous to the operators of quantum mechanics. I include in this appendix footnotes explaining the theory of patches in terms of the theory of quantum mechanics. I know that for most people this won't help at all, but many of my friends (and as I write this all three of darcs' users) are physicists, and this will be helpful to them. To non-physicists, perhaps it will provide some insight into how at least this physicist thinks.

A.2 Introduction

A patch describes a change to the tree. It could be either a primitive patch (such as a file add/remove, a directory rename, or a hunk replacement within a file), or a composite patch describing many such changes. Every patch type must satisfy the conditions described in this appendix. The theory of patches is independent of the data which the patches manipulate, which is what makes it both powerful and useful, as it provides a framework upon which one can build a revision control system in a sane manner.

Although in a sense, the defining property of any patch is that it can be applied to a certain tree, and thus make a certain change, this change does not wholly define the patch. A patch is defined by a *representation*, together with a set of rules for how it behaves (which it has in common with its patch type).

The *representation* of a patch defines what change that particular patch makes, and must be defined in the context of a specific tree. The theory of patches is a theory of the many ways one can change the representation of a patch to place it in the context of a different tree. The patch itself is not changed, since it describes a single change, which must be the same regardless of its representation[1].

So how does one define a tree, or the context of a patch? The simplest way to define a tree is as the result of a series of patches applied to the empty tree[2]. Thus, the context of a patch consists of the set of patches that precede it.

A.3 Applying patches

A.3.1 Hunk patches

Hunks are an example of a complex filepatch. A hunk is a set of lines of a text file to be replaced by a different set of lines. Either of these sets may be empty, which would mean a deletion or insertion of lines.

A.3.2 Token replace patches

Although most filepatches will be hunks, darcs is clever enough to support other types of changes as well. A "token replace" patch replaces all instances of a given token with some other version. A token, here, is defined by a regular expression, which must be of the simple [a–z...] type, indicating which characters are allowed in a token, with all other characters acting as delimiters. For example, a C identifier would be a token with the flag `[A-Za-z_0-9]`.

What makes the token replace patch special is the fact that a token replace can be merged with almost any ordinary hunk, giving exactly what you would want. For example, you might want to change the patch type `TokReplace` to `TokenReplace` (if you decided that saving two characters of space was stupid). If you did this using hunks, it would modify every line where `TokReplace` occurred, and quite likely provoke a conflict with another patch modifying those lines. On the other hand, if you did this using a token replace patch, the only change that it could conflict with would be if someone else had used the token "`TokenReplace`" in their patch rather than TokReplace—and that actually would be a real conflict!

[1]For those comfortable with quantum mechanics, think of a patch as a quantum mechanical operator, and the representation as the basis set. The analogy breaks down pretty quickly, however, since an operator could be described in any complete basis set, while a patch modifying the file `foo` can only be described in the rather small set of contexts which have a file `foo` to be modified.

[2]This is very similar to the second-quantized picture, in which any state is seen as the result of a number of creation operators acting on the vacuum, and provides a similar set of simplifications—in particular, the exclusion principle is very elegantly enforced by the properties of the anti-hermitian fermion creation operators.

A.4 Patch relationships

The simplest relationship between two patches is that of "sequential" patches, which means that the context of the second patch (the one on the left) consists of the first patch (on the right) plus the context of the first patch. The composition of two patches (which is also a patch) refers to the patch which is formed by first applying one and then the other. The composition of two patches, P_1 and P_2 is represented as P_2P_1, where P_1 is to be applied first, then P_2[3]

There is one other very useful relationship that two patches can have, which is to be parallel patches, which means that the two patches have an identical context (i.e. their representation applies to identical trees). This is represented by $P_1 \parallel P_2$. Of course, two patches may also have no simple relationship to one another. In that case, if you want to do something with them, you'll have to manipulate them with respect to other patches until they are either in sequence or in parallel.

The most fundamental and simple property of patches is that they must be invertible. The inverse of a patch is described by: P^{-1}. In the darcs implementation, the inverse is required to be computable from knowledge of the patch only, without knowledge of its context, but that (although convenient) is not required by the theory of patches.

