Principia Softwarica: The Version Control System `git9`

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Ori Bernstein

May 14, 2026

The text and figures are Copyright © 2014–2026 Yoann Padioleau.
All rights reserved.

The source code is Copyright © 2019 Ori Bernstein.
MIT license.

Contents

1 Introduction **9**
- 1.1 Motivations . . . 9
- 1.2 The version control system Git (and `git9`) . . . 9
- 1.3 Other version control systems . . . 10
- 1.4 Getting started . . . 12
- 1.5 Requirements . . . 12
- 1.6 About this document . . . 12
- 1.7 Copyright . . . 12
- 1.8 Acknowledgments . . . 12

2 Overview **14**
- 2.1 Version control system principles . . . 14
  - 2.1.1 Storing past versions . . . 14
  - 2.1.2 Tracking changes . . . 15
  - 2.1.3 Concurrent development . . . 15
  - 2.1.4 Parallel development . . . 15
  - 2.1.5 Centralized versus distributed . . . 15
- 2.2 `git` command-line interface . . . 15
- 2.3 Git concepts and data structures . . . 16
  - 2.3.1 Object store . . . 16
  - 2.3.2 References . . . 18
  - 2.3.3 Staging area . . . 18
- 2.4 Repository format: `.git/` . . . 20
- 2.5 `hello/.git/` . . . 21
  - 2.5.1 Creating a repository . . . 21
  - 2.5.2 Staging a diff . . . 22
  - 2.5.3 Committing a diff . . . 22
  - 2.5.4 Managing branches . . . 23
  - 2.5.5 Inspecting objects . . . 24
  - 2.5.6 Other commands . . . 25
- 2.6 Code organization . . . 25
- 2.7 Software architecture . . . 25
- 2.8 Book structure . . . 25

3 Core Data Structures **26**
- 3.1 Secure Hash Algorithm (SHA1) hashes . . . 26
  - 3.1.1 Binary hashes . . . 26
  - 3.1.2 Hexadecimal hashes . . . 26
  - 3.1.3 Conversion functions . . . 26
- 3.2 `Repository.t` . . . 26

3.3 `Objects.t` . . . 26
3.3.1 `Blob.t` . . . 26
3.3.2 `Tree.t` . . . 26
3.3.3 `Commit.t` . . . 26
3.4 References . . . 26
3.4.1 Reference names . . . 26
3.4.2 `HEAD` . . . 26
3.4.3 Reference content . . . 26
3.4.4 Reference store . . . 26
3.4.5 Objectish references . . . 26
3.5 `Index.t` . . . 26

4 Reading from a Repository **27**
4.1 Objects . . . 27
4.1.1 Decompression . . . 27
4.1.2 Reading a blob . . . 27
4.1.3 Reading a tree . . . 27
4.1.4 Reading a commit . . . 27
4.2 References . . . 27
4.3 Index . . . 27

5 Writing to a Repository **28**
5.1 Objects . . . 28
5.1.1 Compression . . . 28
5.1.2 Locking . . . 28
5.1.3 Writing a blob . . . 28
5.1.4 Writing a tree . . . 28
5.1.5 Writing a commit . . . 28
5.2 References . . . 28
5.3 Index . . . 28

6 Main Functions **29**
6.1 Main commands . . . 29
6.2 `Main.main()` . . . 29
6.3 Getting help: `git help` . . . 29

7 Creating a Repository **30**
7.1 Initializing a new repository: `git init` . . . 30
7.2 Copying an existing repository: `cp -r` . . . 31
7.3 Cloning an existing repository: `git clone` . . . 31
7.4 Opening a repository . . . 31

8 Staging a Diff **32**
8.1 Adding files: `git add` . . . 32
8.2 Removing files: `git rm` . . . 33
8.3 Renaming files: `git mv` . . . 33

9 Committing a Diff **34**
9.1 Committing the index: `git commit` . . . 34
9.2 Computing the trees from an index . . . 38
9.2.1 Grouping index files by directories . . . 38
9.2.2 Building trees from directories . . . 38
9.3 Storing the date, author, and committer . . . 38
9.4 Recording the message . . . 38
9.5 Updating `HEAD` atomically . . . 38
9.5.1 Creating a new reference (first commit) . . . 38
9.5.2 Updating an existing reference . . . 38

10 Branching **39**
10.1 Listing branches: `git branch` . . . 41
10.2 Creating a branch: `git branch <name>` . . . 41
10.3 Deleting a branch: `git branch -d <name>` . . . 41
10.4 Checking out a branch: `git checkout` . . . 41
10.4.1 Computing the index (and work tree) from trees . . . 41
10.4.2 Creating a file from a blob . . . 41
10.5 Resetting a branch: `git reset` . . . 41

11 Merging **42**
11.1 merging branches: `git merge` . . . 42
11.2 Merging two trees . . . 43
11.3 Merging two files . . . 43
11.4 `diff3` and `merge` . . . 43
11.5 Merging conflicts . . . 43
11.6 Committing merged branches: `git commit` . . . 43

12 Inspecting **44**
12.1 Showing the content of an object: `git show` . . . 44
12.1.1 Showing a blob . . . 44
12.1.2 Showing a tree . . . 44
12.1.3 Showing a commit . . . 44
12.2 Representing changes . . . 44
12.2.1 Tree changes . . . 44
12.2.2 File changes . . . 44
12.3 Showing file differences: `git diff` . . . 44
12.3.1 Computing tree changes . . . 46
12.3.2 Computing file changes . . . 46
12.3.3 Showing changes . . . 46
12.4 Commit history walker . . . 46
12.4.1 Basic walker . . . 46
12.4.2 Extra features . . . 46
12.5 Showing the commit history: `git log` . . . 46
12.5.1 Printing a commit . . . 46
12.5.2 Diff summary: `git log --name-status` . . . 46
12.5.3 Comparing trees . . . 46
12.6 Showing the status of a file: `git status` . . . 46
12.6.1 Listing staged modifications . . . 46
12.6.2 Listing unstaged modifications . . . 46

12.6.3 Listing untracked files . . . 46
12.7 Archeology . . . 46
12.7.1 Checking out an old version . . . 46
12.7.2 Showing an old version of a file . . . 46
12.7.3 Showing differences between past versions . . . 46
12.7.4 Showing the history of a file or directory: `git log <path>` . . . 46

13 Packing **47**
13.1 `Pack.t` . . . 47
13.1.1 Pack data . . . 47
13.1.2 Pack Index . . . 47
13.2 IO . . . 47
13.2.1 Reading a pack . . . 47
13.2.2 Writing a pack . . . 47
13.3 Modifications to previous functions and data structures . . . 47
13.3.1 Object Store . . . 47
13.3.2 Refs . . . 47
13.4 Delta . . . 47
13.5 Pack commands . . . 47
13.5.1 `git pack-objects` . . . 47
13.5.2 `git repack` . . . 47
13.5.3 `git fetch-pack` . . . 47
13.5.4 `git receive-pack` . . . 47

14 Exchanging **48**
14.1 Client/server architecture . . . 48
14.2 Pulling updates from another repository: `git pull` . . . 48
14.2.1 Fetching objects locally . . . 50
14.2.2 Fetching references . . . 50
14.2.3 Fetching objects in a pack . . . 50
14.3 Object store walker . . . 50
14.4 Pushing changes to another repository: `git push` . . . 50
14.4.1 Sending objects locally . . . 51
14.4.2 Sending objects in a pack . . . 51
14.5 Cloning a repository: `git clone` . . . 51
14.5.1 Fetching objects . . . 53
14.5.2 Importing references . . . 53

15 Networking **54**
15.1 `git://` protocol . . . 54
15.1.1 Reading packets . . . 54
15.1.2 Writing packets . . . 54
15.2 Capabilities . . . 54
15.3 `git://` client . . . 54
15.3.1 Fetching remote objects . . . 54
15.3.2 Fetching pack files . . . 54
15.3.3 Sending pack files . . . 54
15.4 `git://` server . . . 54
15.4.1 `git-upload-pack` . . . 54
15.4.2 `git-receive-pack` . . . 54

16 Algorithms **55**
16.1 SHA1 . . . 56
16.1.1 Overview . . . 56
16.1.2 32 bits operators . . . 56
16.1.3 Entry point: `sha1()` . . . 56
16.1.4 Endianess . . . 56
16.1.5 Filling blocks . . . 56
16.1.6 Looping over blocks . . . 56
16.1.7 Padding . . . 56
16.2 Unzip . . . 56
16.2.1 Overview . . . 56
16.2.2 Data structures . . . 56
16.2.3 Error management . . . 56
16.2.4 IO helpers . . . 56
16.2.5 Huffman trees . . . 56
16.2.6 Sliding window back references (LZ77) . . . 56
16.2.7 Entry point: `inflate()` . . . 56
16.2.8 `inflate_loop()` . . . 56
16.2.9 Advanced features . . . 57
16.2.10 Optimizations . . . 57
16.3 Zip . . . 57
16.3.1 Entry point: `deflate()` . . . 57
16.4 Diff . . . 57
16.4.1 Overview . . . 57
16.4.2 Data structures . . . 57
16.4.3 Entry point: `Diffs.diff()` . . . 57
16.4.4 Edit distance basic algorithm (O(n^2) in space/time) . . . 57
16.4.5 Myers algorithm (O(n) in space in best-case) . . . 57
16.5 Diff3 . . . 57
16.5.1 Overview . . . 57
16.5.2 Data structures . . . 57
16.5.3 Entry point: `diff3()` . . . 57
16.5.4 Displaying merging conflicts . . . 57

17 Advanced Features TODO **58**
17.1 Tracking symbolic links . . . 58
17.2 Splitting a large repository: submodules . . . 58
17.3 Configuration file: `.git/config` . . . 58
17.3.1 Initialization . . . 58
17.3.2 `Config.t` . . . 58
17.3.3 Reading a configuration file . . . 58
17.3.4 Writing a configuration file . . . 58
17.3.5 Stacked configurations: `/.gitconfig` and `.git/config` . . . 58
17.4 Commit rules and checks: `.git/hooks/` . . . 58
17.5 Ignoring files: `.gitignore` . . . 58
17.6 Archeology: Grafts . . . 58
17.6.1 Reading a graft file . . . 58
17.6.2 Writing a graft file . . . 58
17.6.3 Using a graft file . . . 58
17.7 Rename detection . . . 58

18 Advanced Commands TODO **59**
18.1 Tagging a commit with a name . . . 59
18.1.1 `Tag.t` . . . 59
18.1.2 Reading a tag . . . 59
18.1.3 Writing a tag . . . 59
18.1.4 `git tag` . . . 59
18.1.5 `git clone` and tags . . . 59
18.1.6 `git pull` and tags . . . 59
18.2 Reference history . . . 59
18.2.1 `git reflog` . . . 59
18.3 Stash . . . 59
18.3.1 `git stash` . . . 59
18.4 Developer commands . . . 59
18.4.1 Finding the diff causing a regression: `git bisect` . . . 59
18.4.2 Reverting a change: `git revert` . . . 59
18.4.3 Finding the author of some code: `git blame` . . . 60
18.4.4 Finding code: `git grep` . . . 60
18.4.5 Creating an archive: `git archive` . . . 60
18.5 Advanced merging commands . . . 60
18.5.1 Rebase: `git rebase` . . . 60
18.5.2 Cherry-pick: `git cherry-pick` . . . 61
18.6 Plumbing commands . . . 61
18.6.1 `git fetch` . . . 61
18.6.2 `git symbolic-ref` . . . 61
18.6.3 `git rev-list` . . . 61
18.6.4 `git diff-tree` . . . 61
18.6.5 `git commit-tree` . . . 61
18.6.6 `git ls-tree` . . . 61
18.6.7 `git remote add` . . . 61
18.6.8 `git ls-remote` . . . 61
18.7 Miscellaneous commands . . . 61
18.7.1 `git filter-branch` . . . 61

19 Advanced Networking TODO **62**
19.1 Other clients . . . 62
19.1.1 `ssh://` . . . 62
19.1.2 `http://` . . . 62
19.2 Other servers . . . 62
19.2.1 `http://` . . . 62
19.3 Other client/server capabilities . . . 62
19.3.1 Report status . . . 62
19.3.2 Quiet . . . 62
19.3.3 Thin pack . . . 62
19.3.4 Side band 64k . . . 62

20 Advanced Topics TODO **63**
20.1 Bytes versus ASCII versus UTF-8 . . . 63
20.2 Reliability and signals . . . 63
20.3 Optimizations . . . 63
20.4 Fast import and export . . . 63

20.5 Alternates . . . 63
20.6 Other repository format . . . 63
20.6.1 Bare repository . . . 63
20.6.2 `.git` file . . . 63
20.7 Security . . . 63

21 Conclusion **64**

A Debugging **65**

B Profiling **66**

C Error Management **67**

D Utilities **68**

E Extra Code **69**
E.1 `git9/` . . . 69
E.2 `diff/` . . . 237
E.3 `misc/` . . . 264

Glossary **280**

Index **281**

References **281**

Chapter 1

Introduction

The goal of this book is to explain with full details the source code of a version control system.

