

Darcs User Manual

David Roundy

2.3.0 (release)

Contents

1	Introduction	7
1.1	Features	9
2	Getting started	13
2.1	Creating your repository	13
2.2	Making changes	14
2.3	Making your repository visible to others	14
2.4	Getting changes made to another repository	15
2.5	Moving patches from one repository to another	15
2.5.1	All pulls	15
2.5.2	Send and apply manually	15
2.5.3	Push	16
2.5.4	Push —apply-as	16
2.5.5	Sending signed patches by email	17
3	Configuring darcs	21
3.1	prefs	21
3.2	Environment variables	25
3.3	General-purpose variables	25
3.4	Remote repositories	26
3.5	Highlighted output	29
3.6	Character escaping and non-ASCII character encodings	29
4	Best practices	31
4.1	Creating patches	31
4.1.1	Changes	31
4.1.2	Keeping or discarding changes	32
4.1.3	Unrecording changes	32
4.1.4	Special patches and pending	33
4.2	Using patches	33
4.2.1	Dependencies	34
4.2.2	Branches: just normal repositories	34
4.2.3	Moving patches around—no versions	35
4.2.4	Tags—versions	35

4.2.5	Conflicts	35
4.2.6	Resolving conflicts	37
4.3	Use a Global Cache	37
4.4	Distributed development with one primary developer	37
4.5	Development by a small group of developers in one office	38
4.6	Personal development	39
4.6.1	Private patches	39
5	Repository Formats	43
6	Darcs commands	45
6.1	Common options to darcs commands	46
6.2	Options apart from darcs commands	55
6.3	Getting help	56
6.3.1	darcs help	56
6.4	Creating repositories	56
6.4.1	darcs initialize	56
6.4.2	darcs get	57
6.4.3	darcs put	58
6.5	Modifying the contents of a repository	59
6.5.1	darcs add	59
6.5.2	darcs remove	60
6.5.3	darcs move	61
6.5.4	darcs replace	61
6.6	Working with changes	62
6.6.1	darcs record	62
6.6.2	darcs pull	66
6.6.3	darcs push	68
6.6.4	darcs send	71
6.6.5	darcs apply	74
6.7	Seeing what you've done	77
6.7.1	darcs whatsnew	77
6.7.2	darcs changes	78
6.7.3	darcs show	79
6.8	More advanced commands	82
6.8.1	darcs tag	82
6.8.2	darcs setpref	83
6.8.3	darcs check	83
6.8.4	darcs optimize	84
6.9	Undoing, redoing and running in circles	86
6.9.1	darcs amend-record	86
6.9.2	darcs rollback	87
6.9.3	darcs unrecord	88
6.9.4	darcs obliterate	90
6.9.5	darcs revert	92
6.9.6	darcs unrevert	92

6.10	Advanced examination of the repository	93
6.10.1	darcs diff	93
6.10.2	darcs annotate	94
6.10.3	darcs show	96
6.11	Rarely needed and obscure commands	98
6.11.1	darcs convert	98
6.11.2	darcs mark-conflicts	99
6.11.3	darcs dist	100
6.11.4	darcs trackdown	100
6.11.5	darcs repair	102
.1	Switching from CVS	102
.2	Switching from arch	103
A	Building darcs	105
A.1	The Easy Way	105
A.2	The Hard Way	105
A.3	The Old Way	106
B	Theory of patches	107
B.1	Background	107
B.2	Introduction	107
B.3	Applying patches	108
B.3.1	Hunk patches	108
B.3.2	Token replace patches	108
B.4	Patch relationships	109
B.4.1	File patches	111
B.4.2	Hunks	111
B.5	Commuting patches	111
B.5.1	Composite patches	111
B.5.2	How merges are actually performed	113
B.6	Conflicts	115
B.7	Patch string formatting	116
C	DarcsRepo format	117
D	The GNU General Public License	119
D.1	Appendix: How to Apply These Terms to Your New Programs	125

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 B.

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 3 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 B.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 B 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.

Chapter 2

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.

2.1 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

¹Note 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.

2.2 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 6.7.1, 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.

2.3 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 repository there:

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

As long as you're running a web server and making your repository 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 repositories by pointing their web browsers at `http://your.server.org/cgi-bin/darcs`.

2.4 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 repository.

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`.

2.5 Moving patches from one repository to another

Darcs is flexible as to how you move patches from one repository 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.

2.5.1 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 repository 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.

2.5.2 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 repository)

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:

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

which allows me to view a sent patch, and then apply the patch directly from `mutt`, sending a confirmation email to the person who sent me the patch. The `autoview` line relies on on the following lines, or something like them, being present in one's `.mailcap`:

```
text/x-patch;                cat; copiousoutput
text/x-darcs-patch;          cat; copiousoutput
```

2.5.3 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
```

2.5.4 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 --reporid /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.

2.5.5 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 repository 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 repository 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').

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. 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 repository. 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
* ^T0my 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.

Like most mail-processing programs, Procmail by default sets a tight umask. However, this will prevent the repository from remaining world-readable; thus, the “umask 022” is required to relax the umask. (Alternatively, you could set Procmail's global `UMASK` variable to a more suitable value.)

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 repository by pointing `darcs changes` at a remote repository:

```
darcs changes --last=10 --repo=http://darcs.net/
```

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

Chapter 3

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.

Microsoft Windows The global darcs directory is `%APPDATA%\darcs\`. This typically expands to `C:\Documents And Settings\user\Application Data\darcs\`. This folder contains the cache, as well as all the per-user settings files: preferences, boring etc... These will become the new defaults that can be overridden on per-repository basis.

3.1 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. If the value has spaces in it, use single quotes, not double quotes, to surround it. Each line only takes one flag. To set multiple defaults for the same command (or for **ALL** commands), use multiple lines.

Note that the use of **ALL** easily can have unpredicted consequences, especially if commands in newer versions of darcs accepts flags that they didn't in previous versions. A command like **obliterate** could be devastating with the "wrong" flags (for example **-all**). Only use safe flags with **ALL**.

<code>~/.darcs/defaults</code>	provides defaults for this user account, on MS Windows 3
<code>repo/_darcs/prefs/defaults</code>	provides defaults for one project,
	overrides changes per user

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 repository), 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 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
```

Similarly, if you need a command to run automatically before darcs performs an action you can use a prehook. Using prehooks it could be possible to canonicalize line endings before recording patches.

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, on MS Windows 3. 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@abridggame.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 preference 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`, on MS Windows 3, which will be used with all of your darcs repositories.

Any file not already managed by darcs and 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`. Note that once a file has been added to darcs, it is not considered boring, even if it matches the boring file filter.

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. Darcs automatically treats files containing `^Z\` or `'\0'` within the first 4096 bytes as being binary 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 “binaries-file” 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, on MS Windows 3.

email The `_darcs/prefs/email` file is used to provide the e-mail address for your repository 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.

sources The `_darcs/prefs/sources` file is used to indicate alternative locations from which to download patches when using a “hashed” repository. This file contains lines such as:

```
cache:/home/droundy/_darcs/cache
readonly:/home/otheruser/_darcs/cache
repo:http://darcs.net
```

This would indicate that darcs should first look in `/home/droundy/_darcs/cache` for patches that might be missing, and if the patch isn’t there, it should save a copy there for future use. In that case, darcs will look in `/home/otheruser/_darcs/cache` to see if that user might have downloaded a copy, but won’t try to save a copy there, of course. Finally, it will look in `http://darcs.net`. Note that the `sources` file can also exist in `~/_darcs/`. Also note that the sources mentioned in your `sources` file will be tried *before* the repository you are pulling from. This can be useful in avoiding downloading patches multiple times when you pull from a remote repository to more than one local repository.

We strongly advise that you enable a global cache directory, which will allow darcs to avoid re-downloading patches (for example, when doing a second `darcs get` of the same repository), and also allows darcs to use hard links to reduce disk usage. To do this, simply

```
mkdir -p $HOME/_darcs/cache
echo cache:$HOME/_darcs/cache > $HOME/_darcs/sources
```

Note that the cache directory should reside on the same filesystem as your repositories, so you may need to vary this. You can also use multiple cache directories on different filesystems, if you have several filesystems on which you use darcs.

On MS Windows 3)

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 repository without the `--quiet` option.

3.2 Environment variables

There are a few environment variables whose contents affect darcs’ behavior. Here is a quick list of all the variables and their documentation in the rest of the manual:

Variable	Section
DARCS_EDITOR, EDITOR, VISUAL	3.3
DARCS_PAGER, PAGER	3.3
HOME	3.3
TERM	??
DARCS_EMAIL, EMAIL	6.1
DARCS_APPLY_FOO	3.4
DARCS_GET_FOO	3.4
DARCS_MGET_FOO	3.4
DARCS_MGETMAX	3.4
DARCS_PROXYUSERPWD	3.4
DARCS_SSH	3.4
DARCS_SCP	3.4
DARCS_SFTP	??
SSH_PORT	3.4
DARCS_ALTERNATIVE_COLOR	3.5
DARCS_ALWAYS_COLOR	3.5
DARCS_DO_COLOR_LINES	3.5
DARCS_DONT_COLOR	3.5
DARCS_DONT_ESCAPE_TRAILING_CR	3.5
DARCS_DONT_ESCAPE_TRAILING_SPACES	3.5
DARCS_DONT_ESCAPE_8BIT	3.6
DARCS_DONT_ESCAPE_ANYTHING	3.6
DARCS_DONT_ESCAPE_ISPRINT	3.6
DARCS_ESCAPE_EXTRA	3.6
DARCS_DONT_ESCAPE_EXTRA	3.6

3.3 General-purpose variables

DARCS_EDITOR, DARCSEDITOR, VISUAL and EDITOR To edit a patch description of email comment, Darcs will invoke an external editor. Your preferred editor can be set as any of the environment variables `$DARCS_EDITOR`, `$DARCSEDITOR`, `$VISUAL` or `$EDITOR`. If none of these are set, `vi(1)` is used. If `vi` crashes or is not found in your `PATH`, `emacs`, `emacs -nw`, `nano` and (on Windows) `edit` are each tried in turn.

DARCS_PAGER and PAGER Darcs will sometimes invoke a pager if it deems output to be too long to fit onscreen. Darcs will use the pager specified by `$DARCS_PAGER` or `$PAGER`. If neither are set, ‘less’ will be used.

DARCS_TMPDIR and TMPDIR Darcs often creates temporary directories. For example, the ‘darcs diff’ command creates two for the working trees to be diffed. By default temporary directories are created in `/tmp`, or if that doesn’t exist, in `._darcs` (within the current repo). This can be overridden by specifying some other directory in the file `._darcs/prefs/tmpdir` or the environment variable `$DARCS_TMPDIR` or `$TMPDIR`.

DARCS_KEEP_TMPDIR If the environment variable `DARCS_KEEP_TMPDIR` is defined, darcs will not remove the temporary directories it creates. This is intended primarily for debugging Darcs itself, but it can also be useful, for example, to determine why your test preference (see ‘darcs setpref’) is failing when you run ‘darcs record’, but working when run manually.

HOME and APPDATA Per-user preferences are set in `$HOME/._darcs` (on Unix) or `%APPDATA%/._darcs` (on Windows). This is also the default location of the cache.

3.4 Remote repositories

DARCS_SSH Repositories of the form `[user@]host:[dir]` are taken to be remote repositories, which Darcs accesses with the external program `ssh(1)`.

The environment variable `$DARCS_SSH` can be used to specify an alternative SSH client. Arguments may be included, separated by whitespace. The value is not interpreted by a shell, so shell constructs cannot be used; in particular, it is not possible for the program name to contain whitespace by using quoting or escaping.

DARCS_SCP and DARCS_SFTP When reading from a remote repository, Darcs will attempt to run ‘darcs transfer-mode’ on the remote host. This will fail if the remote host only has Darcs 1 installed, doesn’t have Darcs installed at all, or only allows SFTP.

If transfer-mode fails, Darcs will fall back on `scp(1)` and `sftp(1)`. The commands invoked can be customized with the environment variables `$DARCS_SCP` and `$DARCS_SFTP` respectively, which behave like `$DARCS_SSH`. If the remote end allows only sftp, try setting `DARCS_SCP=sftp`.

SSH_PORT If this environment variable is set, it will be used as the port number for all SSH calls made by Darcs (when accessing remote repositories over SSH). This is useful if your SSH server does not run on the default port, and your SSH client does not support `ssh_config(5)`. OpenSSH users will probably prefer to put something like ‘Host *.example.net Port 443’ into their `./ssh/config` file.

HTTP_PROXY, HTTPS_PROXY, FTP_PROXY, ALL_PROXY and NO_PROXY If Darcs was built with libcurl, the environment variables HTTP_PROXY, HTTPS_PROXY and FTP_PROXY can be set to the URL of a proxy in the form

```
[protocol://]host[:port]
```

In which case libcurl will use the proxy for the associated protocol (HTTP, HTTPS and FTP). The environment variable ALL_PROXY can be used to set a single proxy for all libcurl requests.

