IGIT

Git's Even Stupider Cousin[1]

1 Introduction

Igit is a file repository, originally inspired by git, but with a much different design. Under most other version control systems, the location of the repository and the files being managed are tightly linked. Igit breaks that link. A repository in one location can be used to manage files anywhere else on the file system, and there can be multiple repositories, with more than one repository in a single directory. In addition, igit allows for a richer and more detailed record of the parts of a project, how they fit together and their historical relationships.

2 Installation and Top-level Organization

Igit is written in Java, so the first thing to do is install Java, if you haven't already.

A *repository* is a record of history for a collection of files. It includes copies of all the files being tracked, information about the history of changes, how these files fit into a directory structure, *etc.* Each repository is stored in a directory, with the default name `.igit`.

Igit can work with multiple repositories; to keep track of them there is a single directory at a fixed location, called `.igithome` by default. The location for `.igithome` must be specified. Under Windows, use the Control Panel:

System Properties → Environment Variables → System variables

For Linux, set the environment variable in `.bashrc` (for example). In either case, set the `igit_home` variable to the path to the `.igithome` directory. So, the directory isn't required to be called `.igithome`; name it whatever you want in the environment variable.

Once the environment variable has been set up, call the `init` command to create the `.igithome` directory. There can be only one "active" repository; choose it with the `use-repo` command. There are also various commands to add, delete and rename the repositories known to `.igithome`.

The `.igithome` directory contains three files: `ignore`, `active` and `list`. The `list` file is a list of paths to all known repositories (*i.e.,* the `.igit` directories), one path per line, prefixed by a symbolic name for the repository. The `active` file is a single line on which one of the paths from `list` appears; this is the currently active repository, and nearly all commands implicitly work on the active repository.

The `ignore` file is used to specify file patterns which are to be ignored by certain commands. Each line of `ignore` is a pattern, and only three patterns are allowed:

[1] See the git manpage, which describes git as "the stupid content tracker."

- A complete literal file name, like `settings.txt`.
- Something of the form `*X`, where `X` might be `.exe` or `.bak`, *etc.* By default, `ignore` includes `*~` since tilde is used for emacs backup files.
- Something of the form `X*`. The most common thing would be `.*`, which appears by default so that you don't accidentally try to add `.igit` to itself, or any number of other `.something` files that appear in Linux.

If a directory matches the pattern, then the entire directory will be ignored (e.g., `.igit` due to the `.*` pattern). There is no facility to edit the `ignore` file other than opening it with a standard text editor.

There's no behind-the-scenes magic to any of these files. It shouldn't be necessary, but if there's some kind of glitch and the `list` or `active` file becomes corrupted, or you want to do something unexpected, then go ahead and edit the files manually.

3 Repositories and Catenas

The purpose of a repository is to track a collection of files and how they've been changed. A *catena* (Latin for "chain") is the series of historical modifications made to a set of files, together with their directory structure. It's similar to "module," "package," or simply "the project," but with history. It can also be thought of as a mini-repository with a repository.

Using the word "catena" helps to minimize confusion. Even "file" can be confusing since it can mean a particular sequence of bytes in storage, or it could be the history of all versions of a file (however "version" is defined!), or it could be the path where you expect to find a some version of a file. To minimize confusion about these meanings for "file," use *path-instance* to mean a file found at a particular path, and *byte-instance* to mean a particular sequence of bytes forming a file.

As a catena develops over time, a series of changes is made to the files in the catena, and a snapshot of these changes is called a *cut*, as in "cut a new version." Each cut is a set of files (in both the path sense and the byte sense), and a catena is a series of cuts. Each catena has a *head* that points to the currently active cut.

Once a cut has been finalized, it cannot be changed. While a cut is open to changes, it's called a *proto-cut*. The `cut` command finalizes the contents of a proto-cut so that it can no longer be changed; the proto-cut becomes a *fixed cut*, a new proto-cut is created and head moves to this proto-cut.[2] If the history of a catena has never branched, then that catena will have a single proto-cut, and all other cuts will be fixed; each branch (and sub-branch) has at most one proto-cut.

The system allows for sub-catenas, and sub-catenas may have their own independent history, so a sub-catena is a catena in its own right. Allowing for sub-catenas *can* lead to a complicated repository structure, but it's perfectly OK to use the system with only a single catena, and some of the complexities described below don't arise.

Every catena has a symbolic name, and every repository starts with a `main` catena. If you prefer, then you can ignore, delete or rename the `main` catena.