1.1 Motivations

Why a version control system (VCS)? Because I think you are a better programmer if you fully understand how things work under the hood, and a VCS is one of the tools a programmer uses the most. Indeed, programmers use VCSs—which manage efficiently changes to a file or set of files—to manage the vital product of their labour: source code.

VCSs allow to store and retrieve past versions of a file, and to record who made each change, when, where, and why. VCSs are mostly used by programmers but you can use a VCS to manage any text-based documents (e.g., the source of this book), configuration files, and even binaries.

For projects with a single developer, a VCS is useful to keep track of changes: it allows the programmer to easily go back to past versions if he messed things up. For projects with multiple developers, a VCS is almost mandatory: it allows programmers to collaborate with each other and even to work on and modify the same files concurrently.

Here are a few questions I hope this book will answer:

- What are the fundamental concepts of a VCS? What are the core algorithms behind a VCS?
- How does a VCS store efficiently multiple versions of a file?
- How does a VCS represent changes to a file, or changes to a set of files? How does a VCS represent the addition, deletion, or renaming of a file?
- How does a VCS allow multiple users to work on and modify the same files at the same time? How can a VCS help coordinate the actions of multiple users?
- How does a VCS support parallel developments in a project? How can some developers work on the main release, some developers on experimental features, and others on fixing bugs on previously shipped releases all at the same time? What is a branch? How does a VCS reconcile multiple branches?
- What is a merge algorithm? When is a merge safe? What is a merge conflict? How does a VCS resolve conflicts?

1.2 The version control system Git (and `git9`)

There are many open source UNIX VCSs (Subversion, Mercurial, etc.) with different user interfaces, storage strategies, concurrency models, or features. I will present a few of those VCSs in Section 1.3. For this book,

I decided to base the presentation on Git[1] because it is one of the most popular VCSs in the open source community. Moreover, when coupled with the hosting website GitHub[2], Git makes it really easy for people to collaborate with each other.

However, Git is a rather large project with more than 200 000 LOC. It is impossible to present all this code in a book of a reasonable size. There are a few clones of Git written in higher-level languages than C, for example, Dulwich[3] written in Python with only 16 000 LOC. I could have based this book on Dulwich, but this would introduce another language in the Principia Softwarica book series in addition to C. This would also in turn require to present the code of the Python interpreter, which contains more than 170 000 lines of C code.

Fortunately, in 2021 Ori Bernstein decided to rewrite Git for Plan 9 and redesigned the program to use far less code. This is why I will explain in this book the code of `git9`[4], which contains about 10 000 lines of code (LOC); most of the code is written in C but about 1000 LOC is written in `rc`, the Plan 9 shell (see SHELL book [Pad18]).

Like for most books in Principia Softwarica, I chose a Plan 9 program because those programs are simple, small, elegant, open source, and they form together a coherent set. Note that `git9` was not part of the original Plan 9 but `git9` follows the Plan 9 philosophy and coding style.

1.3 Other version control systems

Here are a few VCSs that I considered for this book, but which I ultimately discarded:

- The Revision Control System (RCS) [Tic85][5] was the first open source VCS. Like its predecessor, the Source Code Control System (SCCS) [Roc75][6], RCS uses *deltas* to store efficiently the past versions of a file (Section 16.4 will present the Diff algorithm to compute those deltas). However, RCS improves over SCCS by using *reverse-deltas* to enable the user to also retrieve quickly the last version of a file (a recurring operation).

  RCS is still a reasonable choice to manage a small project with a single developer or a small set of developers who can share access to a common directory (for example, by using NFS and symbolic links). However, the use of *locks* in RCS to forbid multiple developers to modify the same file at the same time and its focus on individual files make RCS inadequate for large projects with many independent developers. Moreover, even if RCS is very limited compared to modern VCSs like Git, RCS still contains more than 17 500 LOC.

- The Concurrent Versions System CVS [Gru86][7] was the most popular VCS for over a decade before distributed VCSs (e.g., Git) took over. CVS introduced a few important innovations. First, CVS was designed to operate on a set of files at once[8], with files possibly organized in a tree. With CVS, you can group together related changes to a set of files and you can easily retrieve past versions of a whole project. Secondly, CVS allows multiple users to work on and modify the same files *concurrently* (hence the 'C' in CVS). CVS relies on the `merge` program instead of locks (Section 16.5 will present the Diff3 algorithm which underlies the `merge` program). Finally, CVS introduced later in 1995 a *client/server* architecture where a *repository* could be stored on a remote machine. When coupled with the free hosting site Sourceforge[9], CVS allowed developers to easily start and collaborate on new projects.

  CVS started as a few shell scripts using RCS as a backend. However, today CVS is a very large C program with more than 80 000 LOC. The backend file `src/rcs.s` contains already 9000 LOC.

[1] https://git-scm.com/
[2] https://github.com
[3] https://www.dulwich.io/
[4] https://github.com/oridb/git9
[5] https://www.gnu.org/software/rcs/
[6] http://sccs.sourceforge.net/
[7] http://www.nongnu.org/cvs/
[8] RCS can operate on multiple files by using shell wildcards, but it was not designed for it.
[9] https://sourceforge.net

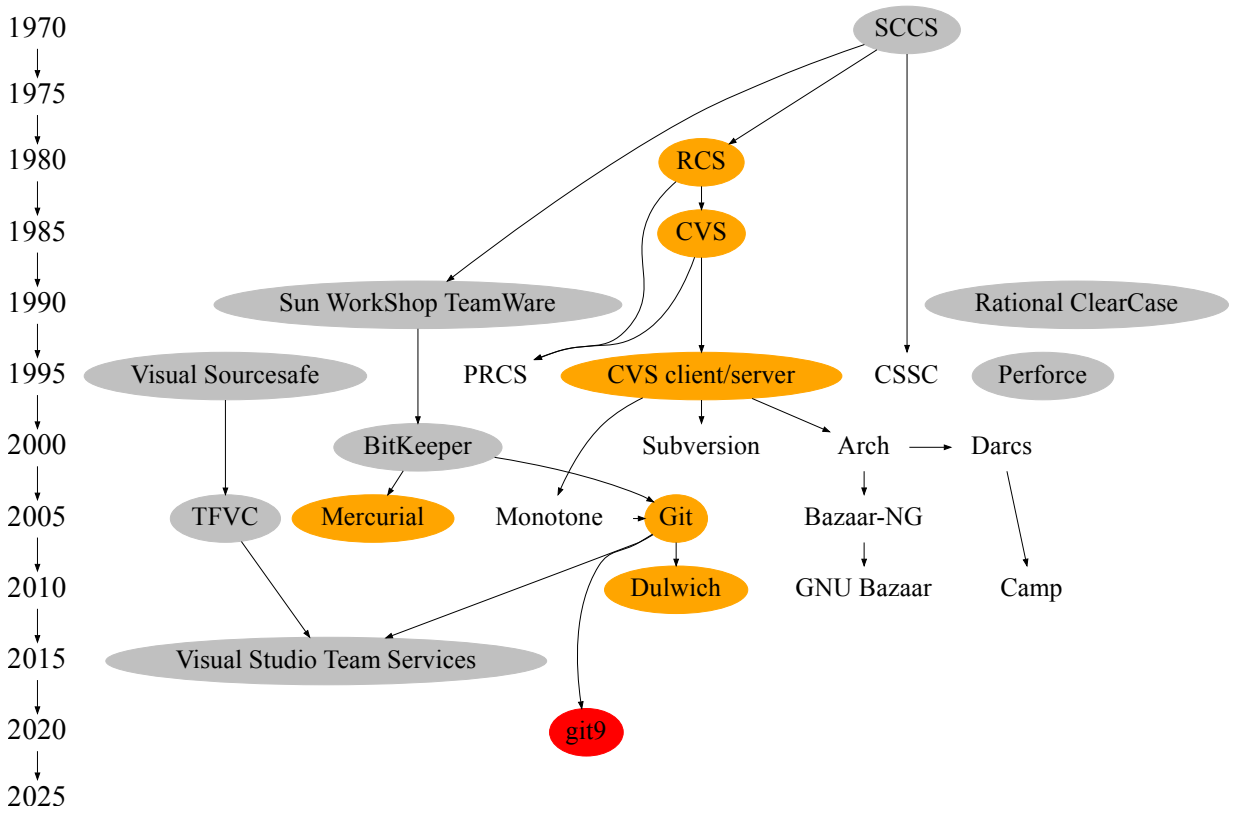


Figure 1.1: Version control systems timeline

- Git [CS14][10] was originally created by Linus Torvalds in 2005 to manage the source code of the Linux kernel. It followed a series of *distributed VCSs* (e.g., TeamWare, BitKeeper, Arch, Darcs, Monotone) designed to overcome the main limitations of CVS (a centralized VCS). Because a distributed VCS (DVCS) does not require access to a central repository, it allows developers to work independently offline. A DVCS makes it also easy and cheap to create *branches* representing parallel developments, and to reconcile later those branches (see Section 2.1.5 for more discussions on the advantages of distributed VCSs over centralized VCSs).

  As I explained in Section 1.2, Git is a large program with more than 200 000 LOC spread over more than 400 files (not including the tests, GUIs, and extra contributions). The first version of Git was small and contained only 1000 LOC[11]. However, it contained only a few low-level commands that were not easy to use. This forced programmers to develop and use the extra program `cogito`[12], which provided an extra layer of higher-level commands (called *porcelain*), but added many more lines of code.

- Mercurial [O'S09][13] is another popular DVCS started also in 2005 by another Linux kernel programmer: Matt Mackal. It is mostly written in Python and relies only on a few C files for critical operations. Its code is arguably easier to understand than Git but it still contains more than 100 000 LOC spread over more than 170 files (not including the tests, extensions, and extra contributions).

Figure 1.1 presents a timeline of major VCSs. `git9` is not as efficient and complete as Git or Mercurial, but it provides almost all of the essential features of those programs with more than an order of magnitude less code.

[10] https://git-scm.com
[11] https://github.com/git/git/commit/e83c5163316f89bfbde7d9ab23ca2e25604af290
[12] http://git.or.cz/cogito/
[13] https://www.mercurial-scm.org/

1.4 Getting started

To play with `git9`, you will first need to install the Plan 9 fork used in Principia Softwarica (see https://www.principia-softwarica.org/getting-started.html). Once installed, you can test `git9` under Plan 9 with the following commands:

```
1 $ cd /tests/
2 $ mkdir hello
3 $ cd hello/
4 $ git/init
5 ...
6 $ echo "Hello Git" > hello.txt
7 $ git/add hello.txt
8 $ git/commit -m "first commit" .
9 heads/front: 56adf1865
```

The command in Line 4 initializes a new *repository* by creating the appropriate metadata in the `/tests/hello/.gi` subdirectory. Line 7 then adds the new file `hello.txt` to the *staging area* of Git (I will explain later in Section 2.3.3 what is a staging area). Finally, Line 8 *commits* what was staged to the repository and records this commit with the message "first commit".

1.5 Requirements

Because most of this book is made of C source code, you will need a good knowledge of the C programming language [KR88] to understand it. A significant part of `git9` is made of `rc` shell scripts so you will need to read the SHELL book [Pad18]. Moreover, those scripts are often using utilities such as `awk` or `sed` so reading the UTILITIES book [Pad25] would be useful.