If the environment variable NO_PROXY is a comma-separated list of host names, access to those hosts will bypass proxies defined by the above variables. For example, it is quite common to avoid proxying requests to machines on the local network with

```
NO_PROXY=localhost,*.localdomain
```

For compatibility with lynx et al, lowercase equivalents of these environment variables (e.g. \$http_proxy) are also understood and are used in preference to the uppercase versions.

If Darcs was not built with libcurl, all these environment variables are silently ignored, and there is no way to use a web proxy.

DARCS_PROXYUSERPWD If Darcs was built with libcurl, and you are using a web proxy that requires authentication, you can set the \$DARCS_PROXYUSERPWD environment variable to the username and password expected by the proxy, separated by a colon. This environment variable is silently ignored if Darcs was not built with libcurl.

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 -
```

Apart from such toy examples, it is likely that you will need to manipulate the argument before passing it to the actual fetcher program. For example, consider the problem of getting read access to a repository on a CIFS (SMB) share without mount privileges:

```
export DARCS_GET_SMB="smbclient -c get"
darcs get smb://fs/twb/Desktop/hello-world
```

The above command will not work for several reasons. Firstly, Darcs will pass it an argument beginning with ‘smb:’, which smbclient does not understand. Secondly, the host and share ‘//fs/twb’ must be presented as a separate argument to the path ‘Desktop/hello-world’. Thirdly, smbclient requires that ‘get’ and the path be a single argument (including a space), rather than two separate arguments. Finally, smbclient’s ‘get’ command writes the file to disk, while Darcs expects it to be printed to standard output.

In principle, we could get around such problems by making the variable contain a shell script, e.g.

```
export DARCS_GET_SMB='sh -c "...; smbclient $x -c \"get $y\"'
```

Unfortunately, Darcs splits the command on whitespace and does not understand that quotation or escaping, so there is no way to make Darcs pass the text after ‘-c’ to sh as a single argument. Therefore, we instead need to put such one-liners in separate, executable scripts.

Continuing our smbclient example, we create an executable script `~/ .darcs/libexec/get_smb` with the following contents:

```
#!/bin/bash -e
IFS=/ read host share file <<<"${1#smb://}"
smbclient //$host/$share -c "get $file -"
```

And at last we can say

```
export DARCS_GET_SMB=~/ .darcs/libexec/get_smb
darcs get smb://fs/twb/Desktop/hello-world
```

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 file name 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.

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

3.5 Highlighted output

If the terminal understands ANSI color escape sequences, 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 understand ANSI colors, like `less -R`, darcs can be forced always to 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. In addition, there is an extra-colorful mode, which is not enabled by default, which can be activated with `DARCS.DO_COLOR_LINES`.

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`.

3.6 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 system's 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 currently does not work on some architectures if darcs is compiled with GHC 6.4 or later. 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 Latin-1 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 4

Best practices

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.

4.1 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 systems use the term *changeset* instead of *patch*).¹ Darcs will create and store these patches based on the changes you make in *working*.

4.1.1 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

¹If 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`.

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.

4.1.2 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).

4.1.3 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 **recorded**. 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.

Keep in mind that the patches are your history, so deleting them with **unrecord** makes it impossible to track what changes you *really* made. Re-doing 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.

²Revert 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**.

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.

4.1.4 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 use **remove**, which cancels out the addfile in pending.

4.2 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 patches depended upon are there. Patches

³In the file `.darcs/patches/pending`.

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.

4.2.1 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.

4.2.2 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 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.

⁴Actually 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.

4.2.3 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 **obliterate**. 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 **obliterate** just erases the patch. Beware that if you obliterate *all* copies of a patch it is completely lost—forever. Therefore you should work with branches when you obliterate patches. The **obliterate** command can wisely be disabled in a dedicated main repository by adding **obliterate 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.

4.2.4 Tags—versions

While **pull** and **obliterate** 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.

4.2.5 Conflicts

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

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

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

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.

If you are using a darcs-2 repository (Section 6.4.1), darcs does *not* consider two patches making the *same* change to be a conflict, much in the same fashion as other version control systems. (The caveat here is two non-identical patches with some identical changes may conflict. For the most part, darcs should just do what you expect).

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 parts⁷ 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 **mark-conflicts**. 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 **mark-conflicts** 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 **mark-conflicts**.

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.

mark-conflicts 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, **mark-conflicts** will pick just one of them, and no delimiters are inserted. So watch out if darcs tells you about a conflict.

mark-conflicts 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, **obliterate** and **get** don’t.

⁷The primitive patches making up the total patch.

4.2.6 Resolving conflicts

A conflict is resolved (not marked, as with the command `mark-conflicts`) 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 `mark-conflicts` (or `pull`). But it can just as well be a tag. So don't forget to fix conflicts before you accidentally "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 **obliterate** 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.

4.3 Use a Global Cache

When working with darcs 2 it is recommended to use a global cache, as this is one of the biggest performance enhancing tools of darcs 2. The global cache acts as a giant patch pool where darcs first looks for a patch when grabbing new patches. This saves time by not downloading the same patch twice from a remote server. It also saves space by storing the patch only once, if you ensure your cache and your repositories are on the same hardlink-supporting filesystem.

Darcs now enables a global patch cache under your home directory by default. Older darcs 2.x versions required this manual step:

```
$ mkdir -p $HOME/.darcs/cache
$ echo cache:$HOME/.darcs/cache > $HOME/.darcs/sources
```

On MS Windows 3, using `cmd.exe` (Command Prompt under Accessories):

```
> md "%UserProfile%\Application Data\darcs\cache" (notice double quotes!)
> echo cache:%UserProfile%\Application Data\darcs\cache > "%UserProfile%\Application Data\darcs\
```

There are some other advanced things you can do in `_darcs/prefs/sources`, such as create per-repository caches, read-only caches and even set a primary source repository above any used in a `darcs get` or `darcs pull` command.

4.4 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 \
    --reporid /path/to/repo --verify /path/to/allowed_keys)
```

This causes darcs apply to be run on any email 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 \
    --reporid ~/darcs)"
```

4.5 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 4.4, 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 does he **push** to the central repository.

4.6 Personal development

It's easy to have several personal development trees using darcs, even when working on a team or with shared code. The number and method of using each personal tree is limited only by such grand limitations as: your disk space, your imagination, available time, etc.

For example, if the central darcs repository for your development team is R_c , you can create a local working directory for feature f_1 . This local working directory contains a full copy of R_c (as of the time of the “darcs get” operation) and can be denoted R_1 . In the midst of working on feature f_1 , you realize it requires the development of a separate feature f_2 . Rather than intermingling f_1 and f_2 in the same working tree, you can create a new working tree for working on f_2 , where that working tree contains the repository R_2 .

While working on f_2 , other developers may have made other changes; these changes can be retrieved on a per-patch selection basis by periodic “darcs pull” operations.

When your work on f_2 is completed, you can publish it for the use of other developers by a “darcs push” (or “darcs send”) from R_2 to R_c . Independently of the publishing of f_2 , you can merge your f_2 work into your f_1 working tree by a “darcs pull R_2 ” in the R_1 development tree (or “darcs push” from R_2 to R_1).

When your work on f_1 is completed, you publish it as well by a “darcs push” from R_1 to R_c .

Your local feature development efforts for f_1 or f_2 can each consist of multiple patches. When pushing or pulling to other trees, these patches can either all be selected or specific patches can be selected. Thus, if you introduce a set of debugging calls into the code, you can commit the debugging code in a distictly separate patch (or patches) that you will not push to R_c .

4.6.1 Private patches

As discussed in the section above, a developer may have various changes to their local development repositories that they do not ever wish to publish to a group repository (e.g. personal debugging code), but which they would like to keep in their local repository, and perhaps even share amongst their local repositories.

This is easily done via darcs, since those private changes can be committed in patches that are separate from other patches; during the process of pushing patches to the common repository (R_c), the developer is queried for which

patches should be moved to (R_c) on a patch-by-patch basis.

The `--complement` flag for the “darcs pull” operation can further simplify this effort. If the developer copies (via “darcs push” or “darcs pull”) all private patches into a special repository/working tree (R_p), then those patches are easily disregarded for pulling by adding `--complement` to the “darcs pull” line and listing R_p after the primary source repository.

The `--complement` flag is only available for “darcs pull”, and not “darcs push” or “darcs send”, requiring the user to have pull access to the target repository. While the actual public repository is often not shared in this manner, it’s simple to create a local version of the public repository to act as the staging area for that public repository.

The following example extends the two feature addition example in the previous section using a local staging repository (R_l) and a private patch repository:

```
$ cd working-dir
$ darcs get http://server/repos/Rc R1

$ darcs get R1 R1
$ cd R1
...development of f1
$ darcs record -m'p1: f1 initial work'
...
$ darcs record -m'p2: my debugging tracepoints'
...

$ cd ..
$ darcs get http://server/repos/Rc R2
$ cd R2
$ darcs pull -p p2 ../R1
... development of f2
$ darcs record -m'p3: f2 finished'

$ cd ..
$ darcs get R1 Rp
$ cd Rp
$ darcs pull -p p2 ../R2

$ cd ../R1
$ darcs pull --complement ../R2 ../Rp
$ darcs send
... for publishing f2 patches to Rc

$ cd ../R1
$ darcs pull ../R2
... updates R1 with f2 changes from R2
... more development of f1
```



```
$ darcs record -m'p4: f1 feature finished.'
```

```
$ cd ../R1
```

```
$ darcs pull --complement ../R1 ../Rp
```

```
$ darcs send
```


Chapter 5

Repository Formats

Darcs 2 supports three repository formats:

- The current format, called ‘darcs-2’. It has the most features.
- The original format, called ‘darcs-1’ or ‘old-fashioned inventory’. It is the only format supported by Darcs releases prior to 2.0. Patches cannot be shared between darcs-2 and darcs-1 repositories.
- An intermediary format, called ‘hashed’. It provides some of the features of darcs-2, while retaining the ability to exchange patches with darcs-1. Patches cannot be shared between darcs-2 and hashed repositories.

Note that references to the hashed format refer to hashed darcs-1 repositories. All darcs-2 repositories are also hashed, and ‘darcs show repo’ in a darcs-2 repository will report the format as ‘hashed, darcs-2’.

The hashed (and darcs-2) format improves on the darcs-1 format in the following ways:

- Improved atomicity of operations. This improves the safety and efficiency of those operations.
- File names in the ‘pristine’ copy of the working tree are hashed. This greatly reduces the chance of a program accidentally treating files in `_darcs/pristine` as part of the working tree.

For example, in darcs-1 running `find -name "*.gif" -delete` instead of `find -name _darcs -prune -o -name "*.gif" -delete` would result in GIF files in pristine files being deleted.

- Support for ‘lazy’ repositories. This causes patches to be copied from the parent repository on demand, rather than during the initial ‘darcs get’. For repositories that only operate on recent patches (such as feature branches), this reduces the repository’s size, and the time and bandwidth taken to create it.

Lazy repositories are first-class repositories, and all operations are supported as long as the parent repository remains accessible. This isn't the case for the 'partial' feature of the darcs-1 format.

- Support for caches that are shared by all repositories. When operating on many similar repositories, the patches and pristine files they have in common are hard links to the cache. This greatly reduces storage requirements, and the time and bandwidth needed to make similar repositories.

The darcs-2 format improves on the hashed format in the following ways:

- The 'exponential merge' problem is *far* less likely to occur (it can still be produced in deliberately pathological cases).
- Identical primitive changes no longer conflict. For example, if two patches both attempt to add a directory 'tests', these patches will not conflict.

See also 'darcs initialize' (6.4.1), 'darcs convert' (6.11.1) and 'darcs get' (6.4.2).

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.

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

²As for the other end, see apply

affects	patches	working directory
record	yes	no
unrecord	yes	no
rollback	yes	yes
revert	no	yes
unrevert	no	yes
pull	yes	yes
obliterate	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 `obliterate`, `unpull`, `unrecord` or `amend-record`.

--verbose, --quiet, --normal-verbosity

Most commands also accept the **--verbose** option, which tells darcs to provide additional output. The amount of verbosity varies from command to command. Commands that accept **--verbose\verb** also accept **--quiet\verb**, which suppresses non-error output, and **--normal-verbosity\verb** which can be used to restore the default verbosity if **--verbose** or **--quiet** is in the defaults file.

--debug, --debug-http

Many commands also accept the `--debug` option, which causes darcs to generate additional output that may be useful for debugging its behavior, but which otherwise would not be interesting. Option `--debug-http` makes darcs output debugging info for libcurl.

`--repor`

Another common option is the `--repor` 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 repository directory when calling the command. This is useful when running darcs in a pipe, as might be the case when running `apply` from a mailer.

`--remote-repo`

Some commands, such as `pull` require a remote repository to be specified, either from the command line or as a default. The `--remote-repo` provides an alternative way to supply this remote repository path. This flag can be seen as temporarily “replacing” the default repository. Setting it causes the command to ignore the default repository (it also does not affect, i.e. overwrite the default repository). On the other hand, if any other repositories are supplied as command line arguments, this flag will be ignored (and the default repository may be overwritten).

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.

When a patch or a group of patches is selected, all patches they depend on get silently selected too. For example: `darcs pull --patches bugfix` means “pull all the patches with ‘bugfix’ in their name, along with any patches they require.” If you really only want patches with ‘bugfix’ in their name, you should

use the `--no-deps` option, which makes darcs exclude any matched patches from the selection which have dependencies that are themselves not explicitly matched by the selection.

For `unrecord`, `unpull` and `obliterate`, patches that depend on the selected patches are silently included, or if `--no-deps` is used selected patches with dependencies on not selected patches are excluded from the selection.

Match Currently `--match` accepts six 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.

In this and the other match types, the argument must be enclosed in double quotes if it contains spaces. You can escape a quote in the argument with a backslash; backslash escapes itself, but it is treated literally if followed by a character other than a double quote or backslash, so it is typically not necessary to escape a backslash. No such escaping is necessary unless the argument is enclosed in double quotes.

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

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

Note that to match regexp metacharacters, such as `(`, literally, they must be escaped with backslash along with any embedded double quotes. To match a literal backslash it must be written quadrupled in general, but often it need not be escaped, since backslash is only special in regexps when followed by a metacharacter. In the following example pairs, the first literal is matched by the second sequence in the match name: `"": "\"`, `"\": "\\\""`, `"\x": "\x"`, `"(": "\("`.

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

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

Note you need to provide the full hash string as above. 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 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 14:00"'
darcs annotate --summary --match 'date "tea time yesterday"'
darcs annotate --summary --match 'date "3 days before last year at 17:00"'
darcs changes --from-match 'date "Sat Jun 30 11:31:30 EDT 2004"'
```