Definition 1 *The inverse of patch P is P^{-1}, which is the "simplest" patch for which the composition $P^{-1}P$ makes no changes to the tree.*

Using this definition, it is trivial to prove the following theorem relating to the inverse of a composition of two patches.

Theorem 1 *The inverse of the composition of two patches is*

$$(P_2P_1)^{-1} = P_1^{-1}P_2^{-1}.$$

Moreover, it is possible to show that the right inverse of a patch is equal to its left inverse. In this respect, patches continue to be analogous to square matrices, and indeed the proofs relating to these properties of the inverse are entirely analogous to the proofs in the case of matrix multiplication. The compositions proofs can also readily be extended to the composition of more than two patches.

A.5 Commuting patches

A.5.1 Composite patches

Composite patches are made up of a series of patches intended to be applied sequentially. They are represented by a list of patches, with the first patch in the list being applied first.

[3]This notation is inspired by the notation of matrix multiplication or the application of operators upon a Hilbert space. In the algebra of patches, there is multiplication (i.e. composition), which is associative but not commutative, but no addition or subtraction.

The first way (of only two) to change the context of a patch is by commutation, which is the process of changing the order of two sequential patches.

Definition 2 *The commutation of patches P_1 and P_2 is represented by*

$$P_2 P_1 \longleftrightarrow {P_1}' {P_2}'.$$

Here P_1' is intended to describe the same change as P_1, with the only difference being that P_1' is applied after P_2' rather than before P_2.

The above definition is obviously rather vague, the reason being that what is the "same change" has not been defined, and we simply assume (and hope) that the code's view of what is the "same change" will match those of its human users. The '$\longleftrightarrow$' operator should be read as something like the == operator in C, indicating that the right hand side performs identical changes to the left hand side, but the two patches are in reversed order. When read in this manner, it is clear that commutation must be a reversible process, and indeed this means that commutation *can* fail, and must fail in certain cases. For example, the creation and deletion of the same file cannot be commuted. When two patches fail to commute, it is said that the second patch depends on the first, meaning that it must have the first patch in its context (remembering that the context of a patch is a set of patches, which is how we represent a tree). [4]

Merge The second way one can change the context of a patch is by a **merge** operation. A merge is an operation that takes two parallel patches and gives a pair of sequential patches. The merge operation is represented by the arrow "$\Longrightarrow$".

Definition 3 *The result of a merge of two patches, P_1 and P_2 is one of two patches, P_1' and P_2', which satisfy the relationship:*

$$P_2 \parallel P_1 \Longrightarrow {P_2}' P_1 \longleftrightarrow {P_1}' P_2.$$

Note that the sequential patches resulting from a merge are *required* to commute. This is an important consideration, as without it most of the manipulations we would like to perform would not be possible. The other important fact is that a merge *cannot fail.* Naively, those two requirements seem contradictory. In reality, what it means is that the result of a merge may be a patch which is much more complex than any we have yet considered[5].

[4]The fact that commutation can fail makes a huge difference in the whole patch formalism. It may be possible to create a formalism in which commutation always succeeds, with the result of what would otherwise be a commutation that fails being something like a virtual particle (which can violate conservation of energy), and it may be that such a formalism would allow strict mathematical proofs (whereas those used in the current formalism are mostly only hand waving "physicist" proofs). However, I'm not sure how you'd deal with a request to delete a file that has not yet been created, for example. Obviously you'd need to create some kind of antifile, which would annihilate with the file when that file finally got created, but I'm not entirely sure how I'd go about doing this. $\ddot\frown$ So I'm sticking with my hand waving formalism.

[5]Alas, I don't know how to prove that the two constraints even *can* be satisfied. The best I have been able to do is to believe that they can be satisfied, and to be unable to find an case in which my implementation fails to satisfy them. These two requirements are the foundation of the entire theory of patches (have you been counting how many foundations it has?).