You do not need to know Git, or more generally any VCSs, to understand this book. However, if while reading this book you have specific questions on the interface of Git, I suggest you to consult the man pages of Git[14], or any of the books on Git (e.g., [CS14]).

1.6 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

Most of this document is made of source code from Ori Bernstein, so those parts are copyright by him. The prose is mine and is licensed under the All Rights Reserved License.

1.8 Acknowledgments

I would like to acknowledge first the author of `git9`, Ori Bernstein, who wrote in some sense most of this book.

I would like also to thank Linus Torvalds, the original author of Git, for designing this simple but powerful VCS. Many of his design decisions seem obvious in retrospect but they are not: most VCSs chose different designs

[14] https://git-scm.com/docs

(for the storage, file permissions, for the way to handle renames, etc.) which in the end are more complicated and usually less efficient than in Git.

Chapter 2

Overview

Before showing the source code of `git9` in the following chapters, I first give an overview in this chapter of the general principles of a VCS. I also quickly describe the command-line interface of `git` (and `git9`), the format of a Git repository, and I use a simple terminal session containing `git` commands to illustrate its major features. Finally, I define terms, explain how the code is organized, and more generally give the background necessary to understand the code I will show later.

2.1 Version control system principles

The main requirements for a modern VCS are to be able to store past versions of a set of files, to track the changes to those files, and to help multiple developers to collaborate with each other on those files by supporting concurrent and parallel development (I will explain soon the differences between those two kinds of development). The next sections will each give more details about those requirements. I will also try to present the major techniques used by popular VCSs to satisfy those requirements. Finally, I will discuss two different architectures for VCSs: centralized and distributed.

2.1.1 Storing past versions

The first requirement of a VCS is to store past versions of a file, hence the 'V' in VCS. However, there is also an 'S' in VCS, which means it must be done in a systematic way. Indeed, one way to understand the usefulness of a VCS is to see how people are struggling when they are managing documents that evolve (e.g., source code, LaTeXfiles, notes) without a VCS. In that situation, people often uses `cp` to *backup* files, but it is inefficient space-wise and error-prone.

The repository

Working copy

Delta

Compression

2.1.2 Tracking changes

The commit

Version identifier

Change granularity

`diff`

`patch`

2.1.3 Concurrent development

Locks

Merging

Merge conflicts

2.1.4 Parallel development

Branch

Merging

A direct acyclic graph of versions

2.1.5 Centralized versus distributed

2.2 `git` command-line interface

I just described the general principles of a VCS, and illustrated some of those principles with examples from RCS, CVS, or Git. I will now focus exclusively on Git and give more details about the interface of its command-line program: `git`.

The command-line interface of `git` is pretty simple:

```
1 $ git
2 usage: git <init|add|rm|commit|branch|checkout|reset|...> [options]
3 $ git init
4 Initialized empty Git repository in /tests/hello/.git/
5 $ git add hello.txt
6 $ git commit -m "first commit" --author "pad <todo@todo>"
```

`git` takes first a *command* as its first argument and then options and extra arguments specific to this command. For example, the command `init` at Line 3 does not need any extra arguments, but `add` at Line 5 requires at least the name of a file or directory, and `commit` at Line 6 usually requires command-line flags. I will gradually describe the different `git` commands, their options, and implementations in the following chapters.

Note that in most of this chapter I will use `git` and not `git9` commands to be more general. The reader can thus easily test those commands on Linux or macOS by using the regular `git` program. It is easy to adapt

those commands to `git9` on Plan 9. The main difference is that `git9` accepts commands separated by a `/` (as in `git/init` or `git/add foo.txt`) instead of space.

2.3 Git concepts and data structures

To understand how Git is implemented, and to some degree to learn how to use Git effectively, you need to know the few concepts underlying Git. Those concepts are the *object store*, the *reference*, and the *staging area*. They correspond also to `git9`'s main data structures, which I will describe fully in Chapter 3. The following sections will just give an overview of those data structures.

2.3.1 Object store

To really understand Git, you need to realize that at its core, Git is a simple *content-addressable storage system*. Given the *hash* of a content (Git uses SHA1 hashes, as explained below), Git can retrieve back quickly the full corresponding content. Git internally is simply a *persistent hash table*, also known as a *key/value store*.

In this store, Git manipulates mainly three kinds of values, known as *objects* in Git terminology (hence the term *object store*):

- A *blob*: to represent the content of a file at a specific time.
- A *tree*: to assign names to blobs or other subtrees
- A *commit*: to associate to a specific toplevel tree a message, an author, a date, and zero or more *parent* commits

You will see another kind of objects in Section 18.1 with the *tag*. In addition to those objects, Git maintains also *references* to specific commits and an *index* of specific blobs, as explained in the next sections.

Regarding the keys of the store, Git uses the SHA1 algorithm [sha93, EJ01] to compute the hash of an object. This algorithm associates an almost unique number of 160 bits (which amounts to 20 bytes, or 40 digits in hexadecimal format) to any content of arbitrary length. Section 16.1 presents the code of this algorithm and gives more information on SHA1. For example, given the content `Hello Git\n`, the SHA1 algorithm will return the number `9f4d96d5b00d98959ea9960f069585ce42b1349a` in hexadecimal format. Here the hashing is not very interesting because the hash is bigger than the original content, but in practice most files under a VCS are far bigger than 20 bytes.

SHA1 is a complex cryptographic hash function. It is outside the scope of this book to explain how and why it works, but what is important for Git is that SHA1 is an almost perfect hash function: given two different content, there is an almost zero probability that SHA1 will return the same hash number. Such an event is called a *collision*, and in practice it should never happen. Thanks to this almost perfect hash function, Git can identify any content of any length with just a 20 bytes number. In Git, the SHA1 of a file is similar to the *inode* of a file in a filesystem (see the KERNEL book [Pad14] for more information on filesystems and inodes)[1], except it references a specific version of a file. In the tree object, which associates names to blobs, Git uses the SHA1 of a blob to identify this blob. In the same way, in the commit object, Git uses the SHA1 of a tree to identify the toplevel tree a commit refers to.

Figure 2.1 presents the Git objects (and their relationships) of the Git repository created in Section 1.4 after the addition of another commit. The first commit, which added `hello.txt`, is at the top left of Figure 2.1. The second commit, below, added the two files `foo.txt` and `bar.txt` under a directory `dir1/`, and also modified `hello.txt`. I will describe soon in Section 2.4 how Git stores those objects on the disk.

Commit objects are linked together, as shown in Figure 2.1 with the second commit linked to the first commit through the `parents` field of the object. Those links represent the history of the repository, also known as its

[1]The author of Git, Linus Torvalds, is also the author of Linux, which explains why he chose a design inspired by filesystems.



Figure 2.1: Git objects relationships for `hello/.git` after two commits.

log. A commit can have multiple parents in the case of a *merge* (I will explain in Section 11.1 how to merge branches in Git). Thus, the log forms a *graph*, or more specifically a *direct acyclic graph* (DAG). Figure 2.2 shows such a graph for an additional series of commits following the first two commits of Figure 2.1. In this example, a developer *forked* first an experimental branch called `branch1` (I will explain briefly how to create a branch in Section 2.5.4 and fully in Section 10.2) following the second commit of Figure 2.1. At the same time, development on the main branch, called the *master branch* in Git terminology, continued and saw two commits: `commit4` and `commit6`. The experimental branch got *merged* later at `commit7` (I will explain in Section 11.1 how to merge branches in Git). This is why this commit has two parents. Later on, a developer created another branch called `experiment` that remained active and did not get merged yet; it follows in parallel the development of the master branch.

2.3.2 References

As I mentionned briefly in the previous section, Git maintains, in addition to the object store, *references* to specific commit objects. Those references have a *name* and a *content*. For example, in Figure 2.2, the reference named `refs/heads/experiment` points to the last commit of the `experiment` branch, and its content is the SHA1 of the `commit9` object. I will explain in Section 2.4 how Git stores those references on the disk.

Git maintains also a special reference called the *HEAD* which points to the last commit of the *current branch* (see the bottom of Figure 2.2). Its content is usually not a SHA1 but the name of another reference. By default, `HEAD` points to the master branch, as shown at the bottom of Figure 2.2, and so its content is the string `ref: refs/heads/master` (and `refs/heads/master` contains the SHA1 of the last commit of the master branch). When you switch branch (I will explain how to switch branch briefly in Section 2.5.4 and fully in Section 10.4), the content of the `HEAD` will change.

A reference pointing to the last commit of a branch is called a *head* in Git terminology and starts with `refs/heads/`. In Figure 2.2, those heads are `refs/heads/experiment` and `refs/heads/master`.

Git maintains a final set of special references called the *remotes*, for example `refs/remotes/origin/master` pointing to `commit8` in Figure 2.2. Those references are used when people are collaborating with each other and I will explain them in Chapter 14.

2.3.3 Staging area

The last important concept of Git is the *staging area*. Git operates mainly on three different areas, as shown at the top of Figure 2.3:

1. The *working copy* contains the current state of the files
2. The *repository* contains all past versions of all the files including the state of the files at the last commit
3. The *staging area* contains what the user wants to commit next

After you modify a set of files in your working copy, for example, `bar.txt`, `foo.txt`, and `misc.txt` at the right of Figure 2.3, you need to indicate which of those files and modifications should be part of the next commit. To do so, you need to *mark* those files by using the command `git add` (or `git rm` if you want to mark for deletion a file).

Once you marked all the relevant files, you can use the command `git commit` to commit the modifications to the repository. For example, in Figure 2.3, only the modifications to `foo.txt` and the deletion of `misc.txt` will be part of the next commit, and not the modifications to `bar.txt`.

Once you add a file in a repository, the file is said to be *tracked* in Git terminology. You can sometimes avoid manipulating explicitely the staging area by using instead the command `git commit -a` to automatically commit all the modifications to the tracked files. However, the staging area and `git add` give more flexibility

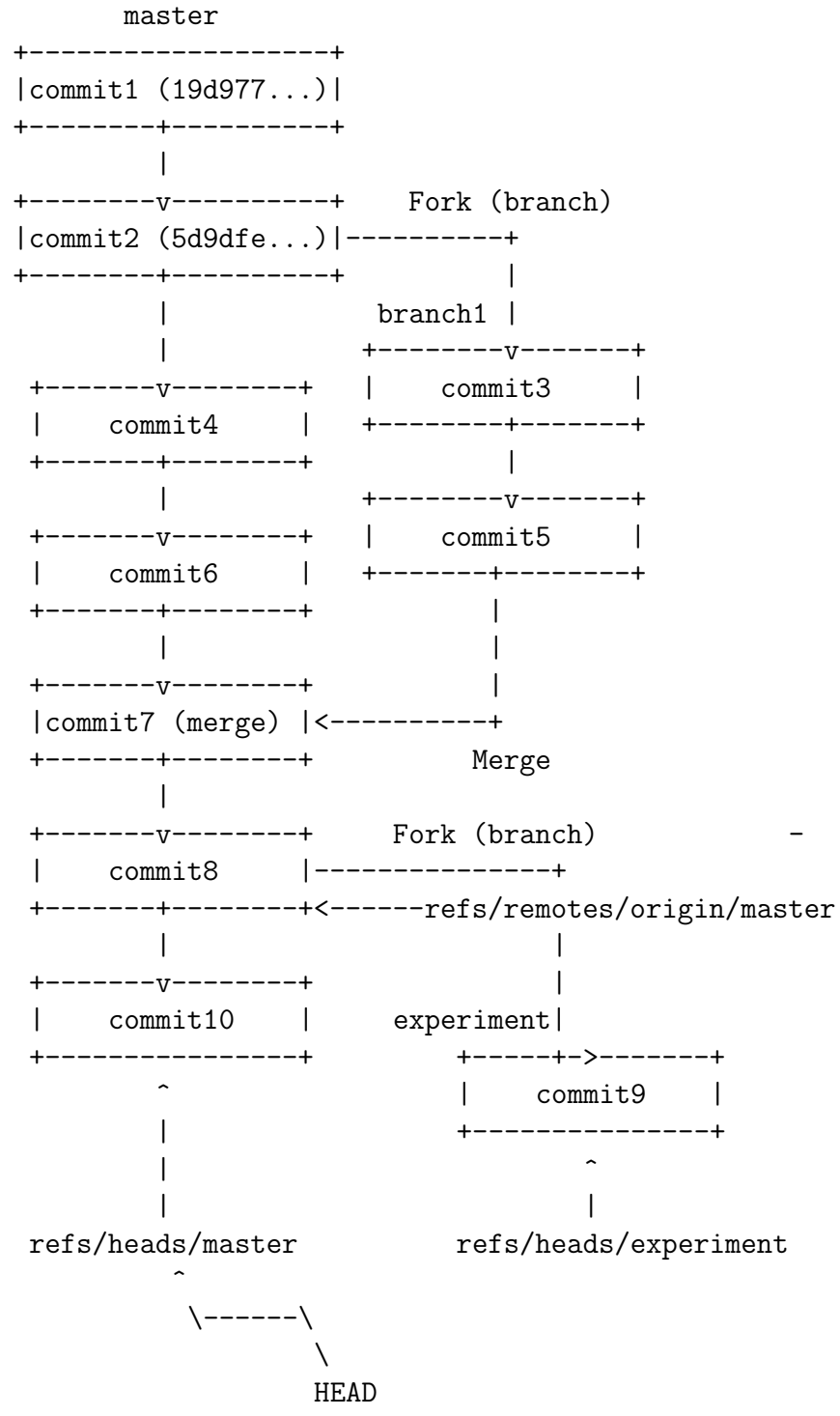


Figure 2.2: Commit graph and references.