[2] A proto-cut is similar to git's "staging area," and the `cut` command is similar to `git commit`.

4 Catenas and Sub-catenas

The catena/sub-catena structure is unrelated to the directory structure of the files being tracked (unless you want it to be). A catena can contain files from multiple directories, and a file, or entire directories, can appear in multiple catenas.

There are two ways in which a catena can rely on (or *use*) a sub-catena. These two ways are *depends on* and *contains*. If catena A depends on catena B, then B is fixed and can't be modified; if A contains B, then B *can* be modified.

The description above may be confusing for semantic reasons. Every catena consists of a series of cuts, and the statement that "A is a sub-catena of B" is more correctly stated as "a particular cut of B relies on a particular cut of A," or "this B-cut uses that A-cut" That's too verbose for continuous use, and the important point is that the sub-catena relationship is defined on a per-cut basis. There's no such thing as a catena that is always and entirely a sub-catena of another catena. When the cut-based aspect of the sub-catena relationship is important, this document may refer to "sub-cuts" instead of "sub-catenas."

Reasons one might use sub-catenas include:

- A Java program might rely on some Python source code, where the Python and Java develop somewhat, but not entirely, independently.
- The files are in widely dispersed directories. In the Java/Python example, each set of source files might be buried at some depth in different places.
- If the code is broken into different modules, like namespaces in C# or packages in Java, then it might be helpful to distinguish the modules.

The choice between contains or depends is fluid, and can be changed at any time. Changing between the two may make it easier to navigate history and narrow down the places where a particular change was made. If A depends on B over a particular range of cuts, then it is not possible for B to have been changed in any of those cuts; changes to B are only possible when A contains B.

Examples

Suppose that catena `main` *depends on* catenas A and B. This means that the particular cuts of A and B are fixed for `main`; they may not be proto-cuts. Thus, `main` uses a particular version (*i.e.*, a cut) of these two sub-catenas, and those versions won't change as `main` is modified. For instance, A and B may be shared libraries that you want to be careful about changing because other people rely on them. As long as you're working on `main`, you can't edit any of the files in A or B. One quirk to bear in mind is that a file may belong to A, B and `main` (*i.e.*, the file maps to the same location on the disk). In a case like this, that file is not editable, even though it is part of `main` (and the file must be identical, in the byte sense, in every case it appears).

Now suppose that `main` *contains* A and B. Then the files in A and B can be edited when working on main. Executing the `cut` command on `main` leads to a new fixed cut for `main`, and also new cuts for A and B (if files in A and B were modified). Again, if

a file appears in multiple catenas, under the same path, then the same byte-instance must be used at every location.

Because depends-on sub-catenas are fixed, they are easy to understand and manage, but the contains relationship raises some messy issues. Suppose that `main` contains A, with the expectation that development of `main` and A will move forward together. As long as `main` is the only catena that contains A, this is easy to understand.

Now suppose that `main1` and `main2` both contain A. If `main1` and `main2` contain cuts from different branches of A, then there is no conflict.[3] Development for each of `main1` $+A$ and `main2` $+A$ proceeds in parallel. Eventually, the two branches of A may be merged, but until that happens, the two versions of A are independent.

It is *forbidden* for a cut of one catena to be contained by more than one other cut. In the example above, if `main1` and `main2` were allowed to contain the same cut of A, then all sorts of mysterious ways to shoot yourself in the foot would arise. Whenever a cut is contained by another catena as a sub-cut, then the super-cut must be unique, and the only way to modify the sub-cut is through it's unique super-cut. The sub-cut may not be modified directly (except to add and delete items), as a catena in its own right.

In summary, bear in mind the following, where A, B and C are different catenas.

- The uses hierachy can go to arbitrary depth, but circular references are not allowed. That is, "A uses B uses A" is forbidden.
- Parallel use is permitted. Something like "A uses (B and C)," together with "B uses C," is acceptable.
- "A depends on B contains C" is fine. "A depends on B" implies that the cut of B is fixed, so the cut of C must be fixed too. That "B contains C," instead of "B depends on C," must be a relic of how the particular cut of B was created.
- "A contains (B_1 and B_2)" is permitted, provided that B_1 and B_2 are different cuts. This is an odd thing to do, and it's possible only when any path-instances common to B_1 and B_2 are identical byte-instances, but it might make sense as a way to test merging B_1 and B_2.