A.5.2 How merges are actually performed

The constraint that any two compatible patches (patches which can successfully be applied to the same tree) can be merged is actually quite difficult to apply. The above merge constraints also imply that the result of a series of merges must be independent of the order of the merges. So I'm putting a whole section here for the interested to see what algorithms I use to actually perform the merges (as this is pretty close to being the most difficult part of the code).

The first case is that in which the two merges don't actually conflict, but don't trivially merge either (e.g. hunk patches on the same file, where the line number has to be shifted as they are merged). This kind of merge can actually be very elegantly dealt with using only commutation and inversion.

There is a handy little theorem which is immensely useful when trying to merge two patches.

Theorem 2 $P_2'P_1 \longleftrightarrow P_1'P_2$ *if and only if* $P_1'^{-1}P_2' \longleftrightarrow P_2P_1^{-1}$*, provided both commutations succeed. If either commute fails, this theorem does not apply.*

This can easily be proven by multiplying both sides of the first commutation by $P_1'^{-1}$ on the left, and by P_1^{-1} on the right. Besides being used in merging, this theorem is also useful in the recursive commutations of mergers. From Theorem 2, we see that the merge of P_1 and P_2' is simply the commutation of P_2 with P_1^{-1} (making sure to do the commutation the right way). Of course, if this commutation fails, the patches conflict. Moreover, one must check that the merged result actually commutes with P_1, as the theorem applies only when *both* commutations are successful.

Of course, there are patches that actually conflict, meaning a merge where the two patches truly cannot both be applied (e.g. trying to create a file and a directory with the same name). We deal with this case by creating a special kind of patch to support the merge, which we will call a "merger". Basically, a merger is a patch that contains the two patches that conflicted, and instructs darcs basically to resolve the conflict. By construction a merger will satisfy the commutation property (see Definition 3) that characterizes all merges. Moreover the merger's properties are what makes the order of merges unimportant (which is a rather critical property for darcs as a whole).

The job of a merger is basically to undo the two conflicting patches, and then apply some sort of a "resolution" of the two instead. In the case of two conflicting hunks, this will look much like what CVS does, where it inserts both versions into the file. In general, of course, the two conflicting patches may both be mergers themselves, in which case the situation is considerably more complicated.

Much of the merger code depends on a routine which recreates from a single merger the entire sequence of patches which led up to that merger (this is, of course, assuming that this is the complicated general case of a merger of mergers of mergers). This "unwind" procedure is rather complicated, but absolutely critical to the merger code, as without it we wouldn't even be able to undo the effects of the patches involved in the merger, since we wouldn't know what patches were all involved in it.

Basically, unwind takes a merger such as

```
M( M(A,B), M(A,M(C,D)))
```

From which it recreates a merge history:

```
C
A
M(A,B)
M( M(A,B), M(A,M(C,D)))
```

(For the curious, yes I can easily unwind this merger in my head [and on paper can unwind insanely more complex mergers]—that's what comes of working for a few months on an algorithm.) Let's start with a simple unwinding. The merger `M(A,B)` simply means that two patches (`A` and `B`) conflicted, and of the two of them `A` is first in the history. The last two patches in the unwinding of any merger are always just this easy. So this unwinds to:

```
A
M(A,B)
```

What about a merger of mergers? How about `M(A,M(C,D))`. In this case we know the two most recent patches are:

```
A
M(A,M(C,D))
```

But obviously the unwinding isn't complete, since we don't yet see where `C` and `D` came from. In this case we take the unwinding of `M(C,D)` and drop its latest patch (which is `M(C,D)` itself) and place that at the beginning of our patch train:

```
C
A
M(A,M(C,D))
```

As we look at `M( M(A,B), M(A,M(C,D)))`, we consider the unwindings of each of its subpatches:

```
          C
A         A
M(A,B)    M(A,M(C,D))
```

As we did with `M(A,M(C,D))`, we'll drop the first patch on the right and insert the first patch on the left. That moves us up to the two `A`'s. Since these agree, we can use just one of them (they "should" agree). That leaves us with the `C` which goes first.