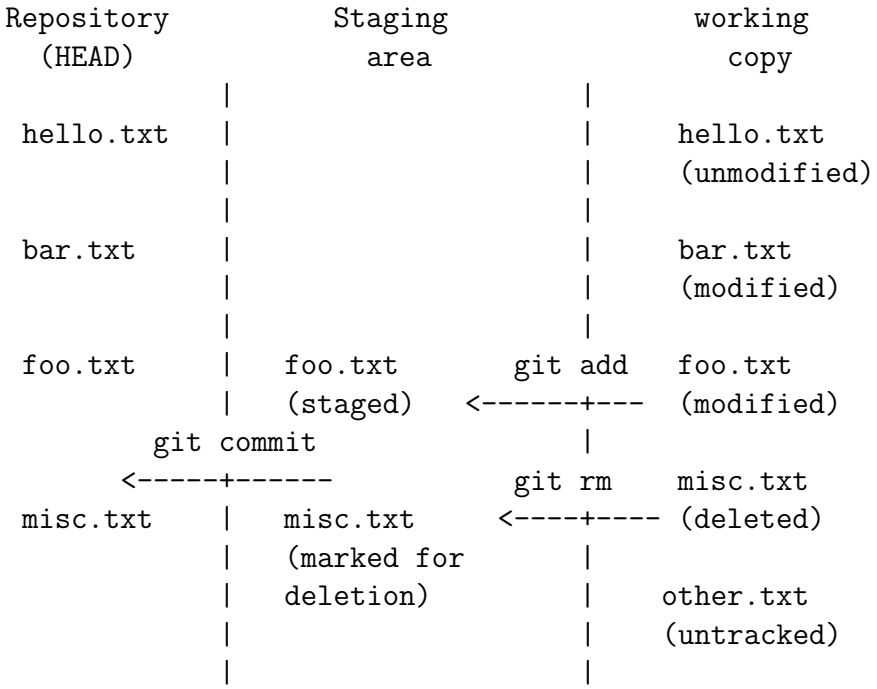


Figure 2.3: Git areas.

to the user by allowing to split a set of modifications in multiple commits. For example, in Figure 2.3, the modification to `bar.txt` can be put in another commit.

Internally, Git uses a data structure called the *index* (also known as the *directory cache*) to manage the staging area. I will explain later in Section 3.5 why this data structure is called an index.

2.4 Repository format: `.git/`

Now that you know the main concepts and data structures underlying Git, it will be easy to understand the format of the `.git/` directory. Here is the sligtly edited output of the UNIX command `tree`, which recursively explores a directory and displays its content as a tree [2]. The command is applied here to the `/tests/hello/.git/` directory after the last command in Section 1.4, when the repository contained just one commit:

```
1 $ pwd
/tests/hello
2 $ tree -F .git/
.git/
|-- HEAD
...
|-- index
|-- objects/
|   |-- 2f/
|   |   `-- 092e9cadfc1eb4a6d2febfddb941f4c1fe6fd6
|   |-- 19/
|   |   `-- d977a48d1d7b7ae8d520dd66190702dd4fb5bc
|   |-- 9f/
```

[2] See http://mama.indstate.edu/users/ice/tree/ for more information about the `tree` program. The `-F` option used in the terminal session above is to add the special mark `'/'` at the end of directory names, to help differentiate regular files from subdirectories.

```
|   |   `-- 4d96d5b00d98959ea9960f069585ce42b1349a
......
`-- refs/
    |-- heads/
    |   `-- master
.....
```

The objects of the key/value store are simply stored in separate files under `.git/objects/`. The hexadecimal digits of the SHA1 of the object are split in two parts: the first two digits are used for the directory name containing the object and the remaining digits for the filename of the object. For example, the file `.git/objects/9f/4d96d5b00d98959ea9960f069585ce42b1349a` contains the blob of `Hello Git\n` (see the blob and its SHA1 at the top right of Figure 2.1).

Git uses the first two digits of the SHA1 to classify objects in separate folders to avoid having too many files in the same directory. Indeed, this would slow down filesystem operations such as opening a file referenced by a specific SHA1 key, a recurring operation under Git (see the KERNEL book [Pad14] for more information on the performance of filesystem operations).

The references are also stored simply in separate files. Git uses the name of the reference as the path to a file under `.git/refs/` (e.g., `.git/refs/heads/master` in the example above).

Finally, the index and the HEAD are stored directly under `.git/` in separate files.

2.5 hello/.git/

In this section, I will give a short tutorial on how to use Git. I will explain the main commands of `git` by using a terminal session starting from an empty `hello/` directory. I will continue the series of commands I introduced in Section 1.4. I will also describe the semantics of those commands by showing their effects on the Git metadata stored under the `hello/.git/` subdirectory (by using mainly the output of the `tree` command, as in the previous section). This will hopefully help you to understand the code of those commands I will present later in the following chapters.

I will again use `git` instead of `git9` in the terminal sessions but the examples are easy to transpose to Plan 9.

2.5.1 Creating a repository

Here are the commands to create a fresh new repository:

```
1 $ cd /tests/
2 $ mkdir hello
3 $ cd hello/
4 $ git init
Initialized empty Git repository in /tests/hello/.git/
5 $ tree -F .git/
.git/
|-- HEAD
...
|-- objects/
`-- refs/
    |-- heads/
6 $ cat .git/HEAD
ref: refs/heads/master
```

The `init` command creates just the directory structure under `.git/`, without any objects or references in it (the `.git/objects/` and `.git/refs/heads` are empty directories above). There is also not yet an index file. There is a `HEAD` file, but its contents, shown with command 6 above references an head that does not exist yet under `.git/refs/heads/`.

2.5.2 Staging a diff

Here is the command to add a new file (or a new version of a file) to the repository:

```
1 $ echo "Hello Git" > hello.txt
2 $ git add hello.txt
3 $ tree -F .git/
.git/
|-- HEAD
...
|-- index
...
|-- objects/
|   |-- 9f/
|   |   `-- 4d96d5b00d98959ea9960f069585ce42b1349a
...
`-- refs/
    |-- heads/
4 $ cat .git/objects/9f/4d96d5b00d98959ea9960f069585ce42b1349a
x^K\312\311OR04`\360H\315\311\311Wp\317,\341^B^@4\201^Ec
```

Adding the first file in a repository has two effects:

1. The creation of a new blob object (here in `.git/objects/9f/4d96...`) containing essentially the compressed form of the content of the added file (hence the cryptic output of the `cat` command above). I will fully explain in Chapter 4 the format of a blob on the disk.

2. The creation of the index file. Again, I will describe precisely the format of the index file in Chapter 4, but the index essentially associates to every tracked filenames by the repository the SHA1 of the blob *currently* corresponding to the filename. In the example above, the index will associate to the filename `hello.txt` the SHA1 `9f4d96...`.

You must use the command `add` to add a new file to the repository, when the file was not yet tracked by the repository. However, you must also use the command `add` to stage the modifications on a tracked file as in Figure 2.3. In that case, the `add` command also creates a new blob with the current content of the file. It also updates the index so the filename of the added file now points to the SHA1 of the new blob.

Note that using `git add` on an unmodified file will have no effect. Indeed, Git will compute the SHA1 of the current content, which will be identical to the SHA1 of an existing blob. Updating the index will have also no effect because the filename will reference the same SHA1.

2.5.3 Committing a diff

Here is the command to commit to the repository what was previously staged:

```
1 $ git commit -m "first commit" --author "pad <todo@todo>"
2 $ tree -F .git/
.git/
|-- HEAD
|-- index
|-- objects/
|   |-- 2f/
|   |   `-- 092e9cadfc1eb4a6d2febfddb941f4c1fe6fd6
|   |-- 19/
|   |   `-- d977a48d1d7b7ae8d520dd66190702dd4fb5bc
|   |-- 9f/
|   |   `-- 4d96d5b00d98959ea9960f069585ce42b1349a
`-- refs/
    |-- heads/
    |   `-- master
3 $ cat .git/refs/heads/master
19d977a48d1d7b7ae8d520dd66190702dd4fb5bc
```

The `commit` command has three effects:

1. The creation of a new tree object, in the example above `.git/objects/2f/092e...` (see Figure 2.1 for the content of this tree). This tree derives from the content of the index. Because the repository contains only one directory (the toplevel root containing just `hello.txt`), `git commit` will create just one tree object, but in general when a project has many directories `git commit` may create many new tree objects.

2. The creation of a new commit object, here `.git/objects/19/d977...`, referencing the newly created tree object (again, see Figure 2.1 for the content of this commit and its relationship to the tree object).

3. The update of the content of the head of the current branch (`refs/heads/master` according to the content of `.git/HEAD`) to contain the SHA1 of the newly created commit object (here `19d977...`), as shown by command 3 above.

Note that it can be tedious to specify each time on the command-line your name and email through the `--author` flag of `git commit`. Git can also leverage a configuration file containing such information in a `.gitconfig` file in your home directory as explained in Section 17.3.

2.5.4 Managing branches

Here is the command to create a new branch:

```
1 $ git branch experiment
2 $ tree -F .git/
.git/
|-- HEAD
...
`-- refs/
    |-- heads/
    |   |-- experiment
    |   `-- master
...
3 $ cat .git/HEAD
```

```
ref: refs/heads/master
4 $ cat .git/refs/heads/master
19d977a48d1d7b7ae8d520dd66190702dd4fb5bc
5 $ cat .git/refs/heads/experiment
19d977a48d1d7b7ae8d520dd66190702dd4fb5bc
```

As you can see, creating a new branch is an extremely cheap operation under Git. It consists just in adding a file under `.git/refs/heads/` with the name of the new branch (here `.git/refs/heads/experiment`), and the SHA1 commit of the current branch for its content (here `19d977...`). However, creating a branch does not switch to this branch as shown by command 3 above.

Here is the command to switch to the new branch:

```
6 $ git checkout experiment
Switched to branch 'experiment'
7 $ cat .git/HEAD
ref: refs/heads/experiment
```

Switching to a branch is usually a more costly operation. `git checkout` modifies `HEAD` (as shown by command 7 above), which is fast, but it also recomputes the index from the tree object that is referenced in the commit of the head of the new branch, as well as from all its subtrees (see Section 10.4.1 for the full description of this operation). It also updates all the files so that they contain the content of the blobs referenced in the index. The more the new branch differs from the current branch, the more costly switching to this new branch will be.

With `git9` you will need to use `git/branch -n experiment` to create a new branch and it will also automatically switch to this branch. There is no `git/checkout`; everything is done with `git/branch`.