5 Paths

It can be helpful to think of the contents of a repository as a parallel file system, with its own directory structure. The question is how the locations of the files in a repository map to the ordinary file system.

Every repository has a *base path*. The base path may be an absolute path (*i.e.*, a path starting at root) or it may be `'.'`. If the base path is an absolute path, then every file tracked by the repository must appear below the base path directory. If the

[3] Remember: if `main` contains A, then it must contain a proto-cut of A, not a fixed cut, and every branch has at most one proto-cut. So, if the cuts of A are different, then they must be from different branches of A.

base path is `'.'`, then every file tracked by the repository must be at or below the directory that contains the repository (*i.e.*, the `.igit` directory).

Every cut of a catena has a *home path*. The home path is always expressed relative to the base path, and may be equal to `'.'`. The home path may never be an absolute path. Every file held by a cut must be found in `base/home`. There may be an arbitrary directory structure below `base/home` for the files held by the cut, and this portion of the path to a file is called the *cut path*. For example,

```
base_path = /from/root/to/work/area
home_path = some/project
cut_path = the/actual/code/item.c
```

means that `item.c` will be found on the disk at

```
/from/root/to/work/area/some/project/the/actual/code/item.c
```

The base and home paths can be changed at any time. By changing the base path, the entire work-area for the repository changes; alternatively, if the base path is `'.'`, then moving the repository to a different location changes the work-area. Another reason to change the base path is because a repository was copied to a different machine.

IMPORTANT: When accessed through the command-line, you may use `'/'` or `'\'`; however, internally, igit uses `'/'` exclusively. It should never be necessary, but if you edit any of the `.igit` database files directly, then be careful to use `'/'` and never use `'\'`. And you had better understand *exactly* what you are doing!

6 Checking In and Out

Remember, a distinction is made between checking in files and cutting a new version. Here is the normal sequence of events.

1. You are working with a particular proto-cut on a catena.
2. You check out various files and edit them.
3. You check in files that belong to the catena. You can do this multiple times, and each time you do so, the proto-cut's current version of the file is updated.
4. You cut a new version. The last checkin for each file is used for this new cut. The cut is now fixed for all time and the head advances to a new proto-cut. This new proto-cut starts off looking identical to the cut you just made.

In theory, there's no reason to check in any given file more than once in step (3), but there are good practical reasons to do so. The repository may be treated as a kind of backup, so that it makes sense to do a checkin every night (say), no matter how broken the current state of the files may be. Or you might want to "put everything away" ("stash," in git jargon) so that you can work on a different branch. Or you might simply want a record of the file before you make a change.

For practical reasons, it's best not to create a new cut too often. These cuts are the primary window onto the catena's history, and it's easier to navigate history if each cut was done for a specific reason. The individual file checkins (step (3), above) allow comments too, so you're not losing the ability to express historical details by limiting the number of cuts.

Typically, if you want the most recent consistent or working version, then you don't want a proto-cut. Instead, you want the cut immediately prior to the proto-cut. The proto-cut could be full of half-finished files from step (3). Of course, this all depends on the system of hygiene you use for checkins and cuts.

7 Names and Branches

Every catena has an overall name, like `main` or `haskell_stuff`, and every cut may have an optional symbolic name defined by the user, like `last_before_vacation` or `version_2`. These symbolic names for cuts are called *tags*. Tags may not start with a digit, and may not include spaces, colons, slashes or back-slashes.

If a cut does not have a tag, then the cut can be identified by its ID number. These numbers are expressed in hexadecimal form and start with low numbers that grow over time as new cuts are created.[4]

If a cut splits into branches, then one of these branches is the *trunk*. Each branch is associated with an integer, 1, 2, *etc.*, and the trunk is branch 1. Branches of branches also have relative trunks, so there is no single trunk for a particular catena, although there is a *primary trunk*, which is defined as the "trunk of the trunk of the trunk," *etc.*

It may happen that you made a poor decision about which branch to treat as the trunk; e.g., it may turn out that what you thought was going to be an experimental version ends up being the production version. The `make-trunk` command allows you to change the trunk designation after the fact.

8 Merging

A set of cuts in the same catena can be merged to form a new proto-cut. Any cuts fed into a merger will be made fixed cuts, if they aren't fixed already.

The presence of sub-catenas adds a few twists. If a cut is used as a sub-cut, then it can be merged if and only if it is a fixed cut. Without this requirement, when looking back at history it would be unclear exactly *what* was merged. For similar reasons, any set of cuts to be merged must have identical home paths.