Only English date specifications are supported—specifically you must use English day and month names. Also, only a limited set of time zones is supported (compatible with GNU coreutils’ date parsing). Unrecognized zones are treated as UTC, which may result in the timestamps printed in change entries not being recognized by the date matching. You can avoid this problem on a POSIX-like system by running darcs in the UTC zone to get the times initially, e.g.:

```
TZ=UTC darcs changes
```

When matching on the ISO format, a partial date is treated as a range. English dates can either refer to a specific day (“6 months ago”, “day before yesterday”), or to an interval from some past date (“last month”) to the present. Putting this all together, if today is “2004-07-24” then the following matches should work:

date	patches selected
2004	from 2004-01-01 up to and including 2004-12-31
2004-01	from 2004-01-01 up to and including 2004-01-31
2004-01-01	during 2004-01-01
today	during 2004-07-24 (starting midnight in your timezone)
yesterday	during 2004-07-23
6 months ago	during 2004-01-23
last 6 months	since 2004-01-23
last month	since 2004-06-23 (not 2004-06-01!)
last week	since 2004-07-16

For more precise control, you may specify an interval, either in a small subset of English or of the ISO 8601 format⁴. If you use the ISO format, note that durations, when specified alone, are interpreted as being relative to the current date and time.

```
darcs annotate --summary --match 'date "between 2004-03-12 and last week"'
darcs annotate --summary --match 'date "after 2005"'
darcs annotate --summary --match 'date "in the last 3 weeks"'
darcs annotate --summary --match 'date "P3M/2006-03-17"'
darcs annotate --summary --match 'date "2004-01-02/2006-03-17"'
darcs annotate --summary --match 'date "P2M6D"'
```

⁴<http://www.w3.org/TR/NOTE-datetime>

You may also prefer to combine date matching with a more specific pattern.

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

The sixth match type accepts a regular expression which is checked against file paths that the patch touches. The syntax is

```
darcs annotate --summary --match 'touch foo/bar.c'
```

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 repository 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@abridgegar.com>. 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 ???. Or, you could specify the author on a per-repository basis using the `_darcs/prefs/author` file as described in section 3.1.

Also, a global author file can be created in your home directory with the name `.darcs/author`, on MS Windows 3. 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.

--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'
```

To use `tortoiseMerge`, you can use

```
--external-merge 'tortoiseMerge /base:"%a" /mine:"%1" /theirs:"%2" /merged:"%o"'
```

(`tortoiseMerge` is a nice merge tool that comes with TortoiseSVN and works well on Windows.)

Note that the command is split into space-separated words and the first one is `execed` with the rest as arguments—it is not a shell command. In particular, on Windows this means that the first command path should not contain spaces and you should make sure the command is in your `PATH`.

The substitution of the `%` escapes is done everywhere. If you need to prevent substitution you can use a double percentage sign, i.e. `%%a` is substituted with `%a`. Here is an example script to use the Emacs' `Ediff` package for merging.

```
#!/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 $1 in
        *["\\"* ) form=$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 3.1, on MS Windows 3) a line such as

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

or

```
ALL external-merge tortoiseMerge /base:"%a" /mine:"%1" /theirs:"%2" /merged:"%o"
```

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

```
--sendmail-command
```

SENDMAIL On Unix, the ‘darcs send’ command relies on `sendmail(8)`. The ‘`--sendmail-command`’ or `$SENDMAIL` environment variable can be used to provide an explicit path to this program; otherwise the standard locations `/usr/sbin/sendmail` and `/usr/lib/sendmail` will be tried.

```
--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.

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

These options control prompting before running the posthook. Use `--prompt-posthook` to have darcs prompt before running the posthook command. You may use `--run-posthook` to reenable the default behavior of running user-specified posthooks.

Some darcs commands export to the posthook command information about the changes being made. In particular, three environment variables are defined. `DARCS_PATCHES` contains a human-readable summary of the patches being acted upon. The format is the same as “darcs changes”. `DARCS_PATCHES_XML` Contains the same details, in the same XML format as “darcs changes”. Finally, `DARCS_FILES` contains a list of the files affected, one file per line. If your repository has filenames including newlines, you’ll just have to cope. Note, however, that *none* of these environment variables are defined when running under windows. Note also that we refuse to pass environment variables greater in size than 10k, in order to avoid triggering E2BIG errors.