2.5.5 Inspecting objects

Git provides a few commands to query the repository. Those commands are useful to understand the internal structures of Git. They are also useful for doing some archeology on the history of a project, as explained in Section 12.7. The first of those commands, `show`, shows the content of a Git object. `git show` takes the SHA1 of an object as a parameter (in hexadecimal format) and pretty prints its content in a readable format. Here are a few examples of this command:

```
1 $ tree -F .git/
...
|-- objects/
|   |-- 2f/
|   |   `-- 092e9cadfc1eb4a6d2febfddb941f4c1fe6fd6
|   |-- 19/
|   |   `-- d977a48d1d7b7ae8d520dd66190702dd4fb5bc
|   |-- 9f/
|   |   `-- 4d96d5b00d98959ea9960f069585ce42b1349a
...
2 $ git show 9f4d96d5b00d98959ea9960f069585ce42b1349a
Hello Git
3 $ git show 2f092e9cadfc1eb4a6d2febfddb941f4c1fe6fd6
tree 2f092e9cadfc1eb4a6d2febfddb941f4c1fe6fd6

hello.txt
```

```
4 $ git show 19d977a48d1d7b7ae8d520dd66190702dd4fb5bc
commit 19d977a48d1d7b7ae8d520dd66190702dd4fb5bc
Author: pad  <todo@todo>
Date:   Fri Sep 29 14:04:46 2017 -0700

    first commit

diff --git a/hello.txt b/hello.txt
new file mode 100644
index 0000000..9f4d96d
--- /dev/null
+++ b/hello.txt
@@ -0,0 +1 @@
+Hello Git
```

The command `show` mainly opens the appropriate file under `.git/objects/`, decompresses the file, and finally displays its content.

Another important query command is `log` to see the full history of a repository. Here is the output of this command on the repository we used this far:

```
1 $ git log
commit 19d977a48d1d7b7ae8d520dd66190702dd4fb5bc
Author: pad  <todo@todo>
Date:   Fri Sep 29 14:04:46 2017 -0700

    first commit
```

The command `log` mainly opens `.git/HEAD` to get a reference to the last commit. Then, it goes through the parent links of the commit object to recursively explore all the linked commit objects (stored again under `.git/objects/`).

2.5.6 Other commands

There are a few other important Git commands: `git merge`, `git status`, `git diff`, `git clone`, `git pull`, or `git push`, but I will introduce them later in this document.

2.6 Code organization

2.7 Software architecture

2.8 Book structure

Chapter 3

Core Data Structures

3.1 Secure Hash Algorithm (SHA1) hashes

3.1.1 Binary hashes

3.1.2 Hexadecimal hashes

3.1.3 Conversion functions

3.2 `Repository.t`

3.3 `Objects.t`

3.3.1 `Blob.t`

3.3.2 `Tree.t`

3.3.3 `Commit.t`

3.4 References

3.4.1 Reference names

3.4.2 `HEAD`

3.4.3 Reference content

3.4.4 Reference store

3.4.5 Objectish references

3.5 `Index.t`

Chapter 4

Reading from a Repository

4.1 Objects

4.1.1 Decompression

4.1.2 Reading a blob

4.1.3 Reading a tree

4.1.4 Reading a commit

4.2 References

4.3 Index

Chapter 5

Writing to a Repository

5.1 Objects

5.1.1 Compression

5.1.2 Locking

5.1.3 Writing a blob

5.1.4 Writing a tree

5.1.5 Writing a commit

5.2 References

5.3 Index

Chapter 6

Main Functions

6.1 Main commands

6.2 `Main.main()`

6.3 Getting help: `git help`

Chapter 7

Creating a Repository

7.1 Initializing a new repository: `git init`

⟨*git9/init.rc* 30a⟩≡
```
#!/bin/rc -e
rfork ne
. /lib/git/common.rc

flagfmt='u:upstream upstream,b:branch branch'; args='name'
eval `{auxrc/getflags $*} || exec auxrc/usage

dir=$1
if(~ $#dir 0)
        dir=.
if(~ $#branch 0)
        branch=master
if(test -e $dir/.git)
        die $dir/.git already exists
⟨init.rc set $name derived from $dir 30b⟩
⟨init.rc check $upstream flag 48a⟩

mkdir -p $dir/.git/refs/^(heads remotes)
mkdir -p $dir/.git/^(fs objects)
>$dir/.git/config {
        echo '[core]'
        echo '	repositoryformatversion = p9.0'
        ⟨init.rc set origin in .git/config 48b⟩
        echo '[branch "'$branch'"]'
        echo '	remote = origin'
}
>$dir/.git/INDEX9
>$dir/.git/HEAD {
        echo ref: refs/heads/$branch
}
exit ''
```

⟨init.rc *set* $name *derived from* $dir 30b⟩≡ (30a)
```
name=`{basename `{cleanname -d `{pwd} $dir}}
```

7.2 Copying an existing repository: `cp -r`

7.3 Cloning an existing repository: `git clone`

7.4 Opening a repository

Chapter 8

Staging a Diff

8.1 Adding files: `git add`

⟨*git9/add.rc* 32a⟩≡
```
#!/bin/rc -e
rfork ne
. /lib/git/common.rc

gitup

flagfmt='r:remove'; args='file ...'
eval `{auxrc/getflags $*} || exec auxrc/usage

s=A

⟨add.rc if $remove flag 33d⟩
⟨add.rc sanity check $* 32b⟩

⟨add.rc set $paths derived from $* 33a⟩
walk -f ./$paths | grep -v '^(./)?.git/' | \
        sed 's/^/'$s' NOQID 0 /' >> .git/INDEX9
exit ''
```

⟨add.rc *sanity check* `$*` 32b⟩≡ (32a)
```
if(~ $#* 0)
        exec auxrc/usage
```

⟨*fn* `gitup` *(common.rc)* 32c⟩≡ (69)
```
fn gitup{
        gitroot=`$nl{git/conf -r >[2]/dev/null}
        if(~ $#gitroot 0)
                die 'not a git repository'
        gitrel=`$nl{pwd | drop $gitroot | sed 's@^/@@'}
        if(~ $#gitrel 0)
                gitrel='.'
        if(! cd $gitroot)
                die cd $gitroot: no repo there

        ⟨gitup() (common.rc) start git/fs 33b⟩
}
```

⟨*global* `nl` *(common.rc)* 32d⟩≡ (69)
```
nl='
'
```

⟨add.rc *set* $paths *derived from* $* 33a⟩≡ (32a)
```
paths='$nl{cleanname -d $gitrel $* | drop $gitroot}
```

⟨gitup() *(common.rc) start git/fs* 33b⟩≡ (32c)
```
gitfs=$gitroot/.git/fs
startfs=()
mkdir -p $gitfs
if(! test -e $gitfs/ctl)
        startfs=true
if(! grep -s '^repo '$gitroot'$' $gitfs/ctl >[2]/dev/null)
        startfs=true
if(~ $#startfs 1)
        git/fs >[1=2]
if not
        status=''
```

8.2 Removing files: `git rm`

⟨*git9/rm.rc* 33c⟩≡
```
#!/bin/rc -e

exec git/add -r $*
```

⟨add.rc *if* $remove *flag* 33d⟩≡ (32a)
```
if(~ $remove 1)
        s=R
```

8.3 Renaming files: `git mv`

Chapter 9

Committing a Diff

9.1 Committing the index: `git commit`

⟨*git9/commit.rc* 34⟩≡

```
#!/bin/rc -e
rfork ne
. /lib/git/common.rc

⟨fn findbranch(commit.rc) 35a⟩
⟨fn cleanmsg(commit.rc) 35b⟩
⟨fn editmsg(commit.rc) 35c⟩
⟨fn parents(commit.rc) 36a⟩
⟨fn commit(commit.rc) 36b⟩
⟨fn update(commit.rc) 36c⟩
⟨fn filterdiff(commit.rc) 36d⟩
⟨fn pickhunks(commit.rc) 37a⟩
⟨fn apply(commit.rc) 37b⟩

gitup

flagfmt='m:msg message, r:revise, e:edit, p:partial'; args='[file ...]'
eval `{auxrc/getflags $*} || exec auxrc/usage

wdir=/mnt/git
ramfs -m $wdir
msgfile=$wdir/git-msg.$pid
if(~ $#msg 1)
        echo $msg >$msgfile.tmp
if not if(~ $#revise 1){
        msg=1
        echo revising commit `{cat $gitfs/HEAD/hash}
        cat $gitfs/HEAD/msg >$msgfile.tmp
}

files=()
if(test -f .git/merge-parents)
        files=`$nl{git/query -c `{cat .git/merge-parents} | sed 's/^..//'}
if(! ~ $#* 0)
        files=($files `$nl{git/walk -fRMA -c `$nl{cleanname -d $gitrel $*}})
if(~ $status '' || ~ $#files 0 && ! test -f .git/merge-parents && ~ $#revise 0)
        die 'nothing to commit'
if(~ $#partial 1){
        for(i in $files)
                pickhunks $i
        apply
```

```
}

 @{
        flag e +
        whoami
        findbranch
        parents
        editmsg
        commit
        update
} || die 'could not commit:' $status
exit ''
```

⟨*fn* `findbranch`*(commit.rc)* 35a⟩≡ (34)

```
fn findbranch{
        branch=`{git/branch}
        if(test -e $gitfs/branch/$branch/tree){
                refpath=.git/refs/$branch
                initial=false
        }
        if not if(test -e $gitfs/object/$branch/tree){
                refpath=.git/HEAD
                initial=false
        }
        if not if(! test -e $gitfs/HEAD/tree){
                refpath=.git/refs/$branch
                initial=true
        }
        if not
                die 'invalid branch:' $branch
}
```

⟨*fn* `cleanmsg`*(commit.rc)* 35b⟩≡ (34)

```
# Remove commentary lines.
# Remove leading and trailing empty lines.
# Combine consecutive empty lines between paragraphs.
# Remove trailing spaces from lines.
# Ensure there's trailing newline.
fn cleanmsg{
        awk '
        /^[     ]*#/ {next}
        /^[     ]*$/ {empty = 1; next}

        wet && empty {printf "\n"}
        {wet = 1; empty = 0}
        {sub(/[         ]+$/, ""); print $0}
        '
}
```

⟨*fn* `editmsg`*(commit.rc)* 35c⟩≡ (34)

```
fn editmsg{
        if(! test -s $msgfile.tmp){
                >$msgfile.tmp {
                        echo '# Author:' $name '<'$email'>'
                        echo '#'
                        for(p in $parents)
                                echo '# parent:' $p
                        git/walk -fAMR $files | subst '^' '# '
                        echo '#'
                        echo '# Commit message:'
```

```
                }
                edit=1
        }
        if(! ~ $#edit 0){
                giteditor=`{git/conf core.editor}
                if(~ $#editor 0)
                        editor=$giteditor
                if(~ $#editor 0)
                        editor=hold
                $editor $msgfile.tmp
        }
        cleanmsg < $msgfile.tmp > $msgfile
        if(! test -s $msgfile)
                die 'empty commit message'
}
```

⟨*fn* parents*(commit.rc)* 36a⟩≡ (34)

```
fn parents{
        if(! ~ $#revise 0)
                parents=`{cat $gitfs/HEAD/parent}
        if not if(test -f .git/merge-parents)
                parents=`{cat .git/merge-parents | sort | uniq}
        if not if(~ $initial true)
                parents=()
        if not
                parents=`{git/query $branch}
}
```

⟨*fn* commit*(commit.rc)* 36b⟩≡ (34)

```
fn commit{
        msg=``''{cat $msgfile}
        if(! ~ $#parents 0)
                pflags='-p'^$parents
        hash=`{git/save -n $"name -e $"email -m $"msg $pflags $files || die $status}
}
```

⟨*fn* update*(commit.rc)* 36c⟩≡ (34)

```
fn update{
        mkdir -p `{basename -d $refpath}
        # Paranoia: let's not mangle the repo.
        if(~ $#hash 0)
                die 'botched commit'
        rm -f .git/merge-parents
        echo $branch: $hash
        echo $hash > $refpath
        for(f in $files){
                if(! test -e $f && ! test -e .git/object/$hash/tree/$f)
                        echo R NOQID 0 $f >> .git/INDEX9
                if not
                        echo T NOQID 0 $f >> .git/INDEX9
        }
}
```