Furthermore, all cuts to be merged must have identical sub-cut hierarchies. For any depends-on sub-cuts, this means that the sub-cuts must be identical (have the same ID number). For contained sub-cuts, it means that the tree must have the same shape, and for each set of corresponding nodes of the tree, corresponding cuts must

[4] When an argument to a command is allowed to be a tag or a hexadecimal ID, then the tag lookup happens first. If one of your tags has a name like `abba`, which could be interpreted as a hexadecimal cut ID, and you want to refer to the cut with ID number `abba`, then prefix the hexademical value with a `0` to make it unambiguous.

be from the same catena. It's fine if the files in these sub-cuts have changed, and you may add and delete files from the sub-cuts, but the contain/depends-on trees must look the same.

Here's an example to illustrate why identical sub-cut hierarchies are required. Suppose that you want to merge C_1 and C_2, and that C_1 depends on D_1, and C_2 contains E_1 which also contains D_2, where $D_1 \neq D_2$. How should C_1 and C_2 be merged? There's no single approach that works for every scenario. If you do need to merge cuts that have different sub-cut hierarchies, then make them consistent before merging. In any reasonable case, this will require only a few minor adjustments before performing the merger.

The crucial rule to remember is that the merger isn't complete until every file in every cut feeding into the merger has been *reconciled.* When a merger is initiated, any file that differs from any of its "merge-partners" is marked as unreconciled. Merging is the process of working through the set of unreconciled files until they have all been reconciled. The final act of a merger is always to invoke `merger-complete`.

9 File History and Shuffling

In addition to the historical record based on cuts and catenas, there is an independent file-oriented record of history. A *shuffle* allows file-level historical relationships to be noted.

When files are checked in, there's a relatively obvious way that a newly checked in file depends on a previous version. If a file is checked in at a particular cut path, then that file depends on the previous check-in at that cut path, whether the previous check-in was made to the same cut or a previous cut. In a similar way, a file depends on any versions of the file feeding into a merger. These relationships are noted automatically.[5]

When refactoring, it's common to "shuffle the guts" of some files. Simple examples are taking a file and breaking it into two, or combining two files into one. More generally, you might start with N files, do a massive rearrangement, and end up with M files, perhaps using entirely different names, and moving them to different directories. The `shuffle` command allows you to make note of these relationships.

The use of shuffle specifications is entirely optional, but it can be handy, bearing in mind that there's a gray area. Copying and pasting a few lines from file A to file B probably isn't important, but moving an entire function might be.

10 Commands

Nearly all commands work with the active respository, and most commands work with the cut at a catena's head. If a command takes *catena* as an argument, then *catena* should be the symbolic name for a catena, and the command works on the cut

[5]When a file is checked in, the cut on which it was checked in is noted. If the file remains unchanged over a series of cuts, then the file's history has no record of the fact that the file is held by these intermediate cuts. This remains true over a series of branches and mergers.

at the given catena's head. The cut at a catena's head is sometimes called the *active* cut or the *current* cut.

`help`
: Prints a list of commands with very brief help.

`init`
: After setting up the `igit_home` environment variable, call this to create the contents of the directory to be found at `igit_home`.

  This is always the first thing you must do.

`create` ⟨*dir_name*⟩ ⟨*sym_name*⟩
: Create a new empty repository in the current directory, and make it the active repository. The base path is set to `'.'`.

  Both arguments are optional. By default, the directory will be called `.igit`, but you can make it anything you like with *dir_name*. So that you don't accidentally try to check the repository into itself, it's probably best to start *dir_name* with a period (`.igit` instead of `igit`). The `ignore` file, found in `igit_home`, is set up to ignore any file or directory whose name starts with a period.

  Every repository can have a symbolic name, which is optionally provided with *sym_name*. Without a symbolic name, the name for a repository is the path to it. Using symbolic names allows you to refer to `jim_repo` (say), instead of having to type `long/path/to/stuff/from/jim`. Symbolic names may not start with a digit.

`add-repo` ⟨*dir_path*⟩ ⟨*sym_name*⟩
: Similar to `create`, but it assumes that the repository already exists. The existing repository directory is added to `.igithome`.

  The *sym_name* is optional, but *dir_path* is required. The *dir_path* can be specified in relative terms; the working directory does not need to contain the repository directory.

  This command does not make the newly added repository the active repository.