The catch is that things don't always turn out this easily. There is no guarantee that the two `A`'s would come out at the same time, and if they didn't, we'd have to rearrange things until they did. Or if there was no way to rearrange things so that they would agree, we have to go on to plan B, which I will explain now.

Consider the case of `M( M(A,B), M(C,D))`. We can easily unwind the two subpatches

```
A         C
M(A,B)    M(C,D)
```

Now we need to reconcile the `A` and `C`. How do we do this? Well, as usual, the solution is to use the most wonderful Theorem 2. In this case we have to use it in the reverse of how we used it when merging, since we know that `A` and `C` could either one be the *last* patch applied before `M(A,B)` or `M(C,D)`. So we can find `C'` using

$$A^{-1}C \longleftrightarrow C'A'^{-1}$$

Giving an unwinding of

```
C'
A
M(A,B)
M( M(A,B), M(C,D) )
```

There is a bit more complexity to the unwinding process (mostly having to do with cases where you have deeper nesting), but I think the general principles that are followed are pretty much included in the above discussion.

It can sometimes be handy to have a canonical representation of a given patch. We achieve this by defining a canonical form for each patch type, and a function "`canonize`" which takes a patch and puts it into canonical form. This routine is used by the diff function to create an optimal patch (based on an LCS algorithm) from a simple hunk describing the old and new version of a file. Note that canonization may fail, if the patch is internally inconsistent.

A simpler, faster (and more generally useful) cousin of canonize is the coalescing function. This takes two sequential patches, and tries to turn them into one patch. This function is used to deal with "split" patches, which are created when the commutation of a primitive patch can only be represented by a composite patch. In this case the resulting composite patch must return to the original primitive patch when the commutation is reversed, which a split patch accomplishes by trying to coalesce its contents each time it is commuted.

A.5.3 File patches

A file patch is a patch which only modifies a single file. There are some rules which can be made about file patches in general, which makes them a handy class. For example, commutation of two filepatches is trivial if they modify different files. There is an exception when one of the files has a name ending with "-conflict", in which case it may not commute with a file having the same name, but without the "-conflict." If they happen to modify the same file, we'll have to check whether or not they commute.

There is another handy function, which primarily affects file patches (although it can also affect other patches, such as rename patches or dir add/remove patches), which is the submerge-in-directory function. This function changes the patch to act on a patch within a subdirectory rather than in the current directory, and is useful when performing the recursive diff.

A.5.4 Hunks

The hunk is the simplest patch that has a commuting pattern in which the commuted patches differ from the originals (rather than simple success or failure). This makes commuting or merging two hunks a tad tedious. Hunks, of course, can be coalesced if they have any overlap. Note that coalesce code doesn't check if the two patches are conflicting. If you are coalescing two conflicting hunks, you've already got a bug somewhere.

One of the most important pieces of code is the canonization of a hunk, which is where the "diff" algorithm is performed. This algorithm begins with chopping off the identical beginnings and endings of the old and new hunks. This isn't strictly necessary, but is a good idea, since this process is $O(n)$, while the primary diff algorithm is something considerably more painful than that... actually the head would be dealt with all right, but with more space complexity. I think it's more efficient to just chop the head and tail off first.

A.6 Conflicts

There are a couple of simple constraints on the routine which determines how to resolve two conflicting patches (which is called 'glump'). These must be satisfied in order that the result of a series of merges is always independent of their order. Firstly, the output of glump cannot change when the order of the two conflicting patches is switched. If it did, then commuting the merger could change the resulting patch, which would be bad. Secondly, the result of the merge of three (or more) conflicting patches cannot depend on the order in which the merges are performed.

The conflict resolution code (glump) begins by "unravelling" the merger into a set of sequences of patches. Each sequence of patches corresponds to one non-conflicted patch that got merged together with the others. The result of the unravelling of a series of merges must obviously be independent of the order in which those merges are performed. This unravelling code (which uses the unwind code mentioned above) uses probably the second most complicated algorithm. Fortunately, if we can successfully unravel the merger, almost any function of the unravelled merger satisfies the two constraints mentioned above that the conflict resolution code must satisfy.