⟨*fn* filterdiff*(commit.rc)* 36d⟩≡ (34)

```
fn filterdiff{
        awk '
                function dump(){
                        if(n == 0)
                                return
```

```
                        print "--- a/"f >>"/dev/cons"
                        print "+++ b/"f >>"/dev/cons"
                        for(i = 0; i < n; i++)
                                print lines[i] >>"/dev/cons"
                        printf("Commit this hunk?[n]  ") >>"/dev/cons"
                        getline ans < "/dev/cons"
                        if(ans ~ /[yY]/)
                                for(i = 0; i < n; i++)
                                        print lines[i]
                        n = 0
                }
                /^---/{ next }
                /^\+\+\+/{
                        f = substr($0, 5)
                        print "--- "f
                        print "+++ "f
                        n = 0
                        next
                }
                /^@@/{ dump() }
                { lines[n++] = $0 }
                END{ dump() }
        '
}
```

⟨*fn* `pickhunks` *(commit.rc)* 37a⟩≡ (34)

```
fn pickhunks{
        p=$wdir/patch
        if(test -f .git/fs/HEAD/tree/$i)
                #<[12]/fd/0 >[13]/fd/1 { diff -u .git/fs/HEAD/tree/$1 $1 | filterdiff >>$p }
                { diff -u .git/fs/HEAD/tree/$1 $1 | filterdiff >>$p } || status=''
        if not
                diff -u /dev/null $1 >>$p || status=''
}
```

⟨*fn* `apply` *(commit.rc)* 37b⟩≡ (34)

```
fn apply{
        t=$wdir/tree/
        p=$wdir/patch
        ln=`{wc -l $p}
        if(test $ln(1) -lt 3)
                die 'no hunks seleted'
        f=`{patch -n $p}
        mkdir $t
        @{cd .git/fs/HEAD/tree; tar c $f >[2]/dev/null} | @{cd $t; tar xT}
        for(i in $f){
                d=`{basename -d $i}
                if(! ~ $d .){
                        mkdir -p $t^$d
                        bind -bc $t^$d $d
                }
        }
        bind -bc $t .
        patch <$p >/dev/null || die 'failed partial patch: '^$status
}
```

9.2 Computing the trees from an index

9.2.1 Grouping index files by directories

9.2.2 Building trees from directories

9.3 Storing the date, author, and committer

9.4 Recording the message

9.5 Updating `HEAD` atomically

9.5.1 Creating a new reference (first commit)

9.5.2 Updating an existing reference

Chapter 10

Branching

⟨*git9/branch.rc* 39⟩≡

```
#!/bin/rc -e
rfork en
. /lib/git/common.rc

gitup

flagfmt='a:listall, b:baseref ref, r:remove, n:newbr, s:stay, m:merge, M:nomod'
args='[branch]'
eval ''{auxrc/getflags $*} || exec auxrc/usage

modified=()
deleted=()
failed=()
if(~ $#* 0){
        if(~ $#listall 0)
                awk '$1=="branch"{print $2}' < $gitfs/ctl
        if not
                cd .git/refs/ && walk -f heads remotes | sort
        exit
}
if(! ~ $#* 1)
        exec auxrc/usage

branch=$1
if(~ $branch refs/heads/*)
        new=$name
if not if(~ $branch heads/*)
        new=refs/$branch
if not
        new=refs/heads/$branch

# where are we now?
orig=`{git/query HEAD}
origbranch=refs/`{awk '$1=="branch"{print $2}' < $gitfs/ctl}

# if we're switching branches to a branch that doesn't exist,
# but we have a remote branch that exists upstream, create a
# new head to mirror it.
if(~ $#newbr 0){
        if(! ~ $#baseref 0)
                die update would clobber $branch with $baseref
        if(! test -e .git/$new){
                baseref=`$nl{echo -n $new | sed s@refs/heads/@refs/remotes/origin/@}
```

```
                if(! base=‘{git/query $baseref})
                        exit ’bad ref’
        }
}

# figure out where we want the new branch to point to
if (~ $#baseref 1)
        base=‘{git/query $baseref} || exit ’bad ref’
if not if(~ $#newbr 0)
        base=‘{git/query $new} || exit ’bad ref’
if not
        base=‘{git/query HEAD} || exit ’bad ref’

modified=‘$nl{git/query -c HEAD $base | grep ’^[^-]’ | subst ’^..’}
deleted=‘$nl{git/query -c HEAD $base | grep ’^-’ | subst ’^..’}

# if we remove the current branch without switching, bad things happen
if(~ $remove 1 && ~ $origbranch $new)
        die ’cannot remove current branch’
# if we’re not merging, don’t clobber existing changes.
if(~ $#merge 0 && ~ $#remove 0){
        if(! ~ $#modified 0 || ! ~ $#deleted 0){
                git/walk -fRMA $modified $deleted ||
                        die ’uncommitted changes would be clobbered’
        }
}
if(~ $remove 1){
        rm -f .git/$new
        echo ’removed branch’ $new
        exit
}
commit=‘{git/query $base} || die ’branch does not exist:’ $base
basedir=‘{git/query -p $base}
dirtypaths=()
if(! ~ $#modified 0 || ! ~ $#deleted 0)
        dirtypaths=‘$nl{git/walk -cfRMA $modified $deleted}

>>[3].git/INDEX9{
        for(d in $deleted){
                if(! test -d $d){
                        rm -f $d
                        echo R NOQID 0 $d >>[1=3]
                }
        }
        for(m in $modified $deleted) if(! ~ $m $dirtypaths){
                d=‘$nl{basename -d $m}
                mkdir -p $d
                # Modifications can turn a file into
                # a directory, or vice versa, so we
                # need to delete and copy the files
                # over.
                a=dir
                b=dir
                if(test -f $m)
                        a=file
                if(test -f $basedir/tree/$m)
                        b=file
                if(! ~ $a $b){
                        rm -rf $m
                        echo R NOQID 0 $m >>[1=3]
```

```
                }
                if(~ $b file){
                        echo T NOQID 0 $m >>[1=3]
                        cp -x -- $basedir/tree/$m $m || {echo cp failed: $m; failed=1}
                        touch -c $m
                }
        }
}
modified=()
deleted=()

for(ours in $dirtypaths){
        common=$gitfs/object/$orig/tree/$ours
        theirs=$gitfs/object/$base/tree/$ours
        merge1 $ours $ours $common $theirs
        st=$status
        if(! ~ $st '')
                >[1=2] echo merge failed $ours: $st
}
dirtypaths=()

if(~ $new */*)
        mkdir -p .git/`{basename -d $new}
echo $commit > .git/$new
if(! ~ $#stay 0)
        exit
if(~ $failed 1){
        echo 'pull failed: fix errors and try again' >[1=2]
        exit nopull
}
echo ref: $new > .git/HEAD
echo $new: `{git/query $new}
exit ''
```

10.1 Listing branches: `git branch`

10.2 Creating a branch: `git branch <name>`

10.3 Deleting a branch: `git branch -d <name>`

10.4 Checking out a branch: `git checkout`

10.4.1 Computing the index (and work tree) from trees

10.4.2 Creating a file from a blob

10.5 Resetting a branch: `git reset`

Chapter 11

Merging

11.1 merging branches: `git merge`

⟨*git9/merge.rc* 42⟩≡

```
#!/bin/rc -e
rfork ne
. /lib/git/common.rc

fn merge{
        ourbr=$gitfs/object/$1/tree
        basebr=$gitfs/object/$2/tree
        theirbr=$gitfs/object/$3/tree

        all=‘$nl{{git/query -c $1 $2; git/query -c $2 $3} | sed ’s/^..//’ | sort | uniq}
        for(f in $all){
                ours=$ourbr/$f
                base=$basebr/$f
                theirs=$theirbr/$f
                merge1 ./$f $ours $base $theirs
        }
}

gitup

flagfmt=’’; args=’theirs’
eval ‘’’{auxrc/getflags $*} || exec auxrc/usage

if(! ~ $#* 1)
        exec auxrc/usage

theirs=‘{git/query $1}
ours=‘{git/query HEAD}
base=‘{git/query $theirs ^ ’ ’ ^ $ours ^ ’@’}

if(~ $base $theirs)
        die ’nothing to merge, doofus’
if(! git/walk -q)
        die ’dirty work tree, refusing to merge’
if(~ $base $ours){
        >[1=2] echo ’fast forwarding...’
        echo $theirs > .git/refs/‘{git/branch}
        git/revert .
        exit ’’
}
echo $ours >> .git/merge-parents
```

```
echo $theirs >> .git/merge-parents

merge $ours $base $theirs
echo 'merge complete: remember to commit'
exit ''
```

11.2 Merging two trees

11.3 Merging two files

11.4 `diff3` and `merge`

11.5 Merging conflicts

11.6 Committing merged branches: `git commit`

Chapter 12

Inspecting

12.1 Showing the content of an object: `git show`

12.1.1 Showing a blob

12.1.2 Showing a tree

12.1.3 Showing a commit

12.2 Representing changes

12.2.1 Tree changes

12.2.2 File changes

12.3 Showing file differences: `git diff`

⟨*git9/diff.rc* 44⟩≡

```
#!/bin/rc
rfork ne
. /lib/git/common.rc

gitup

flagfmt='c:commit branch, s:summarize, u:uncommitted'; args='[file ...]'
eval ''{auxrc/getflags $*} || exec auxrc/usage

bparam=(-b $commit)
if(~ $#commit 0){
        commit=HEAD
        cparam=()
        bparam=()
}

files=()
filt=MAR
if(~ $#uncommitted 1)
        filt=MARU
if(! ~ $#* 0)
        files='{cleanname -d $gitrel $*}

branch='{git/query -p $commit}
```

```
if(~ $summarize 1 || ~ $uncommitted 1){
        git/walk $bparam -r$gitrel -f$filt $cparam $files
        exit
}

showed=()
mntgen /mnt/scratch
bind $branch/tree/ /mnt/scratch/a
bind . /mnt/scratch/b
for(f in `$nl{git/walk $bparam -c -f$filt $cparam $files}){
        if(~ $#showed 0){
                echo diff `{git/query $commit} uncommitted
                showed=1
        }
        cd /mnt/scratch
        a=a/$f
        b=b/$f
        if(! test -f a/$f)
                a=/dev/null
        if(! test -f b/$f)
                b=/dev/null
        diff -u $a $b
}
exit ''
```

12.3.1 Computing tree changes

12.3.2 Computing file changes

12.3.3 Showing changes

12.4 Commit history walker

12.4.1 Basic walker

12.4.2 Extra features

max entries

Reverse walking

Exclude set

Time range

Path restrictions

Topological order

12.5 Showing the commit history: `git log`

12.5.1 Printing a commit

12.5.2 Diff summary: `git log --name-status`

12.5.3 Comparing trees

12.6 Showing the status of a file: `git status`

12.6.1 Listing staged modifications

12.6.2 Listing unstaged modifications

12.6.3 Listing untracked files

12.7 Archeology

12.7.1 Checking out an old version

12.7.2 Showing an old version of a file

12.7.3 Showing differences between past versions

12.7.4 Showing the history of a file or directory: `git log <path>`

Chapter 13

Packing

13.1 `Pack.t`

13.1.1 Pack data

13.1.2 Pack Index

Index pack v1

Index pack v2

13.2 IO

13.2.1 Reading a pack

13.2.2 Writing a pack

13.3 Modifications to previous functions and data structures

13.3.1 Object Store

13.3.2 Refs

13.4 Delta

13.5 Pack commands

13.5.1 `git pack-objects`

13.5.2 `git repack`

13.5.3 `git fetch-pack`

13.5.4 `git receive-pack`

Chapter 14

Exchanging

⟨init.rc *check* $upstream *flag* 48a⟩≡ (30a)
```
if(~ $#upstream 0){
        upstream=`{git/conf 'defaults "origin".baseurl'}
        if(! ~ $#upstream 0)
                upstream=$upstream/$name
}
```

⟨init.rc *set origin in* .git/config 48b⟩≡ (30a)
```
if(! ~ $#upstream 0){
        echo '[remote "origin"]'
        echo '  url = '$upstream
}
```