`delete-repo` ⟨*name*⟩
: Remove a repository from `igit_home`. The *name* can be a symbolic name, if one is defined, or a path to the repository directory. If *name* is given as a path, then it can be given in relative terms.

  As with `add-repo`, this is merely editing `igit_home/list`. You could edit it directly (if you're careful) to accomplish the same thing. In particular, it does not delete the repository from the file system.

`rename-repo` ⟨*old_name*⟩ ⟨*new_name*⟩
: This is exactly the same as running `delete-repo`, followed by `add-repo`. The *old_name* may be a symbolic name or a path. The *new_name* is optional. If *new_name* is omitted, then this simply forgets any symbolic name for the path (although the repository remains known to the system); otherwise, *new_name* becomes the new symbolic name.

`list-repos`
: Lists all known repositories.

`use-repo` ⟨*name*⟩
: Make the given repository active. *name* may be a symbolic name or a path leading to the repository directory.

`new-catena` ⟨*name*⟩
: Define a new catena with the given name.

`list-catena`
: Prints the names of all known catenas. See below for `list-catena` *catena*.

`delete-catena` ⟨*name*⟩
: Be careful with this since everything about the catena will be gone. It all remains as orphans, but it would be hard to recover. Another reason to be careful with this is that a catena can only be deleted if no cut in the catena appears as a sub-catena elsewhere. Checking this requires visiting every cut of the catena, and that could take some time.

`set-base` ⟨*path*⟩
: The base path is used repository-wide (see page 4). If *path* is not `'.'`, then it will be converted to an absolute path.

`get-base`
: Reports the value set by `set-base`.

`set-home` ⟨*catena*⟩ ⟨*path*⟩
: Every cut of a catena has a home path (see page 4).

`get-home` ⟨*catena*⟩
: Reports the value set by `set-home`.

`add-depends` ⟨*catena*⟩ ⟨*sub-catena*⟩
: Make the cut at *catena*'s head depend on the cut at *sub-catena*'s head.

  The system checks whether there are any files common to the two catenas (*i.e.*, they have the same path-instance), and this command fails if the byte-instances are not identical.

`add-contains` ⟨*catena*⟩ ⟨*sub-catena*⟩
: Like `add-depends`, but with a contains relationship.

  Once *sub-cantena* is part of *catena* (either depends on or contains), you can use `add-depends` or `add-contains` at any time to change the relationship.

`list-subs` ⟨*catena*⟩
: Show a tree of all sub-catenas of the given catena.

`add` ⟨*catena*⟩ ⟨*path*⟩ ⟨*optional_comment*⟩
: Make the file at the given *path* part of the current cut of *catena* and check it in. The file must exist before this command is invoked.

  The *path* must express the location of the desired file in the usual way (*i.e.*, as a path from the current working directory). Igit will resolve *path* to a cut path relative to `base` and `home`.

Consider an example to make this clear. Suppose that the *path* is given as `initial/bit/more/bits/to/the_file`.

- If `base/home` is `/from/root/then/initial/bit`, then the file will enter the cut under `more/bits/to/the_file`. That is, the cut path will be `more/bits/to`.
- If `base/home` ends with `something/else`, so that there are no directories in common between the end of `base/home` and the beginning of *path*, then the file will enter under `initial/bit/more/bits/to/the_file`. The entire *path* becomes the cut path.
- If either base or home is `'.'`, then `base/home` is resolved in the obvious way to an absolute path, and the two rules above are applied.

`add-as` ⟨*catena*⟩ ⟨*path_1*⟩ ⟨*path_2*⟩

Works like `add` ⟨*catena*⟩ ⟨*path_1*⟩, except that the file is stored within the catena using the cut path *path_2*. This side-steps any issues with the base and home paths.

`add-dir` ⟨*catena*⟩ ⟨*path*⟩ `-f -e` ⟨*excluded_paths*⟩

Like `add`, but *path* must be a directory, and everything in the directory is added, recursively. Be careful since this will add everything that's not explictly excluded by the `ignore` file.

There are two optional switches, and if both switches are used, then `-f` must come before `-e`. The `-f` switch ("files only") means not to recurse into any sub-directories of *path*. Use the `-e` switch, with a list of paths to files and directories to exclude the indicated items. These excluded paths should be given relative to *path*; for example,

`add-dir` ⟨*catena*⟩ `some/path -e dir/item.c`

means to exclude the file found at `some/path/dir/item.c`.

There is no opportunity for a checkin comment since the number of files could be large.