```
--prehook=COMMAND, --no-prehook
```

To provide a command that should be run before a darcs command is executed, use **--prehook** to specify the command. An example use is for people who want to have a command run whenever a patch is to be recorded, such as translating line endings before recording patches. Using **--no-prehook** will disable running the command.

--run-prehook, --prompt-prehook

These options control prompting before running the prehook. See the posthook documentation above for details.

--ssh-cm, --no-ssh-cm

For commands which invoke ssh, darcs will normally multiplex ssh sessions over a single connection as long as your version of ssh has the ControlMaster feature from OpenSSH versions 3.9 and above. This option will avoid darcs trying to use this feature even if your ssh supports it.

--http-pipelining, --no-http-pipelining

When compiled with libcurl (version 7.18.0 and above), darcs can use HTTP pipelining. It is enabled by default for libcurl (version 7.19.1 and above). This option will make darcs enable or disable HTTP pipelining, overwriting default. Note that if HTTP pipelining is really used depends on the server.

--no-cache

Do not use patch caches.

--umask

By default, Darcs will use your current umask. The option **--umask** will cause Darcs to switch to a different umask before writing to the repository.

--dont-restrict-paths, --restrict-paths

By default darcs is only allowed to manage and modify files and directories contained inside the current repository and not being part of any darcs repository's meta data (including the current one). This is mainly for security, to protect you from spoofed patches modifying arbitrary files with sensitive data—say, in your home directory—or tampering with any repository's meta data to switch off this safety guard.

But sometimes you may want to manage a group of “sub” repositories’ preference files with a global repository, or use darcs in some other advanced way. The best way is probably to put ALL `dont-restrict-paths` in `_darcs/prefs/defaults`. This turns off all sanity checking for file paths in patches.

Path checking can be temporarily turned on with `--restrict-paths` on the command line, when pulling or applying unknown patches.

`--allow-unrelated-repos`

By default darcs checks and warns user if repositories are unrelated when doing pull, push and send. This option makes darcs skip this check.

`--just-this-repo`

This option limits the check or repair to the current repo and omits any caches or other repos listed as a source of patches.

`--check`

This option specifies checking mode.

`--repair`

This option specifies repair mode.

6.2 Options apart from darcs commands

`--help, --overview`

Calling darcs with just `--help` as an argument gives a brief summary of what commands are available. The `--overview` 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 Getting help

6.3.1 `darcs help`

Without arguments, `'darcs help'` prints a categorized list of `darcs` commands and a short description of each one. With an extra argument, `'darcs help foo'` prints detailed help about the `darcs` command `foo`.

Usage: `darcs help [OPTION]... [<DARCS_COMMAND> [DARCS_SUBCOMMAND]]`

Options:

Display help about `darcs` and `darcs` commands.

6.4 Creating repositories

6.4.1 `darcs initialize`

The `'darcs initialize'` command turns the current directory into a Darcs repository. Any existing files and subdirectories become UNSAVED changes in the working tree: record them with `'darcs add -r'` and `'darcs record'`.

When converting a project to Darcs from some other VCS, translating the full revision history to native Darcs patches is recommended. (The Darcs wiki lists utilities for this.) Because Darcs is optimized for small patches, simply importing the latest revision as a single large patch can PERMANENTLY degrade Darcs performance in your repository by an order of magnitude.

This command creates the `'_darcs'` directory, which stores version control metadata. It also contains per-repository settings in `_darcs/prefs/`, which you can read about in the user manual.

In addition to the default `'darcs-2'` format, there are two backward compatibility formats for the `_darcs` directory. These formats are only useful if some of your contributors do not have access to Darcs 2.0 or higher. In that case, you need to use the original format (called `'old-fashioned inventory'` or `'darcs-1'`) for any repositories those contributors access.

As patches cannot be shared between `darcs-2` and `darcs-1` repositories, you cannot use the `darcs-2` format for private branches of such a project. Instead, you should use the `'hashed'` format, which provides most of the features of the `darcs-2` format, while retaining the ability to share patches with `darcs-1` repositories. The `'darcs get'` command will do this by default.

Once all contributors have access to Darcs 2.0 or higher, a `darcs-1` project can be migrated to `darcs-2` using the `'darcs convert'` command.

Darcs will create a hashed repository by default when you ‘darcs get’ a repository in old-fashioned inventory format. Once all contributors have upgraded to Darcs 2.0 or later, use ‘darcs convert’ to convert the project to the darcs-2 format.

Initialize is commonly abbreviated to ‘init’.

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

Options:

<code>--hashed</code>	Some new features. Compatible with older repos
<code>--darcs-2</code>	All features. Related repos must use same format [DEFAULT]
<code>--old-fashioned-inventory</code>	Minimal features. What older repos use.
<code>--reporid DIRECTORY</code>	specify the repository directory in which to run

Make the current directory a repository.

6.4.2 darcs get

Get creates a local copy of a repository. The optional second argument specifies a destination directory for the new copy; if omitted, it is inferred from the source location.

By default Darcs will copy every patch from the original repository. This means the copy is completely independent of the original; you can operate on the new repository even when the original is inaccessible. If you expect the original repository to remain accessible, you can use `-lazy` to avoid copying patches until they are needed (‘copy on demand’). This is particularly useful when copying a remote repository with a long history that you don’t care about.

The `-lazy` option isn’t as useful for local copies, because Darcs will automatically use ‘hard linking’ where possible. As well as saving time and space, you can move or delete the original repository without affecting a complete, hard-linked copy. Hard linking requires that the copy be on the same filesystem and the original repository, and that the filesystem support hard linking. This includes NTFS, HFS+ and all general-purpose Unix filesystems (such as ext3, UFS and ZFS). FAT does not support hard links.

Darcs get will not copy unrecorded changes to the source repository’s working tree.

It is often desirable to make a copy of a repository that excludes some patches. For example, if releases are tagged then ‘darcs get `-tag .`’ would make a copy of the repository as at the latest release.

An untagged repository state can still be identified unambiguously by a context file, as generated by ‘darcs changes `-context`’. Given the name of such a file, the `-context` option will create a repository that includes only the patches from that context. When a user reports a bug in an unreleased version of your project, the recommended way to find out exactly what version they were running is to have them include a context file in the bug report.

You can also make a copy of an untagged state using the `-to-patch` or `-to-match` options, which exclude patches ‘after’ the first matching patch. Because

these options treat the set of patches as an ordered sequence, you may get different results after reordering with ‘darcs optimize’, so tagging is preferred.

If the source repository is in a legacy darcs-1 format and contains at least one checkpoint (see ‘darcs optimize’), the `-partial` option will create a partial repository. A partial repository discards history from before the checkpoint in order to reduce resource requirements. For modern darcs-2 repositories, `-partial` is a deprecated alias for the `-lazy` option.

A repository created by ‘darcs get’ will be in the best available format: it will be able to exchange patches with the source repository, but will not be directly readable by Darcs binaries older than 2.0.0. Use the ‘`-old-fashioned-inventory`’ option if the latter is required.

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

Options:

<code>--repo-name DIRECTORY</code>	<code>--reporidir DIRECTORY</code>	path of output directory
<code>--partial</code>		get partial repository using checkpoint (old-fashioned format only)
<code>--lazy</code>		get patch files only as needed
<code>--ephemeral</code>		don't save patch files in the repository
<code>--complete</code>		get a complete copy of the repository
<code>--to-match PATTERN</code>		select changes up to a patch matching PATTERN
<code>--to-patch REGEXP</code>		select changes up to a patch matching REGEXP
<code>-t --tag REGEXP</code>		select tag matching REGEXP
<code>--context FILENAME</code>		version specified by the context in FILENAME
<code>--set-default</code>		set default repository [DEFAULT]
<code>--no-set-default</code>		don't set default repository
<code>--set-scripts-executable</code>		make scripts executable
<code>--dont-set-scripts-executable</code>		don't make scripts executable
<code>--nolinks</code>		do not link repository or pristine to sibling
<code>--hashed</code>		Convert darcs-1 format to hashed format
<code>--old-fashioned-inventory</code>		Convert from hashed to darcs-1 format

Advanced options:

<code>--ssh-cm</code>	use SSH ControlMaster feature
<code>--no-ssh-cm</code>	don't use SSH ControlMaster feature [DEFAULT]
<code>--http-pipelining</code>	enable HTTP pipelining
<code>--no-http-pipelining</code>	disable HTTP pipelining [DEFAULT]
<code>--no-cache</code>	don't use patch caches

Create a local copy of a repository.

6.4.3 darcs put

The ‘darcs put’ command creates a copy of the current repository. It is currently very inefficient, so when creating local copies you should use ‘`darcs get . x`’ instead of ‘`darcs put x`’.

Currently this command just uses ‘darcs init’ to create the target repository, then ‘darcs push -all’ to copy patches to it. Options passed to ‘darcs put’ are passed to the init and/or push commands as appropriate. See those commands for an explanation of each option.

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

Options:

<code>--to-match PATTERN</code>	select changes up to a patch matching PATTERN
<code>--to-patch REGEXP</code>	select changes up to a patch matching REGEXP
<code>-t --tag REGEXP</code>	select tag matching REGEXP
<code>--context FILENAME</code>	version specified by the context in FILENAME
<code>--set-scripts-executable</code>	make scripts executable
<code>--dont-set-scripts-executable</code>	don't make scripts executable
<code>--hashed</code>	Convert darcs-1 format to hashed format
<code>--old-fashioned-inventory</code>	Convert from hashed to darcs-1 format
<code>--set-default</code>	set default repository [DEFAULT]
<code>--no-set-default</code>	don't set default repository
<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Advanced options:

<code>--apply-as USERNAME</code>	apply patch as another user using sudo
<code>--apply-as-myself</code>	don't use sudo to apply as another user [DEFAULT]
<code>--ssh-cm</code>	use SSH ControlMaster feature
<code>--no-ssh-cm</code>	don't use SSH ControlMaster feature [DEFAULT]
<code>--http-pipelining</code>	enable HTTP pipelining
<code>--no-http-pipelining</code>	disable HTTP pipelining [DEFAULT]
<code>--no-cache</code>	don't use patch caches

Makes a copy of the repository

6.5 Modifying the contents of a repository

6.5.1 darcs add

Generally a repository contains both files that should be version controlled (such as source code) and files that Darcs should ignore (such as executables compiled from the source code). The ‘darcs add’ command is used to tell Darcs which files to version control.

When an existing project is first imported into a Darcs repository, it is common to run ‘darcs add -r *’ or ‘darcs record -l’ to add all initial source files into darcs.

Adding symbolic links (symlinks) is not supported.

Darcs will ignore all files and folders that look ‘boring’. The `-boring` option overrides this behaviour.

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.

Options:

Advanced options:

6.5.2 darcs remove

Note that applying a removal patch to a repository (e.g. by pulling the patch) will ALWAYS affect the working tree of that repository.

Options:

Advanced options:

--umask UMASK	specify umask to use when writing	Remove
files from version control.		

6.5.3 darcs move

Darcs cannot reliably distinguish between a file being deleted and a new one added, and a file being moved. Therefore Darcs always assumes the former, and provides the ‘darcs mv’ command to let Darcs know when you want the latter. This command will also move the file in the working tree (unlike ‘darcs remove’).

Darcs will not rename a file if another file in the same folder has the same name, except for case. The `-case-ok` option overrides this behaviour. Windows and OS X usually use filesystems that do not allow files a folder to have the same name except for case (for example, ‘ReadMe’ and ‘README’). If `-case-ok` is used, the repository might be unusable on those systems!

Usage: `darcs move [OPTION] ... <SOURCE> ... <DESTINATION>`

Options:

<code>--case-ok</code>	don't refuse to add files differing only in case
<code>--reserved-ok</code>	don't refuse to add files with Windows-reserved names
<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Advanced options:

<code>--umask UMASK</code>	specify umask to use when writing	Move
----------------------------	-----------------------------------	------

or rename files.

6.5.4 darcs replace

In addition to line-based patches, Darcs supports a limited form of lexical substitution. Files are treated as sequences of words, and each occurrence of the old word is replaced by the new word. This is intended to provide a clean way to rename a function or variable. Such renamings typically affect lines all through the source code, so a traditional line-based patch would be very likely to conflict with other branches, requiring manual merging.

Files are tokenized according to one simple rule: words are strings of valid token characters, and everything between them (punctuation and whitespace) is discarded.

The tokenizer treats files as byte strings, so it is not possible for `-token-chars` to include multi-byte characters, such as the non-ASCII parts of UTF-8. Similarly, trying to replace a ‘high-bit’ character from a unibyte encoding will also result in replacement of the same byte in files with different encodings. For example, an acute a from ISO 8859-1 will also match an alpha from ISO 8859-7.

By default, valid token characters are letters, numbers and the underscore (i.e. `[A-Za-z0-9_]`). However if the old and/or new token contains either a hyphen or period, BOTH hyphen and period are treated as valid by default (i.e. `[A-Za-z0-9_.-]`).

The set of valid characters can be customized using the `-token-chars` option. The argument must be surrounded by square brackets. If a hyphen occurs between two characters in the set, it is treated as a set range. For example, in most locales `[A-Z]` denotes all uppercase letters. If the first character is a

caret, valid tokens are taken to be the complement of the remaining characters. For example, `[^ \n\t]` declares all characters except the space, tab and newline as valid within a word. Unlike the `tr(1)` and `grep(1)` utilities, character classes (such as `[[[:alnum:]]]`) are NOT supported.

If you choose to use `-token-chars`, you are **STRONGLY** encouraged to do so consistently. The consequences of using multiple replace patches with different `-token-chars` arguments on the same file are not well tested nor well understood.

By default Darcs will refuse to perform a replacement if the new token is already in use, because the replacements would be not be distinguishable from the existing tokens. This behaviour can be overridden by supplying the `-force` option, but an attempt to ‘darcs rollback’ the resulting patch will affect these existing tokens.

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

Options:

<code>--token-chars "[CHARS]"</code>	define token to contain these characters	
<code>-f --force</code>	proceed with replace even if ‘new’ token	
<code>--no-force</code>	already exists	
<code>--reporid DIRECTORY</code>	don’t force the replace if it looks scary	
	specify the repository directory in which to	
	run	

Advanced options:

<code>--ignore-times</code>	don’t trust the file modification times	
<code>--umask UMASK</code>	specify umask to use when writing	Sub-

stitute one word for another. 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.

6.6 Working with changes

6.6.1 darcs record

Record is used to name a set of changes and record the patch to the repository.

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

Options:

-m	--patch-name PATCHNAME	name of patch
-A	--author EMAIL	specify author id
	--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
-a	--all	answer yes to all patches
	--pipe	ask user interactively for the patch metadata
-i	--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
-l	--look-for-adds	look for (non-boring) files that could be added
	--dont-look-for-adds	don't look for any files that could be added
		[DEFAULT]
	--repodir DIRECTORY	specify the repository directory in which to run

Advanced options:

--logfile FILE	give patch name and comment in file
--delete-logfile	delete the logfile when done
--compress	create compressed patches
--dont-compress	don't create compressed patches
--ignore-times	don't trust the file modification times
--umask UMASK	specify umask to use when writing
--set-scripts-executable	make scripts executable
--dont-set-scripts-executable	don't make scripts executable

Save changes in the working copy to the repository as a patch.

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 repository. 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.

It is possible to record a patch which has no actual changes but which has specific dependencies. This type of patch can be thought of as a “partial tag”. The **darcs tag** command will record a patch with no actual changes but which depends on the entire current inventory of the repository. The **darcs record --ask-deps** with no selected changes will record a patch that depends on only those patches selected via the **--ask-deps** operation, resulting in a patch which describes a set of patches; the presence of this primary patch in a repository implies the presence of (at least) the depended-upon patches.

--no-test, --test

If you configure darcs to run a test suite, darcs will run this test on the recorded repository 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.

--set-scripts-executable

If you pass `--set-scripts-executable` to `darcs record`, `darcs` will set scripts executable in the test directory before running the test.

`--pipe`

If you run `record` with the `--pipe` option, you will be prompted for the patch date, author, and the long comment. The long comment will extend until the end of file or stdin is reached (ctrl-D on Unixy systems, ctrl-Z on systems running a Microsoft OS).

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. Here's an example of what the `--pipe` prompts look like:

```
What is the date? Mon Nov 15 13:38:01 EST 2004
Who is the author? David Roundy
What is the log? One or more comment lines
```

`--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.

6.6.2 darcs pull

Pull is used to bring changes made in another repository into the current repository (that is, either the one in the current directory, or the one specified with the `--reporidir` option). Pull allows you to bring over all or some of the patches that are in that repository but not in this one. Pull accepts arguments, which are URLs 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.

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

Options:

<code>--matches PATTERN</code>	select patches matching PATTERN
<code>-p --patches REGEXP</code>	select patches matching REGEXP
<code>-t --tags REGEXP</code>	select tags matching REGEXP
<code>-a --all</code>	answer yes to all patches
<code>-i --interactive</code>	prompt user interactively
<code>--mark-conflicts</code>	mark conflicts [DEFAULT]
<code>--allow-conflicts</code>	allow conflicts, but don't mark them
<code>--dont-allow-conflicts</code>	fail on patches that create conflicts
<code>--external-merge COMMAND</code>	use external tool to merge conflicts
<code>--test</code>	run the test script
<code>--no-test</code>	don't run the test script
<code>--dry-run</code>	don't actually take the action
<code>--xml-output</code>	generate XML formatted output
<code>-s --summary</code>	summarize changes
<code>--no-summary</code>	don't summarize changes
<code>--no-deps</code>	don't automatically fulfill dependencies
<code>--dont-prompt-for-dependencies</code>	don't ask about patches that are depended on
<code>--prompt-for-dependencies</code>	prompt about patches that are depended on
<code>--set-default</code>	set default repository [DEFAULT]
<code>--no-set-default</code>	don't set default repository
<code>--reporidir DIRECTORY</code>	specify the repository directory in which to run
<code>--ignore-unrelated-repos</code>	do not check if repositories are unrelated

Advanced options:

<code>--intersection</code>	take intersection of all repositories
<code>--union</code>	take union of all repositories [DEFAULT]
<code>--complement</code>	take complement of repositories (in order listed)
<code>--compress</code>	create compressed patches
<code>--dont-compress</code>	don't create compressed patches
<code>--nolinks</code>	do not link repository or pristine to sibling
<code>--ignore-times</code>	don't trust the file modification times
<code>--remote-repo URL</code>	specify the remote repository URL to work with
<code>--set-scripts-executable</code>	make scripts executable
<code>--dont-set-scripts-executable</code>	don't make scripts executable
<code>--umask UMASK</code>	specify umask to use when writing
<code>--restrict-paths</code>	don't allow darcs to touch external files or repo metadata
<code>--dont-restrict-paths</code>	allow darcs to modify any file or directory (unsafe)
<code>--ssh-cm</code>	use SSH ControlMaster feature
<code>--no-ssh-cm</code>	don't use SSH ControlMaster feature [DEFAULT]
<code>--http-pipelining</code>	enable HTTP pipelining
<code>--no-http-pipelining</code>	disable HTTP pipelining [DEFAULT]
<code>--no-cache</code>	don't use patch caches

Copy and apply patches from another repository to this one.

`--intersection, --union [DEFAULT], --complement`

If you provide more than one repository as an argument to pull, darcs' behavior is determined by the presence of the `--complement`, `--intersection`, and `--union` flags.

- The default (`--union`) behavior is to pull any patches that are in any of the specified repositories ($R_1 \cup R_2 \cup R_3 \dots$).
- If you instead specify the `--intersection` flag, darcs will only pull those patches which are present in all source repositories ($R_1 \cap R_2 \cap R_3 \dots$).
- If you specify the `--complement` flag, darcs will only pull elements in the first repository that do not exist in any of the remaining repositories⁵ ($R_1 \setminus (R_2 \cup R_3 \cup \dots)$).

`--external-merge`

⁵The first thing darcs will do is remove duplicates, keeping only the first specification. This is noticeable for the complement operation, since mathematically $S \setminus S \rightarrow \emptyset$, one would expect that “`darcs pull --complement repo1 repo1`” would result in no pulls, but the duplicate elimination removes the second `repo1`, reducing the above to effectively “`darcs pull repo1`”. The expected functionality could be seen via “`darcs get -a repo1 repo2; darcs pull --complement repo1 repo2`”, but there are easier ways of doing nothing!

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

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

The `--patches`, `--matches`, `--tags`, and `--no-deps` options can be used to select which patches to pull, as described in subsection 6.1.

`--no-test, --test`

If you specify the `--test` option, pull will run the test (if a test exists) on a scratch copy of the repository 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 repository lacks. Thus, the following syntax can be used to show you all the patch differences between two repositories:

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

6.6.3 darcs push

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

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

Options:

	<code>--matches PATTERN</code>	select patches matching PATTERN
<code>-p</code>	<code>--patches REGEXP</code>	select patches matching REGEXP
<code>-t</code>	<code>--tags REGEXP</code>	select tags matching REGEXP
	<code>--no-deps</code>	don't automatically fulfill dependencies
	<code>--dont-prompt-for-dependencies</code>	don't ask about patches that are depended on by matched patches (with <code>-match</code> or <code>-patch</code>)
	<code>--prompt-for-dependencies</code>	prompt about patches that are depended on by matched patches [DEFAULT]
<code>-a</code>	<code>--all</code>	answer yes to all patches
<code>-i</code>	<code>--interactive</code>	prompt user interactively
	<code>--sign</code>	sign the patch with your gpg key
	<code>--sign-as KEYID</code>	sign the patch with a given keyid
	<code>--sign-ssl IDFILE</code>	sign the patch using openssl with a given private key
	<code>--dont-sign</code>	don't sign the patch
	<code>--dry-run</code>	don't actually take the action
	<code>--xml-output</code>	generate XML formatted output
<code>-s</code>	<code>--summary</code>	summarize changes
	<code>--no-summary</code>	don't summarize changes
	<code>--reporidir DIRECTORY</code>	specify the repository directory in which to run
	<code>--set-default</code>	set default repository [DEFAULT]
	<code>--no-set-default</code>	don't set default repository
	<code>--ignore-unrelated-repos</code>	do not check if repositories are unrelated

Advanced options:

<code>--apply-as USERNAME</code>	apply patch as another user using sudo
<code>--apply-as-myself</code>	don't use sudo to apply as another user [DEFAULT]
<code>--nolinks</code>	do not link repository or pristine to sibling
<code>--remote-repo URL</code>	specify the remote repository URL to work with
<code>--ssh-cm</code>	use SSH ControlMaster feature
<code>--no-ssh-cm</code>	don't use SSH ControlMaster feature [DEFAULT]
<code>--http-pipelining</code>	enable HTTP pipelining
<code>--no-http-pipelining</code>	disable HTTP pipelining [DEFAULT]
<code>--no-cache</code>	don't use patch caches

Copy and apply patches from this repository to another one.

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 3.4). 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, --no-deps

The **--patches**, **--matches**, **--tags**, and **--no-deps** options can be used to select which patches to push, as described in subsection 6.1.

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 repository 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 repository 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.

6.6.4 darcs send

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.

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

Options:

<code>--matches PATTERN</code>	select patches matching PATTERN
<code>-p --patches REGEXP</code>	select patches matching REGEXP
<code>-t --tags REGEXP</code>	select tags matching REGEXP
<code>--no-deps</code>	don't automatically fulfill dependencies
<code>--dont-prompt-for-dependencies</code>	don't ask about patches that are depended on by matched patches (with <code>-match</code> or <code>-patch</code>)
<code>--prompt-for-dependencies</code>	prompt about patches that are depended on by matched patches [DEFAULT]
<code>-a --all</code>	answer yes to all patches
<code>-i --interactive</code>	prompt user interactively
<code>--from EMAIL</code>	specify email address
<code>-A --author EMAIL</code>	specify author id
<code>--to EMAIL</code>	specify destination email
<code>--cc EMAIL</code>	mail results to additional EMAIL(s).
<code>--subject SUBJECT</code>	Requires <code>-reply</code> specify mail subject
<code>--in-reply-to EMAIL</code>	specify in-reply-to header
<code>-o --output FILE</code>	specify output filename
<code>-O --output-auto-name[=DIRECTORY]</code>	output to automatically named file in DIRECTORY, default: current directory
<code>--sign</code>	sign the patch with your gpg key
<code>--sign-as KEYID</code>	sign the patch with a given keyid
<code>--sign-ssl IDFILE</code>	sign the patch using openssl with a given private key
<code>--dont-sign</code>	don't sign the patch
<code>--dry-run</code>	don't actually take the action
<code>--xml-output</code>	generate XML formatted output
<code>-s --summary</code>	summarize changes
<code>--no-summary</code>	don't summarize changes
<code>--edit-description</code>	edit the patch bundle description
<code>--dont-edit-description</code>	don't edit the patch bundle description
<code>--set-default</code>	set default repository [DEFAULT]
<code>--no-set-default</code>	don't set default repository
<code>--repodir DIRECTORY</code>	specify the repository directory in which to run
<code>--sendmail-command COMMAND</code>	specify sendmail command
<code>--ignore-unrelated-repos</code>	do not check if repositories are unrelated

Advanced options:

<code>--logfile FILE</code>	give patch name and comment in file
<code>--delete-logfile</code>	delete the logfile when done
<code>--remote-repo URL</code>	specify the remote repository URL to work with
<code>--context FILENAME</code>	send to context stored in FILENAME
<code>--ssh-cm</code>	use SSH ControlMaster feature
<code>--no-ssh-cm</code>	don't use SSH ControlMaster feature [DEFAULT]
<code>--http-pipelining</code>	enable HTTP pipelining
<code>--no-http-pipelining</code>	disable HTTP pipelining [DEFAULT]
<code>--no-cache</code>	don't use patch caches

Send by email a bundle of one or more patches.

Do not confuse the `--author` options with the return address that `darcs send` will set for your patch bundle.

For example, if you have two email addresses A and B:

If you use `--author A` but your machine is configured to send mail from address B by default, then the return address on your message will be B.

If you use `--from A` and your mail client supports setting the From: address arbitrarily (some non-Unix-like mail clients, especially, may not support this), then the return address will be A; if it does not support this, then the return address will be B.

If you supply neither `--from` nor `--author`, then the return address will be B.

In addition, unless you specify the sendmail command with `--sendmail-command`, `darcs` sends email using the default email command on your computer. This default command is determined by the `configure` script. Thus, on some non-Unix-like OSes, `--from` is likely to not work at all.

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

The `--output`, `--output-auto-name`, 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 specify `--output-auto-name`, the patch bundle is saved to a file with an automatically generated name. If you give one or more `--to` arguments, the bundle of patches is sent to those locations. The locations may either be email addresses or urls that the patch should be submitted to via HTTP.

If you don't provide any of these options, `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 `_darcs/prefs/post` exists in the target repository, `darcs` will upload to the URL contained in that file, which may either be a `mailto: URL`, or an `http:// URL`. In the latter case, the patch is posted to that URL.

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

--subject

Use the **--subject** flag to set the subject of the e-mail to be sent. If you don't provide a subject on the command line, darcs will make one up based on names of the patches in the patch bundle.

--in-reply-to

Use the **--in-reply-to** flag to set the In-Reply-To and References headers of the e-mail to be sent. By default no additional headers are included so e-mail will not be treated as reply by mail readers.

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

The **--patches**, **--matches**, **--tags**, and **--no-deps** options can be used to select which patches to send, as described in subsection 6.1.

--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 a message to go along with your patches.

--sendmail-command

If you want to use a command different from the default one for sending email, you need to specify a command line with the **--sendmail-command** option. The command line 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 URL-encoded values. Additionally you can add **%<** to the end of the command line if the command expects the complete email message on standard input. E.g. the command lines for evolution and msmtpl look like this:

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

6.6.5 darcs apply

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

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

Options:

<code>--verify PUBRING</code>	verify that the patch was signed by a key in PUBRING
<code>--verify-ssl KEYS</code>	verify using openssl with authorized keys from file KEYS
<code>--no-verify</code>	don't verify patch signature
<code>-a --all</code>	answer yes to all patches
<code>-i --interactive</code>	prompt user interactively
<code>--dry-run</code>	don't actually take the action
<code>--xml-output</code>	generate XML formatted output
<code>--mark-conflicts</code>	mark conflicts
<code>--allow-conflicts</code>	allow conflicts, but don't mark them
<code>--no-resolve-conflicts</code>	equivalent to <code>--dont-allow-conflicts</code> , for backwards compatibility
<code>--dont-allow-conflicts</code>	fail on patches that create conflicts [DEFAULT]
<code>--external-merge COMMAND</code>	use external tool to merge conflicts
<code>--no-test</code>	don't run the test script
<code>--test</code>	run the test script
<code>--leave-test-directory</code>	don't remove the test directory
<code>--remove-test-directory</code>	remove the test directory
<code>--reporidir DIRECTORY</code>	specify the repository directory in which to run

Advanced options:

<code>--reply FROM</code>	reply to email-based patch using FROM address
<code>--cc EMAIL</code>	mail results to additional EMAIL(s). Requires <code>--reply</code>
<code>--happy-forwarding</code>	forward unsigned messages without extra header
<code>--sendmail-command COMMAND</code>	specify sendmail command
<code>--ignore-times</code>	don't trust the file modification times
<code>--compress</code>	create compressed patches
<code>--dont-compress</code>	don't create compressed patches
<code>--set-scripts-executable</code>	make scripts executable
<code>--dont-set-scripts-executable</code>	don't make scripts executable
<code>--umask UMASK</code>	specify umask to use when writing
<code>--restrict-paths</code>	don't allow darcs to touch external files or repo metadata
<code>--dont-restrict-paths</code>	allow darcs to modify any file or directory (unsafe)

Apply a patch bundle created by '`darcs 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. Darcs also interprets an argument of ‘’ to mean it should read the file from standard input. 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.

--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 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 --reporir ~/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: In an attempt to make sure no one else can read your email, mutt seems to set the umask such that patches created with the above macro are not world-readable, so 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 repository 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 repository. 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, --interactive`

If you provide the `--interactive` 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` which might be set in your “defaults” file.

`--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 command line if the command expects the complete mail on standard input. For example, the command line 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.

6.7 Seeing what you've done

6.7.1 darcs whatsnew

The ‘darcs whatsnew’ command lists unrecorded changes to the working tree. If you specify a set of files and directories, only unrecorded changes to those files and directories are listed.

With the `-summary` option, the changes are condensed to one line per file, with mnemonics to indicate the nature and extent of the change. The `-look-for-adds` option causes candidates for ‘darcs add’ to be included in the summary output.

By default, ‘darcs whatsnew’ uses Darcs’ internal format for changes. To see some context (unchanged lines) around each change, use the `-unified` option. To view changes in conventional ‘diff’ format, use the ‘darcs diff’ command; but note that ‘darcs whatsnew’ is faster.

This command exits unsuccessfully (returns a non-zero exit status) if there are no unrecorded changes.

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

Options:

<code>-s</code>	<code>--summary</code>	summarize changes
	<code>--no-summary</code>	don't summarize changes
<code>-u</code>	<code>--unified</code>	output patch in a darcs-specific format similar to diff -u
<code>-l</code>	<code>--look-for-adds</code>	look for (non-boring) files that could be added
	<code>--dont-look-for-adds</code>	don't look for any files that could be added
		[DEFAULT]
	<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Advanced options:

<code>--ignore-times</code>	don't trust the file modification times	
<code>--boring</code>	don't skip boring files	List
<code>--no-cache</code>	don't use patch caches	

unrecorded changes in the working tree.

6.7.2 darcs changes

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

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
-n	--index N-M	select a range of patches
	--matches PATTERN	select patches matching PATTERN
-p	--patches REGEXP	select patches matching REGEXP
-t	--tags REGEXP	select tags matching REGEXP
	--max-count NUMBER	return only NUMBER results
	--only-to-files	show only changes to specified files
	--context	give output suitable for get -context
	--xml-output	generate XML formatted output
	--human-readable	give human-readable output
	--number	number the changes
	--count	output count of changes
-s	--summary	summarize changes
	--no-summary	don't summarize changes
	--reverse	show changes in reverse order
	--repo URL	specify the repository URL
	--repodir DIRECTORY	specify the repository directory in which to run
-a	--all	answer yes to all patches
-i	--interactive	prompt user interactively

Advanced options:

--ssh-cm	use SSH ControlMaster feature
--no-ssh-cm	don't use SSH ControlMaster feature [DEFAULT]
--http-pipelining	enable HTTP pipelining
--no-http-pipelining	disable HTTP pipelining [DEFAULT]
--no-cache	don't use patch caches

Gives a changelog-style summary of the repository history.

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.