14.1 Client/server architecture

14.2 Pulling updates from another repository: `git pull`

⟨*git9/pull.rc* 48c⟩≡
```
#!/bin/rc -e
rfork en
. /lib/git/common.rc

fn update{
        upstream=$1
        url=$2
        dir=$3
        dflag=()
        if(! ~ $#debug 0)
                dflag='-d'
        heads=`$nl{
                # safe to split in awk, branch names may not
                # have spaces or newlines.
                git/query `$nl{
                        cd .git/refs && \
                        walk -f -emp heads remotes | \
                        sort -rn | \
                        awk '{print $2}'
                }
        }
        if(! ~ $#heads 0)
                heads='-h'^$heads
        {git/get $dflag -u $upstream $heads $url >[2=3] || die $status} | awk '
```

```
        /^remote refs\/(heads|tags)\//{
                ref=$2
                hash=$3
                gsub("^refs/heads", "refs/remotes/'$upstream'", ref)
                outfile = ".git/"ref
                print outfile > "/env/outfile"
                system("mkdir -p '$nl{basename -d $outfile}");\
                close("/env/outfile")
                print hash > outfile;
                close(outfile);
        }
        ' |[3] tr '\x0d' '\x0a'
}

gitup

flagfmt='d:debug, q:quiet, f:fetchonly,
        u:upstream upstream'
args=''
eval ''{auxrc/getflags $*} || exec auxrc/usage

if(~ $#upstream 0)
        upstream=origin
remote='$nl{git/conf 'remote "'$upstream'".url'}
if(~ $#remote 0){
        remote=$upstream
        upstream=THEM
}

update $upstream $remote
if (~ $fetchonly 1)
        exit

local='{git/branch}
remote='{git/branch | subst '^(refs/)?heads' 'remotes/'$upstream}

# we have local commits, but the remote hasn't changed.
# in this case, we want to keep the local commits untouched.
if(~ '{git/query HEAD $remote @} '{git/query $remote}){
        echo 'up to date' >[1=2]
        exit
}
# The remote repository and our HEAD have diverged: we
# need to merge.
if(! ~ '{git/query HEAD $remote @} '{git/query HEAD}){
        >[1=2]{
                echo ours:      '{git/query HEAD}
                echo theirs:    '{git/query $remote}
                echo common:    '{git/query HEAD $remote @}
                echo git/merge $remote
        }
        exit diverged
}

oldcommit='{git/query $local}
newcommit='{git/query $remote}

if(~ $#quiet 0){
        git/log -s -e $oldcommit'..'$newcommit
        echo $remote':' $oldcommit '=>' $newcommit
```

```
}

git/branch -mnb $remote $local
exit ''
```

14.2.1 Fetching objects locally

Find top common commits

Find missing objects

Iterating over objects

Adding Objects

14.2.2 Fetching references

Refspecs

Selecting the refs

Parsing ref tuples

14.2.3 Fetching objects in a pack

14.3 Object store walker

14.4 Pushing changes to another repository: `git push`

⟨*git9/push.rc* 50⟩≡

```
#!/bin/rc -e
rfork en
. /lib/git/common.rc

gitup

flagfmt='a:pushall, b:branch branch, f:force, d:debug,
         r:remove remove, u:upstream upstream' args=''
eval ``''{auxrc/getflags $*} || exec auxrc/usage
if(! ~ $#* 0)
        exec auxrc/usage

if(~ $pushall 1)
        branch=`$nl{cd .git/refs/heads && walk -f}
if(~ $#branch 0)
        branch=`{git/branch}
if(~ $#branch 0)
        die 'no branches'
if(~ $force 1)
        force=-f
if(~ $debug 1)
        debug='-d'

if(~ $#upstream 0)
        upstream=origin

remotes=`$nl{git/conf -a 'remote "'$upstream'".url'}
```

```
if(~ $#remotes 0)
        remotes=$upstream
branch=-b^$branch
if(! ~ $#remove 0)
        remove=-r^$remove
for(remote in $remotes){
        updates=`$nl{git/send $debug $force $branch $remove $remote} || die $status
        for(ln in $updates){
                u=`{echo $ln}
                refpath=`{echo $u(2) | subst '^refs/heads/' '.git/refs/remotes/'$upstream'/'}
                switch($u(1)){
                case update;
                        mkdir -p `{basename -d $refpath}
                        echo $u(4) > $refpath
                        echo $u(2)^':' $u(3) '=>' $u(4)
                case delete;
                        echo $u(2)^': removed'
                        rm -f $refpath
                case uptodate;
                        echo $u(2)^': up to date'
                }
        }
}
exit ''
```

14.4.1 Sending objects locally

14.4.2 Sending objects in a pack

14.5 Cloning a repository: `git clone`

⟨*git9/clone.rc* 51⟩≡

```
#!/bin/rc
rfork en
. /lib/git/common.rc

flagfmt='d:debug, b:branch branch'; args='remote [local]'
eval ``{auxrc/getflags $*} || exec auxrc/usage
if(~ $debug 1)
        debug=(-d)

remote=$1
local=$2

if(~ $#remote 0)
        exec auxrc/usage
if(~ $#local 0)
        local=`$nl{basename `$nl{echo $remote | sed 's@.*/([^/]+)/*$@\1@'} .git}
if(~ $#branch 1)
        branchflag=(-b $branch)

if(test -e $local && ~ `{ls $local | sed 1q | wc -l} 1)
        die 'destination already exists:' $local

fn clone{
        flag +e
        mkdir -p $local/.git
```

```
mkdir -p $local/.git/fs
mkdir -p $local/.git/objects/pack/
mkdir -p $local/.git/refs/heads/

cd $local

>>.git/config {
        echo '[remote "origin"]'
        echo '  url='$remote
}
{git/get  $debug $branchflag $remote >[2=3] | awk '
        BEGIN{
                headref=""
                if(ENVIRON["branch"] != "")
                        headref="refs/remotes/origin/"ENVIRON["branch"]
                headhash=""
        }
        /^symref / && headref == "" {
                if($2 == "HEAD"){
                        gsub("^refs/heads", "refs/remotes/origin", $3)
                        gsub("^refs/tags", "refs/remotes/origin/tags", $3)
                }
        }
        /^remote /{
                if($2=="HEAD"){
                        headhash=$3
                }else if(match($2, "^refs/(heads|tags)/")){
                        gsub("^refs/heads", "refs/remotes/origin", $2)
                        if($2 == headref || (headref == "" && $3 == headhash))
                                headref=$2
                        outfile = ".git/" $2
                        outdir = outfile
                        gsub("/?[^/]*/?$", "", outdir)
                        system(sprintf("mkdir -p %q", outdir))
                        print $3 > outfile
                        close(outfile)
                }
        }
        END{
                if(headref != ""){
                        remote = headref;
                        refdir = headref;
                        gsub("/?[^/]*/?$", "", refdir)
                        gsub("^refs/remotes/origin", "refs/heads", headref)
                        system("mkdir -p `{basename -d .git/"headref"}");
                        system("cp .git/" remote " .git/" headref)
                        print "ref: " headref > ".git/HEAD"
                }else if(headhash != ""){
                        print "warning: detached head "headhash > "/fd/2"
                        print headhash > ".git/HEAD"
                }
        }
'} |[3] tr '\x0d' '\x0a' || die 'could not clone repository'

tree=.git/fs/HEAD/tree
lbranch=`{git/branch}
rbranch=`{echo $lbranch | subst 'heads' 'remotes/origin'}
echo checking out repository...
if(test -f .git/refs/$rbranch){
        mkdir -p `{basename -d .git/refs/$lbranch}
```

```
                cp .git/refs/$rbranch .git/refs/$lbranch
                git/fs
                @ {builtin cd $tree && tar cif /fd/1 .} | @ {tar xf /fd/0} \
                        || die 'checkout failed:' $status
                {for(f in `$nl{cd $tree && walk -f})
                        echo 'T NOQID 0 '$f} > .git/INDEX9
        }
        if not{
                echo no default branch >[1=2]
                echo check out your code with git/branch >[1=2]
        }
}

fn sigint {
        echo cancelled clone $remote: cleaning $local >[1=2]
        unmount $local/.git/fs >[2]/dev/null
        rm -rf $local
        exit interrupted
}

 @{clone}
st=$status
if(! ~ $st ''){
        echo failed to clone $remote: cleaning $local >[1=2]
        unmount $local/.git/fs >[2]/dev/null
        rm -rf $local
        exit $st
}
exit ''
```

14.5.1 Fetching objects

14.5.2 Importing references

Chapter 15

Networking

15.1 git:// protocol

15.1.1 Reading packets

15.1.2 Writing packets

15.2 Capabilities

15.3 git:// client

15.3.1 Fetching remote objects

15.3.2 Fetching pack files

`read_pkt_refs()`

`_handle_upload_pack_head()`

`_handle_upload_pack_tail()`

15.3.3 Sending pack files

15.4 git:// server

15.4.1 git-upload-pack

15.4.2 git-receive-pack

Chapter 16

Algorithms

16.1 SHA1

16.1.1 Overview

16.1.2 32 bits operators

16.1.3 Entry point: `sha1()`

16.1.4 Endianess

16.1.5 Filling blocks

16.1.6 Looping over blocks

16.1.7 Padding

16.2 Unzip

16.2.1 Overview

16.2.2 Data structures

16.2.3 Error management

16.2.4 IO helpers

Input bits helpers

Output helpers

16.2.5 Huffman trees

16.2.6 Sliding window back references (LZ77)

16.2.7 Entry point: `inflate()`

16.2.8 `inflate_loop()`

Head

Block

CData

zlen

`zdist`

Dist

CRC

Done

16.2.9 Advanced features

Adler32 CRC

No compression block type

Specialized huffman tree

16.2.10 Optimizations

`NeedBits`

`DistOne`

16.3 Zip

16.3.1 Entry point: `deflate()`

16.4 Diff

16.4.1 Overview

16.4.2 Data structures

16.4.3 Entry point: `Diffs.diff()`

16.4.4 Edit distance basic algorithm (O(n^2) in space/time)

16.4.5 Myers algorithm (O(n) in space in best-case)

16.5 Diff3

16.5.1 Overview

16.5.2 Data structures

16.5.3 Entry point: `diff3()`

16.5.4 Displaying merging conflicts

Chapter 17

Advanced Features TODO

17.1 Tracking symbolic links

17.2 Splitting a large repository: submodules

17.3 Configuration file: `.git/config`

17.3.1 Initialization

17.3.2 `Config.t`

17.3.3 Reading a configuration file

17.3.4 Writing a configuration file

17.3.5 Stacked configurations: `/.gitconfig` and `.git/config`

17.4 Commit rules and checks: `.git/hooks/`

17.5 Ignoring files: `.gitignore`

17.6 Archeology: Grafts

17.6.1 Reading a graft file

17.6.2 Writing a graft file

17.6.3 Using a graft file

17.7 Rename detection

Chapter 18

Advanced Commands TODO

18.1 Tagging a commit with a name

18.1.1 `Tag.t`

18.1.2 Reading a tag

18.1.3 Writing a tag

18.1.4 `git tag`

18.1.5 `git clone` and tags

18.1.6 `git pull` and tags

18.2 Reference history

18.2.1 `git reflog`

18.3 Stash

18.3.1 `git stash`

18.4 Developer commands

18.4.1 Finding the diff causing a regression: `git bisect`

18.4.2 Reverting a change: `git revert`

⟨*git9/revert.rc* 59⟩≡

```
#!/bin/rc
rfork en
. /lib/git/common.rc

gitup

flagfmt='c:query query' args='file ...'
if (! eval ''{auxrc/getflags $*} || ~ $#* 0)
        exec auxrc/usage

if(~ $#query 0)
```

```
        query=HEAD
commit=‘{git/query -p $query}

files=‘$nl{cleanname -d $gitrel $* | drop $gitroot}
for(f in ‘$nl{git/walk -c -fRM -b $query $files}){
        mkdir -p ‘{basename -d $f}
        cp -x -- $commit/tree/$f $f
        touch -c $f
        git/add $f
}
exit ''
```