`remove` ⟨*catena*⟩ ⟨*path*⟩

Opposite of `add`. The given *path* must be a cut path. All record of the file is deleted from the current cut. Any previous check-ins made for the file on the current cut remain, although they will only be accessible by walking the file history starting from an earlier cut.

`remove-dir` ⟨*catena*⟩ ⟨*path*⟩

Like `remove`, but recursive for an entire directory.

`remove-sub` ⟨*catena*⟩ ⟨*sub-catena*⟩

Remove the dependence of *catena* on *sub-catena* for the current cut.

`rename` ⟨*catena_1*⟩ ⟨*catena_2*⟩

To rename a catena. These names are purely symbolic, so this change happens everywhere, including earlier cuts, depends-on relationships, *etc.*

`rename` ⟨*catena*⟩ ⟨*path_1*⟩ ⟨*path_2*⟩ ⟨*comment*⟩

Within the given *catena*, change a file location from *path_1* to *path_2*, providing an optional comment. Both of the path arguments must be cut paths.

This command makes internal changes to the repository, but it is the user's responsibility to rename the file on the hard drive. Simply checking out the file from *path_2* (after invoking this command) would work, but any changes made to the file, as stored at *path_1* on the hard drive would be lost if those changes have not been checked in before calling this command.

This command exists so that the system can note the fact that a file checked in from *path_2* was obtained by editing the file that used to be stored at *path_1*. If, instead, you use `remove` on *path_1*, followed by `add` on *path_2*, then the record of the historical relationship between the two versions is lost. So this is like a tiny `shuffle` (see below).

`checkin` ⟨*catena*⟩ ⟨*path*⟩ ⟨*optional_comment*⟩

Check in the version of the file at *path* to the current cut of *catena* (which must be a proto-cut). The *path* may be an explicit cut path, or it will be resolved to the correct cut path. This cut path must be known to the catena from an earlier `add`, `add-dir` or `rename`.

The *catena* may not be a sub-catena. If *path* belongs directly to catena, then it's clear what this will do. If *path* belongs to a sub-catena of *catena*, then the command will check in on the sub-catena (or multiple sub-catenas if the path appears multiple times).

If the optional comment is omitted, then there is an interactive prompt. You can check in an identical version of the same file multiple times. Doing this may make sense if you want to add more check-in comments.

`checkin-all` ⟨*catena*⟩ ⟨*optional_comment*⟩

As above, but this checks in every file tracked by *catena*, including sub-catenas.

Paths are dealt with a little differently here. Whereas `add` and `checkin` first look on the file system to find the file in question, and make the path to the file conform to the existing `home/base`, this command works the other way. For each file, the location to which that file would be written (`base/home/cut_path`) is checked for a file, and that file is checked in. So `base/home` needs to point to the location from which the files are to be checked in.

There's no provision for comments as each file is checked in since there could be many files. Instead, there is one comment shared by the entire set. This comment is stored with the (proto) cut into which the files are checked in. So a series of calls to `checkin-all` extends the comment to be longer and longer, and you can edit earlier comments due to `checkin-all` up until the point where `cut` is called.

`checkin-as` ⟨*catena*⟩ ⟨*path_1*⟩ ⟨*path_2*⟩ ⟨*date*⟩

Treat the file at *path_1* as though it were really at *path_2*, and the checkin occurred on the given date. If *path2* is not yet known to the catena, then it will be added. If *path_2* is already known to the catena, then this action will be rejected if *date* is out of sequence. Time travel is OK, but not in a way that leads to paradoxes.

This function exists for the situation in which you want to create an igit repository using a pre-existing set of files.

`checkout` ⟨*catena*⟩ ⟨*path*⟩

Copies the file found at *path* to the file system. The *path* must be a cut path, and some care may be needed. If *catena* has several sub-catenas, then *path* could lead to different files in each sub-catena, in which case all of the files will be checked out. This would happen if two sub-catenas have a parallel directory structure and file names.

Warning: This overwrites existing files with no warning.

`checkout-to` ⟨*catena*⟩ ⟨*path_1*⟩ ⟨*path_2*⟩

Like `checkout`, but the file (or files) is (or are) copied to *path_2*, which is not resolved in any way; it's either an absolute path or it's given relative to the current working directory – and it must include the file name.

`checkout-all` ⟨*catena*⟩

Copies every file from *catena* and its sub-catenas.