```
--max-count
```

If `changes` is given a `--max-count` option, it only outputs up to as that number of changes.

```
--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.

6.7.3 `darcs show`

The `show` command provides access to several subcommands which can be used to investigate the state of a repository.

`darcs show authors`

The ‘`darcs show authors`’ command lists the authors of the current repository, sorted by the number of patches contributed. With the `-verbose` option, this command simply lists the author of each patch (without aggregation or sorting).

An author’s name or email address may change over time. To tell Darcs when multiple author strings refer to the same individual, create an ‘`.authorspellings`’ file in the root of the working tree. Each line in this file begins with an author’s canonical name and address, and may be followed by a comma and zero or more extended regular expressions, separated by commas. Blank lines and lines beginning with two hyphens, are ignored.

Any patch with an author string that matches the canonical address or any of the associated regexps is considered to be the work of that author. All matching is case-insensitive and partial (it can match a substring).

Currently this canonicalization step is done only in ‘darcs show authors’. Other commands, such as ‘darcs changes’ use author strings verbatim.

An example .authorspelling file is:

– This is a comment. Fred Nurk ;fred@example.com; John Snagge ;snagge@bbc.co.uk;, John, snagge@, js@(si—mit).edu

Usage: darcs show authors [OPTION]...

Options:

–repodir DIRECTORY specify the repository directory in which to run

List authors by patch count.

darcs show contents

Show contents can be used to display an earlier version of some file(s). If you give show contents no version arguments, it displays the recorded version of the file(s).

Usage: darcs show contents [OPTION]... [FILE]...

Options:

–match PATTERN select a single patch matching PATTERN
 -p –patch REGEXP select a single patch matching REGEXP
 -t –tag REGEXP select tag matching REGEXP
 -n –index N select one patch
 –repodir DIRECTORY specify the repository directory in which to run

Outputs a specific version of a file.

darcs show files

The ‘darcs show files’ command lists those files and directories in the working tree that are under version control. This command is primarily for scripting purposes; end users will probably want ‘darcs whatsnew –summary’.

A file is ‘pending’ if it has been added but not recorded. By default, pending files (and directories) are listed; the –no-pending option prevents this.

By default ‘darcs show files’ lists both files and directories, but the alias ‘darcs show manifest’ only lists files. The –files, –directories, –no-files and –no-directories modify this behaviour.

By default entries are one-per-line (i.e. newline separated). This can cause problems if the files themselves contain newlines or other control characters. To get around this, the –null option uses the null character instead. The script interpreting output from this command needs to understand this idiom; ‘xargs -0’ is such a command.

For example, to list version-controlled files by size:

darcs show files -0 — xargs -0 ls -ldS

Usage: darcs show files [OPTION]...

Options:

<code>--files</code>	include files in output [DEFAULT]
<code>--no-files</code>	don't include files in output
<code>--directories</code>	include directories in output [DEFAULT]
<code>--no-directories</code>	don't include directories in output
<code>--pending</code>	reflect pending patches in output [DEFAULT]
<code>--no-pending</code>	only included recorded patches in output
<code>-0 --null</code>	separate file names by NUL characters
<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Show version-controlled files in the working copy.

darcs show tags

The `tags` command writes a list of all tags in the repository to standard output.

Tab characters (ASCII character 9) in tag names are changed to spaces for better interoperability with shell tools. A warning is printed if this happens.

Usage: `darcs show tags [OPTION]...`

Options:

<code>--repodir DIRECTORY</code>	specify the repository directory in which to run
----------------------------------	--

Show all tags in the repository.

darcs show repo

The `repo` command displays information about the current repository (location, type, etc.). Some of this information is already available by inspecting files within the `_darcs` directory and some is internal information that is informational only (i.e. for developers). This command collects all of the repository information into a readily available source.

Usage: `darcs show repo [OPTION]...`

Options:

<code>--repodir DIRECTORY</code>	specify the repository directory in which to run
<code>--files</code>	include files in output [DEFAULT]
<code>--no-files</code>	don't include files in output
<code>--xml-output</code>	generate XML formatted output

Show repository summary information

The `show repo` displays information about the current repository: the location, the type, etc.

This is provided as informational output for two purposes: curious users and scripts invoking `darcs`. For the latter, this information can be parsed to facilitate the script; for example, `darcs show repo | grep Root: | awk {print $2}` can be used to locate the top-level `_darcs` directory from anyplace within a `darcs` repository working directory.

`--files, --no-files`

If the `--files` option is specified (the default), then the `show repo` operation will read patch information from the repository and display the number of patches in the repository. The `--no-files` option can be used to suppress this operation (and improve performance).

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

By default, the `show repo` displays output in human readable form, but the `--xml-output` option can be used to obtain XML-formatted to facilitate regular parsing by external tools.

6.8 More advanced commands

6.8.1 darcs tag

The ‘darcs tag’ command names the current repository state, so that it can easily be referred to later. Every ‘important’ state should be tagged; in particular it is good practice to tag each stable release with a number or codename. Advice on release numbering can be found at <http://producingoss.com/en/development-cycle.html>.

To reproduce the state of a repository ‘R’ as at tag ‘t’, use the command ‘darcs get -tag t R’. The command ‘darcs show tags’ lists all tags in the current repository.

Tagging also provides significant performance benefits: when Darcs reaches a shared tag that depends on all antecedent patches, it can simply stop processing.

Like normal patches, a tag has a name, an author, a timestamp and an optional long description, but it does not change the working tree. A tag can have any name, but it is generally best to pick a naming scheme and stick to it.

The ‘darcs tag’ command accepts the `-pipe` and `-checkpoint` options, which behave as described in ‘darcs record’ and ‘darcs optimize’ respectively.

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

Options:

<code>-m</code>	<code>--patch-name PATCHNAME</code>	name of patch
<code>-A</code>	<code>--author EMAIL</code>	specify author id
	<code>--checkpoint</code>	create a checkpoint file (see <code>get -partial</code>)
	<code>--pipe</code>	ask user interactively for the patch metadata
<code>-i</code>	<code>--interactive</code>	prompt user interactively
	<code>--edit-long-comment</code>	edit the long comment by default
	<code>--skip-long-comment</code>	don't give a long comment
	<code>--prompt-long-comment</code>	prompt for whether to edit the long comment
	<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Advanced options:

<code>--compress</code>	create compressed patches	
<code>--dont-compress</code>	don't create compressed patches	Name
<code>--umask UMASK</code>	specify umask to use when writing	

the current repository state for future reference.

6.8.2 darcs setpref

When working on project with multiple repositories and contributors, it is sometimes desirable for a preference to be set consistently project-wide. This is achieved by treating a preference set with 'darcs setpref' as an unrecorded change, which can then be recorded and then treated like any other patch.

Valid preferences are:

`test` – a shell command that runs regression tests
`predist` – a shell command to run before 'darcs dist'
`boringfile` – the path to a version-controlled boring file
`binariesfile` – the path to a version-controlled binaries file

For example, a project using GNU autotools, with a 'make test' target to perform regression tests, might enable Darcs' integrated regression testing with the following command:

```
darcs setpref test 'autoconf && ./configure && make && make test'
```

Note that merging is not currently implemented for preferences: if two patches attempt to set the same preference, the last patch applied to the repository will always take precedence. This is considered a low-priority bug, because preferences are seldom set.

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

Options:

<code>--repodir DIRECTORY</code>	specify the repository directory in which to run
----------------------------------	--

Advanced options:

<code>--umask UMASK</code>	specify umask to use when writing	Set a preference (test, predist, boringfile or binariesfile).
----------------------------	-----------------------------------	---

6.8.3 darcs check

This command verifies that the patches in the repository, when applied successively to an empty tree, result in the pristine tree. If not, the differences are printed and Darcs exits unsuccessfully (with a non-zero exit status).

If the repository is in darcs-1 format and has a checkpoint, you can use the `-partial` option to start checking from the latest checkpoint. This is the default for partial darcs-1 repositories; the `-complete` option to forces a full check.

If a regression test is defined (see 'darcs setpref') it will be run by 'darcs check'. Use the `-no-test` option to disable this.

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

Options:

<code>--complete</code>	get a complete copy of the repository
<code>--partial</code>	get partial repository using checkpoint (old-fashioned format only)
<code>--no-test</code>	don't run the test script
<code>--test</code>	run the test script
<code>--leave-test-directory</code>	don't remove the test directory
<code>--remove-test-directory</code>	remove the test directory
<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Check the repository for consistency.

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 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 3.1).

6.8.4 darcs optimize

The ‘darcs optimize’ command modifies the current repository in an attempt to reduce its resource requirements. By default a single fast, safe optimization is performed; additional optimization techniques can be enabled by passing options to ‘darcs optimize’.

The default optimization moves recent patches (those not included in the latest tag) to the ‘front’, reducing the amount that a typical remote command needs to download. It should also reduce the CPU time needed for some operations.

The ‘darcs optimize -relink’ command hard-links patches that the current repository has in common with its peers. Peers are those repositories listed in `_darcs/prefs/sources`, or defined with the ‘-sibling’ option (which can be used multiple times).

Darcs uses hard-links automatically, so this command is rarely needed. It is most useful if you used ‘cp -r’ instead of ‘darcs get’ to copy a repository, or if you pulled the same patch from a remote repository into multiple local repositories.

A ‘`darcs optimize --relink-pristine`’ command is also available, but generally SHOULD NOT be used. It results in a relatively small space saving at the cost of making many Darcs commands MUCH slower.

By default patches are compressed with zlib (RFC 1951) to reduce storage (and download) size. In exceptional circumstances, it may be preferable to avoid compression. In this case the ‘`--dont-compress`’ option can be used (e.g. with ‘`darcs record`’) to avoid compression.

The ‘`darcs optimize --uncompress`’ and ‘`darcs optimize --compress`’ commands can be used to ensure existing patches in the current repository are respectively uncompressed or compressed. Note that repositories in the legacy ‘old-fashioned-inventory’ format have a `.gz` extension on patch files even when uncompressed.

If the repository is in ‘old-fashioned-inventory’ format, the ‘`darcs optimize --checkpoint`’ command creates a checkpoint of the latest tag. This checkpoint is used by ‘`darcs get --partial`’ to create partial repositories. With the ‘`--tag`’ option, checkpoints for older tags can be created. In newer repository formats, this feature has been replaced by ‘`darcs get --lazy`’, which does not require checkpoints.

There is one more optimization which CAN NOT be performed by this command. Every time you record a patch, a new inventory file is written to `._darcs/inventories/`, and old inventories are never reaped.

If `._darcs/inventories/` is consuming a relatively large amount of space, you can safely reclaim it by using ‘`darcs get`’ to make a complete copy of the repo. When doing so, don’t forget to copy over any unsaved changes you have made to the working tree or to unversioned files in `._darcs/prefs/` (such as `._darcs/prefs/author`).

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

Options:

<code>--checkpoint</code>	create a checkpoint file (see <code>get --partial</code>)
<code>-t --tag TAGNAME</code>	name of version to checkpoint
<code>--repodir DIRECTORY</code>	specify the repository directory in which to run
<code>--reorder-patches</code>	reorder the patches in the repository
<code>--sibling URL</code>	specify a sibling directory
<code>--relink</code>	relink random internal data to a sibling
<code>--relink-pristine</code>	relink pristine tree (not recommended)

Advanced options:

<code>--compress</code>	create compressed patches	
<code>--dont-compress</code>	don’t create compressed patches	Op-
<code>--uncompress</code>	uncompress patches	
<code>--umask UMASK</code>	specify umask to use when writing	

optimize the repository.

`--reorder-patches`

The `--reorder-patches` option causes Darcs to create an optimal order-

ing 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.

6.9 Undoing, redoing and running in circles

6.9.1 darcs amend-record

Amend-record updates a ‘draft’ patch with additions or improvements, resulting in a single ‘finished’ patch. This is better than recording the additions and improvements as separate patches, because then whenever the ‘draft’ patch is copied between repositories, you would need to make sure all the extra patches are copied, too.

Do not copy draft patches between repositories, because a finished patch cannot be copied into a repository that contains a draft of the same patch. If this has already happened, ‘darcs obliterate’ can be used to remove the draft patch.

Do not run amend-record in repository that other developers can pull from, because if they pull while an amend-record is in progress, their repository may be corrupted.