A.7 Patch string formatting

Of course, in order to store our patches in a file, we'll have to save them as some sort of strings. The convention is that each patch string will end with a newline, but on parsing we skip any amount of whitespace between patches.

Composite patch A patch made up of a few other patches.

```
{
  <put patches here> (indented two)
}
```

Split patch A split patch is similar to a composite patch (identical in how it's stored), but rather than being composed of several patches grouped together, it is created from one patch that has been split apart, typically through a merge or commutation.

```
(
  <put patches here> (indented two)
)
```

Hunk Replace a hunk (set of contiguous lines) of text with a new hunk.

```
hunk FILE LINE#
-LINE
...
+LINE
...
```

Token replace Replace a token with a new token. Note that this format means that the white space must not be allowed within a token. If you know of a practical application of whitespace within a token, let me know and I may change this.

```
replace FILENAME [REGEX] OLD NEW
```

Binary file modification Modify a binary file

```
binary FILENAME
oldhex
*HEXHEXHEX
...
newhex
*HEXHEXHEX
...
```

Add file Add an empty file to the tree.

```
addfile filename
```

Remove file Delete a file from the tree.

```
rmfile filename
```

Move Rename a file or directory.

```
move oldname newname
```

Change Pref Change one of the preference settings. Darcs stores a number of simple string settings. Among these are the name of the test script and the name of the script that must be called prior to packing in a make dist.

```
changepref prefname
oldval
newval
```

Add dir Add an empty directory to the tree.

```
adddir filename
```

Remove dir Delete a directory from the tree.

```
rmdir filename
```

Merger patches Merge two patches. The MERGERVERSION is included to allow some degree of backwards compatibility if the merger algorithm needs to be changed.

```
merger MERGERVERSION
<first patch>
<second patch>
```

Conflictor patches The conflictor patch type is the replacement for the old merger patch type. FIXME: More explanation should be added here.

```
conflict
<CONFLICTING PATCH SEQUENCE>
with
<OLDER PATCH SEQUENCE>
tcilfnoc
```

Named patches Named patches are displayed as a "patch id" which is in square brackets, followed by a patch. Optionally, after the patch id (but before the patch itself) can come a list of dependencies surrounded by angle brackets. Each dependency consists of a patch id.

Appendix B

DarcsRepo format

A repository consists of a working directory, which has within it a directory called `_darcs`. There must also be subdirectories within `_darcs` named `current` and `patches`. The `current` directory, called the *pristine tree*, contains the version of the tree which has been recorded, while `patches` contains the actual patches which are in the repository.

WARNING! Viewing files in current is perfectly acceptable, but if you view them with an editor (e.g. vi or Emacs), that editor may create temporary files in the pristine tree (`_darcs/pristine/` or `_darcs/current/`), which will temporarily cause your repository to be inconsistent. So *don't record any patches while viewing files in _darcs/current with an editor!* A better plan would be to restrict yourself to viewing these files with a pager such as more or less.

Also within `_darcs` is the `inventory` file, which lists all the patches that are in the repo. Moreover, it also gives the order of the representation of the patches as they are stored. Given a source of patches, i.e. any other set of repositories which have between them all the patches contained in a given repo, that repo can be reproduced based on only the information in the `inventory` file. Under those circumstances, the order of the patches specified in the `inventory` file would be unimportant, as this order is only needed to provide context for the interpretation of the stored patches in this repository.

There is a very special patch which may be stored in `patches` which is called 'pending'. This patch describes any changes which have not yet been recorded, and cannot be determined by a simple diff. For example, file additions or renames are placed in pending until they are recorded. Similarly, token replaces are stored in pending until they are recorded.

The `_darcs` directory also contains a directory called "`prefs`", which is described in Chapter 4.

Appendix C

The GNU General Public License

Version 2, June 1991

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

C.1 Appendix: How to Apply These Terms to Your New Programs

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.