`checkout-all-to` ⟨*catena*⟩ ⟨*path*⟩

Like `checkout-all`, but it goes to *path*, which must be a directory.

`cut` ⟨*catena*⟩ ⟨*optional_comment*⟩

Fix the state of the proto-cut at *catena*'s head, create a new proto-cut and move the head to the new proto-cut. The generates new cuts for any sub-catenas too, and moves their heads forward.

If the comment is not provided with the command, then it will be requested interactively.

`list-catena` ⟨*catena*⟩

The argument is optional. Without any argument, this lists all known catenas (see above). If *catena* is specified, then it lists the files and any sub-catenas of the current cut of *catena*.

`branch` ⟨*catena*⟩ ⟨*n*⟩
`bud` ⟨*catena*⟩ ⟨*n*⟩

These two commands do the same thing, but in different scenarios. If head points to a proto-cut, use `branch`, and if head points to a fixed cut, use `bud`. Distinguishing these cases makes mistakes less likely.

The second argument, *n*, is optional. For `branch`, the default value is 2, and the effect is to `cut` the current proto-cut, then create *n* new proto cuts as descendants. For `bud`, the default value is 1. All of these new branches start off identical.

For `branch`, one of the branches starts off as the trunk, and head moves to a non-trunk branch. It moves to a non-trunk because, presumably, the `branch` was performed because you want to try something experimental, and the right place for experiments is not on the trunk. For `bud`, the original cut (where head points) is made the trunk, and the head is moved to one of the branches.

When *catena* contains a sub-catena, the sub-catena branches too (recursively).

`head-id` ⟨*catena*⟩

Print the ID number of the current location of *catena*'s head. The value associated with a particular cut never changes.

`tag` ⟨*catena*⟩ ⟨*name*⟩ ⟨*location*⟩

Associate *name* with a particular cut. The *location* is optional; if it is omitted, then *catena*'s head gets this tag. If *location* is given, then it should be a hexadecimal ID number.

Each catena has its own namespace for these tags, so the same *name* can be used in different catenas.

`untag` ⟨*catena*⟩ ⟨*name*⟩

Delete a tag. Opposite of `tag`.

`list-tags` ⟨*catena*⟩

Print all the tags defined with `tag` (in no particular order). The ID number is given, then the tag.

`head-to` ⟨*catena*⟩ ⟨*location*⟩

Move the head of *catena* to the given *location*, where *location* is a hexadecimal ID number or symbolic tag. This does not move the head for any sub-catenas of *catena*.

`head-forward` ⟨*catena*⟩ ⟨*step*⟩

Move *catena*'s head forward; how far forward depends on *step*. If *step* is omitted, then step forward to the next cut; if *step* is an integer then step forward by that many cuts, but stop if the next step requires choosing among different branches (and print the choices). If *step* is `'f'`, then step forward to the next proto-cut, but stop if you hit a branch-point. If *step* is `'F'`, then step to the next proto-cut, choosing the trunk at each branch.

For both `head-forward` and `head-back` (below), the head for any sub-catenas of *catena* is moved too.

`head-back` ⟨*catena*⟩ ⟨*step*⟩

Like `head-forward`, but stop when the next step requires choosing among merged cuts. For `head-back`, *step* must be an integer and may not be `'f'` or `'F'`. There is no "backward-looking trunk," so there is no clear way to make a default choice among merged cuts.

`merge` ⟨*catena*⟩ ⟨*branch_1*⟩ ⟨*branch_2*⟩ ... ⟨*branch_n*⟩
`merge-to` ⟨*catena*⟩ ⟨*branch_1*⟩ ⟨*branch_2*⟩ ... ⟨*branch_n*⟩

The `merge` command takes at least two *branch* arguments specifying which cuts are to be merged to form a new proto-cut. The *branch* values may be given as tags or as cut ID numbers. The initial location of *catena*'s head doesn't matter for `merge`, but the head will point to the merged cut after `merge`.

The `merge-to` command merges the given *branch* values to *catena*'s head, which must be a proto-cut. So, you can use `merge` to bring together a set of cuts, then add more cuts to the merger with `merge-to`. Each `merge` generates a new proto-cut, while `merge-to` adds to an existing proto-cut. One way to approach a big octopus merger is to `merge` a few cuts, reconcile everything, add a few more cuts with `merge-to`, reconcile the new additions, *etc.*

With each `merge` or `merge-to`, the set of newly added files is examined. If there are any new path-instances, then an arbitrary byte-instance of that path-instance

is copied into the destination proto-cut, and the source for that byte-instance is marked as reconciled. Each of the byte-instances being merged will be compared to the byte-instances already in the proto-cut, and any new ones will be flagged as unreconciled.