When recording a draft patch, it is a good idea to start the name with ‘DRAFT:’ so that other developers know it is not finished. When finished, remove it with ‘darcs amend-record –edit-long-comment’. To change the patch name without starting an editor, use `–patch-name`.

Like ‘darcs record’, if you call amend-record with files as arguments, you will only be asked about changes to those files. So to amend a patch to `foo.c` with improvements in `bar.c`, you would run:

```
darcs amend-record –match 'touch foo.c' bar.c
```

It is usually a bad idea to amend another developer’s patch. To make amend-record only ask about your own patches by default, you can add something like ‘amend-record match David Roundy’ to `./darcs/defaults`, where ‘David Roundy’ is your name. On Windows use `C:/Documents And Settings/user/Application Data/darcs/defaults`

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

Options:

	--match PATTERN	select a single patch matching PATTERN
-p	--patch REGEXP	select a single patch matching REGEXP
-n	--index N	select one patch
	--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
-a	--all	answer yes to all patches
-i	--interactive	prompt user interactively
-A	--author EMAIL	specify author id
-m	--patch-name PATCHNAME	name of patch
	--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
-l	--look-for-adds	look for (non-boring) files that could be added
	--dont-look-for-adds	don't look for any files that could be added [DEFAULT]
	--reporir DIRECTORY	specify the repository directory in which to run

Advanced options:

	--compress	create compressed patches
	--dont-compress	don't create compressed patches
	--ignore-times	don't trust the file modification times
	--umask UMASK	specify umask to use when writing
	--set-scripts-executable	make scripts executable
	--dont-set-scripts-executable	don't make scripts executable

Improve a patch before it leaves your repository.

6.9.2 darcs rollback

Rollback is used to undo the effects of one or more patches without actually deleting them. Instead, it creates a new patch reversing selected portions. of those changes. Unlike obliterate and unrecord (which accomplish a similar goal) rollback is perfectly safe, since it leaves in the repository a record of its changes.

Usage: darcs rollback [OPTION]... [FILE or DIRECTORY]...

Options:

<code>--from-match PATTERN</code>	select changes starting with a patch matching PATTERN
<code>--from-patch REGEXP</code>	select changes starting with a patch matching REGEXP
<code>--from-tag REGEXP</code>	select changes starting with a tag matching REGEXP
<code>--last NUMBER</code>	select the last NUMBER patches
<code>--matches PATTERN</code>	select patches matching PATTERN
<code>-p --patches REGEXP</code>	select patches matching REGEXP
<code>-t --tags REGEXP</code>	select tags matching REGEXP
<code>-a --all</code>	answer yes to all patches
<code>-i --interactive</code>	prompt user interactively
<code>-A --author EMAIL</code>	specify author id
<code>-m --patch-name PATCHNAME</code>	name of patch
<code>--edit-long-comment</code>	edit the long comment by default
<code>--skip-long-comment</code>	don't give a long comment
<code>--prompt-long-comment</code>	prompt for whether to edit the long comment
<code>--no-test</code>	don't run the test script
<code>--test</code>	run the test script
<code>--leave-test-directory</code>	don't remove the test directory
<code>--remove-test-directory</code>	remove the test directory
<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Advanced options:

<code>--compress</code>	create compressed patches	
<code>--dont-compress</code>	don't create compressed patches	Record
<code>--umask UMASK</code>	specify umask to use when writing	

a new patch reversing some recorded changes.

If you decide you didn't want to roll back a patch after all, you can reverse its effect by obliterating the rolled-back patch.

Rollback can actually allow you to roll back a subset of the changes made by the selected patch or patches. Many of the options available in rollback behave similarly to the options for unrecord 6.9.3 and record 6.6.1.

6.9.3 darcs unrecord

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. Beware that you should not use this command if you are going to re-record the changes in any way and there is a possibility that another user may have already pulled the patch.

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

Options:

<code>--from-match PATTERN</code>	select changes starting with a patch matching PATTERN
<code>--from-patch REGEXP</code>	select changes starting with a patch matching REGEXP
<code>--from-tag REGEXP</code>	select changes starting with a tag matching REGEXP
<code>--last NUMBER</code>	select the last NUMBER patches
<code>--matches PATTERN</code>	select patches matching PATTERN
<code>-p --patches REGEXP</code>	select patches matching REGEXP
<code>-t --tags REGEXP</code>	select tags matching REGEXP
<code>--no-deps</code>	don't automatically fulfill dependencies
<code>--dont-prompt-for-dependencies</code>	don't ask about patches that are depended on by matched patches (with <code>-match</code> or <code>-patch</code>)
<code>--prompt-for-dependencies</code>	prompt about patches that are depended on by matched patches [DEFAULT]
<code>-a --all</code>	answer yes to all patches
<code>-i --interactive</code>	prompt user interactively
<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Advanced options:

<code>--compress</code>	create compressed patches	
<code>--dont-compress</code>	don't create compressed patches	Re-
<code>--umask UMASK</code>	specify umask to use when writing	

move recorded patches without changing the working copy.

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.⁶

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

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 obliterate! 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 repository. Attempting to do so may cause repository corruption.

⁶The 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.

`--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, --no-deps`

The `--patches`, `--matches`, `--tags`, and `--no-deps` options can be used to select which patches to unrecord, as described in subsection 6.1.

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. Or if you use `--no-deps` you won't be asked about patches that can't be unrecorded due to depending patches.

Selecting patches can be slow, so darcs cuts the search at the last optimized tag. Use the `--from` or `--last` options to search more or fewer patches.

6.9.4 darcs obliterate

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!

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

Options:

<code>--from-match PATTERN</code>	select changes starting with a patch matching PATTERN
<code>--from-patch REGEXP</code>	select changes starting with a patch matching REGEXP
<code>--from-tag REGEXP</code>	select changes starting with a tag matching REGEXP
<code>--last NUMBER</code>	select the last NUMBER patches
<code>--matches PATTERN</code>	select patches matching PATTERN
<code>-p --patches REGEXP</code>	select patches matching REGEXP
<code>-t --tags REGEXP</code>	select tags matching REGEXP
<code>--no-deps</code>	don't automatically fulfill dependencies
<code>--dont-prompt-for-dependencies</code>	don't ask about patches that are depended on by matched patches (with <code>-match</code> or <code>-patch</code>)
<code>--prompt-for-dependencies</code>	prompt about patches that are depended on by matched patches [DEFAULT]
<code>-a --all</code>	answer yes to all patches
<code>-i --interactive</code>	prompt user interactively
<code>--reporidir DIRECTORY</code>	specify the repository directory in which to run
<code>-s --summary</code>	summarize changes
<code>--no-summary</code>	don't summarize changes

Advanced options:

<code>--compress</code>	create compressed patches	
<code>--dont-compress</code>	don't create compressed patches	
<code>--ignore-times</code>	don't trust the file modification times	Delete
<code>--umask UMASK</code>	specify umask to use when writing	

selected patches from the repository. (UNSAFE!) 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 `unpull` was the old name for obliterate. `Unpull` is still a hidden alias for obliterate.

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

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

Usually you only want to obliterate the latest changes, and almost never would you want to obliterate changes before a tag—you would have to have obliterated 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 obliterate more patches in one go, there are the `--from` and `--last` options to set the earliest patch selectable to obliterate.

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

The `--patches`, `--matches`, `--tags`, and `--no-deps` options can be used to select which patches to obliterate, as described in subsection 6.1.

With these options you can specify what patch or patches to be prompted for by obliterate. This is especially useful when you want to obliterate patches with dependencies, since all the dependent patches (but no others) will be included in the choices. Or if you use `--no-deps` you won't be asked about patches that can't be obliterated due to depending patches.

Selecting patches can be slow, so darcs cuts the search at the last optimized tag. Use the `--from` or `--last` options to search more or fewer patches.

6.9.5 darcs revert

The 'darcs revert' command discards unrecorded changes the working tree. As with 'darcs record', you will be asked which hunks (changes) to revert. The `-all` switch can be used to avoid such prompting. If files or directories are specified, other parts of the working tree are not reverted.

In you accidentally reverted something you wanted to keep (for example, typing 'darcs rev -a' instead of 'darcs rec -a'), you can immediately run 'darcs unrevert' to restore it. This is only guaranteed to work if the repository has not changed since 'darcs revert' ran.

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

Options:

<code>-a</code>	<code>--all</code>	answer yes to all patches
<code>-i</code>	<code>--interactive</code>	prompt user interactively
	<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Advanced options:

<code>--ignore-times</code>	don't trust the file modification times	Dis-
<code>--umask UMASK</code>	specify umask to use when writing	

card unrecorded changes.

6.9.6 darcs unrevert

Unrevert is a rescue command in case you accidentally reverted something you wanted to keep (for example, typing 'darcs rev -a' instead of 'darcs rec -a').

This command may fail if the repository has changed since the revert took place. Darcs will ask for confirmation before executing an interactive command that will DEFINITELY prevent unreversion.

Usage: darcs unrevert [OPTION]...

Options:

<code>--ignore-times</code>	don't trust the file modification times	
<code>-a</code>	<code>--all</code>	answer yes to all patches
<code>-i</code>	<code>--interactive</code>	prompt user interactively
	<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Advanced options:

`--umask UMASK` specify umask to use when writing Undo
the last revert (may fail if changes after the revert).

6.10 Advanced examination of the repository

6.10.1 darcs diff

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

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

Options:

<code>--to-match PATTERN</code>	select changes up to a patch matching PATTERN
<code>--to-patch REGEXP</code>	select changes up to a patch matching REGEXP
<code>--to-tag REGEXP</code>	select changes up to a tag matching REGEXP
<code>--from-match PATTERN</code>	select changes starting with a patch matching PATTERN
<code>--from-patch REGEXP</code>	select changes starting with a patch matching REGEXP
<code>--from-tag REGEXP</code>	select changes starting with a tag matching REGEXP
<code>--match PATTERN</code>	select a single patch matching PATTERN
<code>-p --patch REGEXP</code>	select a single patch matching REGEXP
<code>--last NUMBER</code>	select the last NUMBER patches
<code>-n --index N-M</code>	select a range of patches
<code>--diff-command COMMAND</code>	specify diff command (ignores <code>--diff-opts</code>)
<code>--diff-opts OPTIONS</code>	options to pass to diff
<code>-u --unified</code>	pass <code>-u</code> option to diff
<code>--repodir DIRECTORY</code>	specify the repository directory in which to run
<code>--store-in-memory</code>	do patch application in memory rather than on disk

Create a diff between two versions of the repository.

`--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/
```

```
--diff-command
```

You can use a different program to view differences by including the flag `--diff-command`, e.g.

```
--diff-command 'opendiff %1 %2'.
```

The `%1` and `%2` are replaced with the two versions to be merged. The above example works with the FileMerge.app tool that comes with Apple's developer tools. To use `xxdiff`, you would use

```
--diff-command 'xxdiff %1 %2'
```

To use `kdiff3`, you can use

```
--diff-command 'kdiff3 %1 %2'
```

Note that the command is split into space-separated words and the first one is `execed` with the rest as arguments—it is not a shell command. Also the substitution of the `%` escapes is only done on complete words. See 6.1 for how you might work around this fact, for example, with Emacs' Ediff package.

Note also that the `--diff-opts` flag is ignored if you use this option.

6.10.2 darcs annotate

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.

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

Options:

-s	--summary	summarize changes
	--no-summary	don't summarize changes
-u	--unified	output patch in a darcs-specific format
	--human-readable	similar to diff -u give human-readable output
	--xml-output	generate XML formatted output
	--match PATTERN	select a single patch matching PATTERN
-p	--patch REGEXP	select a single patch matching REGEXP
-t	--tag REGEXP	select tag matching REGEXP
-n	--index N	select one patch
	--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.

--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.

If a patch name and a file name are given, these details are output for the time after that patch was applied.

--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.

6.10.3 darcs show

The `show` command provides access to several subcommands which can be used to investigate the state of a repository.

darcs show authors

The ‘`darcs show authors`’ command lists the authors of the current repository, sorted by the number of patches contributed. With the `-verbose` option, this command simply lists the author of each patch (without aggregation or sorting).

An author’s name or email address may change over time. To tell Darcs when multiple author strings refer to the same individual, create an ‘`.authorspellings`’ file in the root of the working tree. Each line in this file begins with an author’s canonical name and address, and may be followed by a comma and zero or more extended regular expressions, separated by commas. Blank lines and lines beginning with two hyphens, are ignored.

Any patch with an author string that matches the canonical address or any of the associated regexps is considered to be the work of that author. All matching is case-insensitive and partial (it can match a substring).

Currently this canonicalization step is done only in ‘`darcs show authors`’. Other commands, such as ‘`darcs changes`’ use author strings verbatim.

An example `.authorspelling` file is:

```
- This is a comment. Fred Nurk jfred@example.com, John Snagge jsnagge@bbc.co.uk,
John, snagge@, js@(si—mit).edu
```

Usage: `darcs show authors [OPTION]...`

Options:

`--reporidir DIRECTORY` specify the repository directory in which to run

List authors by patch count.

darcs show contents

`Show contents` can be used to display an earlier version of some file(s). If you give `show contents` no version arguments, it displays the recorded version of the file(s).

Usage: `darcs show contents [OPTION]... [FILE]...`

Options:

	<code>--match PATTERN</code>	select a single patch matching PATTERN
<code>-p</code>	<code>--patch REGEXP</code>	select a single patch matching REGEXP
<code>-t</code>	<code>--tag REGEXP</code>	select tag matching REGEXP
<code>-n</code>	<code>--index N</code>	select one patch
	<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Outputs a specific version of a file.

darcs show files

The ‘darcs show files’ command lists those files and directories in the working tree that are under version control. This command is primarily for scripting purposes; end users will probably want ‘darcs whatsnew -summary’.

A file is ‘pending’ if it has been added but not recorded. By default, pending files (and directories) are listed; the `-no-pending` option prevents this.

By default ‘darcs show files’ lists both files and directories, but the alias ‘darcs show manifest’ only lists files. The `-files`, `-directories`, `-no-files` and `-no-directories` modify this behaviour.

By default entries are one-per-line (i.e. newline separated). This can cause problems if the files themselves contain newlines or other control characters. To get around this, the `-null` option uses the null character instead. The script interpreting output from this command needs to understand this idiom; ‘xargs -0’ is such a command.

For example, to list version-controlled files by size:

```
darcs show files -0 — xargs -0 ls -ldS
```

Usage: darcs show files [OPTION]...

Options:

	<code>--files</code>	include files in output [DEFAULT]
	<code>--no-files</code>	don’t include files in output
	<code>--directories</code>	include directories in output [DEFAULT]
	<code>--no-directories</code>	don’t include directories in output
	<code>--pending</code>	reflect pending patches in output [DEFAULT]
	<code>--no-pending</code>	only included recorded patches in output
<code>-0</code>	<code>--null</code>	separate file names by NUL characters
	<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Show version-controlled files in the working copy.

darcs show tags

The tags command writes a list of all tags in the repository to standard output.

Tab characters (ASCII character 9) in tag names are changed to spaces for better interoperability with shell tools. A warning is printed if this happens.

Usage: darcs show tags [OPTION]...

Options:

`--repor dir DIRECTORY` specify the repository directory in which to run
 Show all tags in the repository.

darcs show repo

The `repo` command displays information about the current repository (location, type, etc.). Some of this information is already available by inspecting files within the `_darcs` directory and some is internal information that is informational only (i.e. for developers). This command collects all of the repository information into a readily available source.

Usage: `darcs show repo [OPTION]...`

Options:

`--repor dir DIRECTORY` specify the repository directory in which to run
`--files` include files in output [DEFAULT]
`--no-files` don't include files in output
`--xml-output` generate XML formatted output

Show repository summary information

The `show repo` displays information about the current repository: the location, the type, etc.

This is provided as informational output for two purposes: curious users and scripts invoking `darcs`. For the latter, this information can be parsed to facilitate the script; for example, `darcs show repo | grep Root: | awk {print $2}` can be used to locate the top-level `_darcs` directory from anyplace within a `darcs` repository working directory.

`--files, --no-files`

If the `--files` option is specified (the default), then the `show repo` operation will read patch information from the repository and display the number of patches in the repository. The `--no-files` option can be used to suppress this operation (and improve performance).

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

By default, the `show repo` displays output in human readable form, but the `--xml-output` option can be used to obtain XML-formatted to facilitate regular parsing by external tools.

6.11 Rarely needed and obscure commands

6.11.1 darcs convert

The current repository format is called 'darcs-2'. It was introduced in Darcs 2.0 and became the default for new projects in Darcs 2.2. The 'darcs convert' com-

mand allows existing projects to migrate to this format from the older ‘darcs-1’ format.

This command DOES NOT modify the source repository; a new destination repository is created. It is safe to run this command more than once on a repository (e.g. for testing), before the final conversion.

WARNING: the repository produced by this command is not understood by Darcs 1.x, and patches cannot be exchanged between repositories in darcs-1 and darcs-2 formats.

Furthermore, darcs 2 repositories created by different invocations of this command SHOULD NOT exchange patches, unless those repositories had no patches in common when they were converted. (That is, within a set of repos that exchange patches, no patch should be converted more than once.)

Due to this limitation, migrating a multi-branch project is a little awkward. Sorry! Here is the recommended process:

1. for each branch ‘foo’, tag that branch with ‘foo-final’;
2. merge all branches together (–allow-conflicts may help);
3. run ‘darcs optimize –reorder’ on the result;
4. run ‘darcs convert’ to create a merged darcs-2 repository;
5. re-create each branch by calling ‘darcs get –tag foo-final’ on the darcs-2 repository;
- and finally 6. use ‘darcs obliterate’ to delete the foo-final tags.

Usage: `darcs convert [OPTION]... <SOURCE> [<DESTINATION>]`

Options:

<code>--repo-name DIRECTORY,--repodir DIRECTORY</code>	path of output directory
<code>--set-scripts-executable</code>	make scripts executable
<code>--dont-set-scripts-executable</code>	don’t make scripts executable

Advanced options:

<code>--ssh-cm</code>	use SSH ControlMaster feature
<code>--no-ssh-cm</code>	don’t use SSH ControlMaster feature
	[DEFAULT]
<code>--http-pipelining</code>	enable HTTP pipelining
<code>--no-http-pipelining</code>	disable HTTP pipelining [DEFAULT]
<code>--no-cache</code>	don’t use patch caches

Convert a repository from a legacy format.

6.11.2 darcs mark-conflicts

Darcs requires human guidance to unify changes to the same part of a source file. When a conflict first occurs, darcs will add both choices to the working tree, delimited by the markers ‘v v v’, ‘* * *’ and ‘^ ^ ^’.

However, you might revert or manually delete these markers without actually resolving the conflict. In this case, ‘darcs mark-conflicts’ is useful to show where any unresolved conflicts. It is also useful if ‘darcs apply’ is called with –apply-conflicts, where conflicts aren’t marked initially.

Any unrecorded changes to the working tree WILL be lost forever when you run this command! You will be prompted for confirmation before this takes place.

This command was historically called ‘resolve’, and this deprecated alias still exists for backwards-compatibility.

Usage: `darcs mark-conflicts` [OPTION]...

Options:

<code>--ignore-times</code>	don't trust the file modification times
<code>--repodir DIRECTORY</code>	specify the repository directory in which to run

Advanced options:

<code>--umask UMASK</code>	specify umask to use when writing	Mark unresolved conflicts in working tree, for manual resolution.
----------------------------	-----------------------------------	---

6.11.3 darcs dist

The ‘darcs dist’ command creates a compressed archive (a ‘tarball’) in the repository’s root directory, containing the recorded state of the working tree (unrecorded changes and the `_darcs` directory are excluded).

If a `predist` command is set (see ‘darcs setpref’), that command will be run on the tarball contents prior to archiving. For example, autotools projects would set it to ‘`autoconf && automake`’.

By default, the tarball (and the top-level directory within the tarball) has the same name as the repository, but this can be overridden with the `-dist-name` option.

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

Options:

<code>-d --dist-name DISTNAME</code>	name of version
<code>--repodir DIRECTORY</code>	specify the repository directory in which to run
<code>--match PATTERN</code>	select a single patch matching PATTERN
<code>-p --patch REGEXP</code>	select a single patch matching REGEXP
<code>-t --tag REGEXP</code>	select tag matching REGEXP
<code>-n --index N</code>	select one patch
<code>--store-in-memory</code>	do patch application in memory rather than on disk

Create a distribution tarball.

6.11.4 darcs trackdown

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.

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

Options:

<code>--repodir DIRECTORY</code>	specify the repository directory in which to run
----------------------------------	--

Advanced options:

```
--set-scripts-executable      make scripts executable
--dont-set-scripts-executable  don't make scripts executable
```

Locate the most recent version lacking an error.

Trackdown is helpful for locating when something was broken. It creates a temporary directory with the latest repository 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 obliterate 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.

6.11.5 darcs repair

The ‘darcs repair’ command attempts to fix corruption in the current repository. Currently it can only repair damage to the pristine tree, which is where most corruption occurs.

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

Options:

`--reporir DIRECTORY` specify the repository directory in which to run

Advanced options:

`--umask UMASK` specify umask to use when writing a corrupted repository. Repair

.1 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://wiki.darcs.net/DarcsWiki/ConvertingFromCvs>⁷

.2 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.

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**).

⁷<http://wiki.darcs.net/DarcsWiki/ConvertingFromCvs>

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 move (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://wiki.darcs.net/DarcsWiki/ConvertingFromArch>⁸

⁸<http://wiki.darcs.net/DarcsWiki/ConvertingFromArch>

Appendix A

Building darcs

A.1 The Easy Way

If your distribution provides a pre-built binary package of a recent Darcs release, you are strongly encouraged to use that rather than building Darcs yourself.

If you have (or can install) the `cabal-install` package, this is the next best option, as this will resolve build dependencies automatically. To download, compile and install Darcs and its dependencies via `cabal-install`, simply run

```
cabal update
cabal install darcs
```

A.2 The Hard Way

If you cannot install `cabal-install`, you need to run

```
ghc --make Setup
./Setup configure
./Setup build
./Setup install
```

This will require the following build dependencies:

- GHC 6.8 or higher; and
- Cabal 1.6 or higher.

Additional build dependencies are declared in the `darcs.cabal` file, and `./Setup configure` will tell you if any required build dependencies aren't found. The build dependencies at time of writing (Darcs 2.3) are as follows:

- the C library `zlib`;
- the Haskell packages

- array 1.1 or 1.2;
- containers 1.1 or 1.2;
- directory 1.0;
- filepath 1.1;
- hashed-storage 0.3;
- haskeline 0.6.1 or higher;
- html 1.0;
- mtl 1.0 or 1.1;
- old-time 1.0;
- parsec 2.1 or higher;
- process 1.0;
- random 1.0; and
- regex-compat 0.71 or higher.

Missing *optional* build dependencies are only listed if `./Setup configure` is passed the `--verbose` argument. At time of writing they are:

- the C library libcurl (7.19.1 or higher recommended);
- the Haskell packages
 - bytestring 0.9;
 - bytestring-mmap 0.2 or higher;
 - http 3000 or 3001.1;
 - network 2.2;
 - terminfo 0.3.
 - utf8-string 0.3; and
 - zlib 0.5.

A.3 The Old Way

Darcs can still be built with GNU autotools; this approach is deprecated and will be removed after the 2.3 release. Run

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

The build dependencies are identical to the previous approach, except that instead of the `Cabal` package, you will need

- GNU Make; and
- GNU autoconf 2.50 or higher.

Appendix B

Theory of patches

B.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 to give patch theory better formalized foundations.

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 advise against taking this analogy too seriously, although it could provide some insight into how a physicist might think about darcs.

B.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 representation¹.

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 tree². Thus, the context of a patch consists of the set of patches that precede it.

B.3 Applying patches

B.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.

B.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!

¹For 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.

²This 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.

B.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.

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`

³This 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.

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
```

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 whitespace 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
...
```

Split patch [OBSOLETE!] A split patch is similar to a composite patch 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)
)
```

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.

A simpler, faster (and more generally useful) cousin of `canonize` is the `coalesce` 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.

B.4.1 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. If they happen to modify the same file, we’ll have to check whether or not they commutex.

B.4.2 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.

B.5 Commuting patches

B.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.

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 commutex, 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 commutex. 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 considered⁵.

⁴The 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 antifeile, 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. \curvearrowright So I’m sticking with my hand waving formalism.

⁵Alas, 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

B.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 $P'_2P_1 \longleftrightarrow P'_1P_2$ if and only if $P_1^{-1}P'_2 \longleftrightarrow P_2P_1^{-1}$, provided both commutations succeed. If either commutes 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

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?).

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.

B.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.

B.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.

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>
```

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 C

DarcsRepo format

A repository consists of a working directory, which has within it a directory called `_darcs`. There must also be a subdirectory within `_darcs` named `patches`. The `patches` directory contains the actual patches which are in the repository. There must also be a *pristine tree*, which may either be a directory containing a cache of the version of the tree which has been recorded, or a stub, and may be named either “current” or “pristine”.

WARNING! Viewing files in the pristine cache 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 repository. 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 repository, that repository 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 3.

Appendix D

The GNU General Public License

Version 2, June 1991

Version 2, June 1991
Copyright © 1989, 1991 Free Software Foundation, Inc.

51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent

of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
 - (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
 - (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
 - (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the

right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
 - (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have

received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright

holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR

LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

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

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program’s name and a brief idea of what it does.>

Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) <year> <name of author>
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.

This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James
Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.