The commands described below help to reconcile the files. These commands are `unreconciled`, `reconcile`, `force-reconcile`, `get-versions` and `done-merge`. In particular, you'll always need to call `merger-complete` as the final step.

Certain commands can't be used on a cut until every file has been reconciled. The obvious one is `cut`; others are `set-home`, `add-depends`, `add-contains`, `remove-sub`, `branch` and `bud`. You should think hard before using certain other commands if there's a merger open, notably `delete-catena`, `remove`, `remove-dir` and `rename`. In general, the best strategy is to do the absolute minimum until everything is reconciled, then immediately do a `cut`.

`unreconciled` ⟨*catena*⟩

If *catena*'s head points to a cut undergoing a merger, then this lists any unreconciled items. The list takes the form `ID:path`, where `ID` is the ID number for cut containing the file, and `path` is the cut path.

`reconcile` ⟨*catena*⟩ ⟨*path*⟩

Attempt to automatically reconcile differences for the given cut path. If the optional *path* is omitted, then try to reconcile all items.

`force-reconcile` ⟨*catena*⟩ ⟨*path*⟩ ⟨*id*⟩

Mark the given cut path as reconciled, ignoring any apparent discrepencies. The *id* is a cut ID or symbolic tag, and if *id* is omitted, then this marks all items at *path* as reconciled. If both *path* and *id* are omitted, then the entire merger is considered to be reconciled.

This is command should not be used often, but it may be handy if you know that a particular file is junk – maybe it was an experiment that didn't work out.

`get-versions` ⟨*catena*⟩ ⟨*path*⟩

Copy out all byte-instances associated with the given cut path. They'll appear as `file_name.X.suffix`, where `X` is the cut ID from which the byte-instance came. For comparison, this checks out the "already reconciled" version too. If *path* is omitted, then this will copy out all unreconciled files.

For example, if there are three versions of `stuff.c`, then they might appear as `stuff.189.c`, `stuff.431.c` and `stuff.11.c`, plus `stuff.c` for what has been reconciled so far. If several of the prior versions are identical, then only one of them will be copied out; e.g., if `stuff.189.c` and `stuff.431.c` are identical, then only one of them will appear.

`done-merge` ⟨*catena*⟩ ⟨*path*⟩

After reconciling the files from `get-versions`, call this to delete all of the copies created by `get-versions` and mark all instances of that path as reconciled.

Note that *path* is required, unlike for `get-versions`. This prevents accidentally closing out a merger due to a typo.

`merger-complete` ⟨*catena*⟩

Call this when every path has been reconciled. It will fail if they haven't actually been reconciled.

`shuffle` ⟨*catena*⟩ ⟨*path_1*⟩ ⟨*path_2*⟩ ... ⟨*path_n*⟩

path_1 must appear in the current cut, the other paths must appear in the unique previous cut, and they must all be given as cut paths. This command says that *path_1* came from the other items (in addition to any other items that are already known). Use `shuffle-merge` when the current cut is the result of a merger.

`shuffle-merge` ⟨*catena*⟩ ⟨*path_1*⟩ ⟨*path_2*⟩ ... ⟨*path_n*⟩

Like `shuffle`, but where *catena*'s head is a cut formed by a merger. In this case, there is no unique prior cut, and the paths need to include information about which of the merged branches is intended. Each *path* (other than *path_1*) may start with the cut ID or tag for a cut feeding into the merger, with a colon (`':'`) appearing before the cut path. If the cut ID or tag is omitted, then every instance of the file from all prior cuts is considered to be an antecedent of *path_1*.

`shuffle-arbitrary` ⟨*catena*⟩ ⟨*path_1*⟩ ⟨*path_2*⟩... ⟨*path_n*⟩

This takes the same arguments as `shuffle-merge`, but a cut ID or tag is required for every path after *path_1*. Also, `shuffle-merge` requires that the cuts appearing in each *path* be part of a merger into the cut containing *path_1*; `shuffle-arbitrary` does not.

`make-trunk` ⟨*catena*⟩

Move the head to a cut immediately after a branch, and call this to make the head location the trunk.

`seek-new` ⟨*catena*⟩

Look at the ordinary file system under `home/base` for *catena*'s head and report any files that are not being tracked by the repository.