18.4.3 Finding the author of some code: `git blame`

18.4.4 Finding code: `git grep`

18.4.5 Creating an archive: `git archive`

18.5 Advanced merging commands

18.5.1 Rebase: `git rebase`

⟨*git9/rebase.rc* 60⟩≡

```
#!/bin/rc -e

rfork en

. /lib/git/common.rc
gitup

flagfmt='s:src src, n:nocommit'; args='onto'
eval ‘''{auxrc/getflags $*} || exec auxrc/usage
if(~ $#* 0) exec auxrc/usage

if(~ $#src 0)
        src=‘{git/branch}
nflag=''
if(~ $nocommit 1)
        nflag='-n'

dst=‘{git/query $1}
com=‘{git/query $dst  $src @}
if(~ $dst $com)
        die 'nothing to rebase, doofus'
git/log -se $dst'..'$src  | awk '
BEGIN{
        src=ENVIRON["src"];
        dst=ENVIRON["dst"];
}
{
        if(!done)
                print "git/branch -nb "dst" rebase.wip";
        c=$1; $1="";
        print "git/export "c" | git/import '$nflag'#"$0;
        done=1
}
END{
```

```
        if(!done)
                print "git/branch -nb "dst" ";
        else{
                print "git/branch -nb rebase.wip "src
                print "git/branch -r rebase.wip";
        }
}'
```

18.5.2 Cherry-pick: `git cherry-pick`

18.6 Plumbing commands

18.6.1 `git fetch`

18.6.2 `git symbolic-ref`

18.6.3 `git rev-list`

18.6.4 `git diff-tree`

18.6.5 `git commit-tree`

18.6.6 `git ls-tree`

18.6.7 `git remote add`

18.6.8 `git ls-remote`

18.7 Miscellaneous commands

18.7.1 `git filter-branch`

Chapter 19

Advanced Networking TODO

19.1 Other clients

19.1.1 `ssh://`

19.1.2 `http://`

19.2 Other servers

19.2.1 `http://`

19.3 Other client/server capabilities

19.3.1 Report status

19.3.2 Quiet

19.3.3 Thin pack

19.3.4 Side band 64k

Chapter 20

Advanced Topics TODO

20.1 Bytes versus ASCII versus UTF-8

20.2 Reliability and signals

20.3 Optimizations

20.4 Fast import and export

20.5 Alternates

20.6 Other repository format

20.6.1 Bare repository

20.6.2 `.git` file

20.7 Security

Chapter 21

Conclusion

Appendix A

Debugging

Appendix B

Profiling

Appendix C

Error Management

⟨*fn* `die` *(common.rc)* 67⟩≡ (69)

```
fn die{
        >[1=2] echo $0: $*
        exit $"*
}
```

Appendix D

Utilities

Appendix E

Extra Code

E.1 git9/

git9/common.rc

⟨*git9/common.rc* 69⟩≡
⟨*global* `nl` *(common.rc)* 32d⟩

⟨*fn* `die` *(common.rc)* 67⟩

```
fn usage{
        >[1=2] echo -n 'usage:' $usage
        exit 'usage'
}

fn subst {
        awk '
        BEGIN{ARGC=0}
        {sub(ARGV[1], ARGV[2]); print}
        ' $*
}

fn drop {
        awk '
        BEGIN{ARGC=0}
        {
                if(index($0, ARGV[1]) == 1)
                        $0=substr($0, length(ARGV[1])+1)
                print
        }
        ' $*
}

fn mergeperm {
        if(~ $1 /dev/null && cmp $2 $3>/dev/null)
                status=gone
        if not if (~ $3 /dev/null && cmp $1 $2>/dev/null)
                status=gone
        if not {
                mergedperms='-x'
                if(test -x $2){
                        if(test -x $1 -a -x $3)
                                mergedperms='+x'
                }
                if not{
```

```
                        if(test -x $1 -o -x $3)
                                mergedperms='+x'
                }
                status=()
        }
}

fn whoami{
        name=`$nl{git/conf user.name}
        email=`$nl{git/conf user.email}
        if(test -f /adm/keys.who){
                if(~ $name '')
                        name=`$nl{awk -F'|' '$1=="'$user'" {x=$3} END{print x}' </adm/keys.who}
                if(~ $email '')
                        email=`$nl{awk -F'|' '$1=="'$user'" {x=$5} END{print x}' </adm/keys.who}
        }
        if(~ $name '')
                name=$user
        if(~ $email '')
                email=$user@$sysname
        status=''
}

# merge1 out ours base theirs
fn merge1 {@{
        rfork e
        n=$pid
        out=$1
        ours=$2
        base=$3
        theirs=$4
        tmp=$out.tmp
        while(test -f $tmp){
                tmp=$tmp.$n
                n=`{echo $n + 1 | hoc}
        }

        if(! test -f $ours)
                ours=/dev/null
        if(! test -f $base)
                base=/dev/null
        if(! test -f $theirs)
                theirs=/dev/null

        if(mergeperm $ours $base $theirs){
                mkdir -p `$nl{basename -d $tmp}
                if(! merge3 $ours $base $theirs > $tmp)
                        echo merge needed: $out >[1=2]
                mv $tmp $out
                git/add $out
                chmod $mergedperms $out
        }
        if not {
                rm -f $tmp $out
                echo R NOQID 0 $out >> $gitroot/.git/INDEX9
        }
}}
```

⟨*fn* `gitup` *(common.rc)* 32c⟩

git9/add.rc

git9/branch.rc

git9/clone.rc

git9/commit.rc

git9/compat.rc

⟨*git9/compat.rc* 71⟩≡

```
#!/bin/rc

rfork e

opts=()
args=()
nl='
'

fn cmd_init{
        while(! ~ $#* 0){
                switch($1){
                case --bare
                        # ignore
                case --
                        # go likes to use these
                case -*
                        die unknown command init $*
                case *
                        args=($args $1)
                }
                shift
        }
        git/init $opts $args
}

fn cmd_clone{
        branch=()
        while( ! ~ $#* 0){
                switch($1){
                case -b
                        branch=$2
                        shift
                case --
                        # go likes to use these
                case -*
                        die unknown command clone $*
                case *
                        args=($args $1)
                }
                shift
        }
        git/clone $opts $args
        if(~ $#branch 1)
                git/branch -n -b $1 origin/$1
}

fn cmd_pull{
```

```
        if(~ $1 -*)
                die unknown options for pull $*
        git/pull
}

fn cmd_fetch{
        while(~ $#* 0){
                switch($1){
                case --all
                        opts=($opts -a)
                case -f
                        opts=($opts -u $2)
                        shift
                case --
                        # go likes to use these
                case -*
                        die unknown command clone $*
                case *
                        args=($args $1)
                }
                shift
        }
        git/pull -f $opts
}


fn cmd_checkout{
        if(~ $1 -*)
                die unknown command pull $*
        if(~ $#* 0)
                die git checkout branch
        git/branch $b
}

fn cmd_submodule {
        if(test -f .gitmodules)
                die 'submodules unsupported'
}

fn cmd_rev-parse{
        while(~ $1 -*){
                switch($1){
                case --git-dir
                        echo $gitroot/.git
                        shift
                case --abbrev-ref
                        echo `{dcmd git9/branch | sed s@^heads/@@g}
                        shift
                case *
                        die unknown option $opt
                }
                shift
        }
}

fn cmd_show-ref{
        if(~ $1 -*)
                die unknown command pull $*
        filter=cat
        if(~ $#* 0)
```

```
                filter=cat
        if not
                filter='-e(^|/)'^$*^'$'
        for(b in `$nl{cd $gitroot/.git/refs/ && walk -f})
                echo `{cat $gitroot/.git/refs/$b} refs/$b
}

fn cmd_rev-parse{
        switch($1){
        case --git-dir
                echo `{git/conf -r}^/.git
        case *
                die 'unknown rev-parse '$*
        }
}

fn cmd_remote{
        if({! ~ $#* 3 && ! ~ $#* 4} || ! ~ $1 add)
                die unimplemented remote cmd $*
        name=$2
        url=$3
        if(~ $3 '--')
                url=$4
        >>$gitroot/.git/config{
                echo '[remote "'$name'"]'
                echo '  url='$url
        }
}

fn cmd_log{
        count=()
        format=''
        while(~ $1 -*){
                switch($1){
                case --format
                        format=$2
                        shift
                case '--format='*
                        format=`{echo $1 | sed 's/--format=//g'}
                case -n
                        count=-n$2
                        shift
                case -n*
                        count=$1
                case *
                        dprint option $opt
                }
                shift
        }
        @{cd $gitroot && git/fs}
        switch($format){
        case ''
                git/log $count
        case '%H:%ct'
                for(c in `{git/log -s $count| awk '{print $1}'})
                        echo $c:`{mtime $gitroot/.git/fs/object/$c/msg}
        case '%h %cd'
                for(c in `{git/log -s $count| awk '{print $1}'})
                        echo $c `{date `{mtime $gitroot/.git/fs/object/$c/msg}}
        }
```

```
}

fn cmd_show{
        cmd_log -n1 $*
}

fn cmd_ls-remote{
        if(~ $1 -q)
                shift
        remote=‘$nl{git/conf 'remote "'$1'".url'}
        if(~ $#remote 0)
                remote=$1
        git/get -l $remote | awk '/^remote/{print $3"\t"$2}'
}

fn cmd_version{
        echo git version 2.2.0
}

fn cmd_status{
        echo
}

fn usage{
        echo 'git <command> <args>' >[1=2]
        exit usage
}

fn die {
        >[1=2] echo git $_cmdname: $*
        exit $_cmdname: $*
}

_cmdname=$1
if(~ $0 *compat){
        ramfs -m /n/gitcompat
        touch /n/gitcompat/git
        bind $0 /n/gitcompat/git
        path=( /n/gitcompat $path )
        exec rc
}

if(~ $#gitcompatdebug 1)
        echo running  $* >>/tmp/gitlog

if(~ $1 -c)
        shift 2
if(~ $1 -c*)
        shift 1
if(! test -f '/env/fn#cmd_'$1)
        die git $1: commmand not implemented
if(! ~ $1 init && ! ~ $1 clone)
        gitroot=‘{git/conf -r} || die repo

if(~ $#gitcompatdebug 1)
        cmd_$1 $*(2-) | tee >>/tmp/gitlog
if not
        cmd_$1 $*(2-)
exit ''
```

git9/diff.rc

git9/export.rc

⟨*git9/export.rc* 75⟩≡

```
#!/bin/rc
rfork ne
. /lib/git/common.rc

patchname=/tmp/git.patchname.$pid
patchfile=/tmp/git.patchfile.$pid
fn sigexit{
        rm -f $patchname $patchfile
}

gitup

flagfmt='o:patchdir patchdir'; args='[query]'
eval ''{auxrc/getflags $*} || exec auxrc/usage

if(~ $#patchdir 1 && ! test -d $patchdir)
        mkdir -p $patchdir

q=$*
if(~ $#q 0)
        q=HEAD
commits='{git/query $q || die $status}
n=1
m=$#commits


# sleazy hack: we want to run
# under rfork m for the web ui,
# so don't error if we can't mount
mntgen /mnt/scratch >[2]/dev/null || status=''
for(c in $commits){
        cp='{git/query -p $c}
        pp='{git/query -p $c'~'}
        fc='$nl{git/query -c $c~ $c | sed 's/^..//'}

        @{
                rfork n
                cd /mnt/scratch
                if(test -d $pp/tree)
                        bind $pp/tree a
                if(test -d $cp/tree)
                        bind $cp/tree b

                echo From: '{cat $cp/author}
                echo Date: '{date -uf'WW, DD MMM YYYY hh:mm:ss Z' '{walk -em $cp/author}}
                <$cp/msg awk '
                NR == 1 {
                        n = ENVIRON["n"]
                        m = ENVIRON["m"]
                        msg=$0
                        if(m > 1)
                                patch = sprintf("[PATCH %d/%d]", n, m)
```

```
                        else
                                patch = "[PATCH]"
                        printf "Subject: %s %s\n\n", patch, msg

                        gsub("^[        ]|[     ]$", "", msg)
                        gsub("[^a-zA-Z0-9_]+", "-", msg)
                        printf "%.4d-%s.patch", n, msg >ENVIRON["patchname"]
                        next
                }
                {
                        print
                }'
                echo '---'
                echo diff `{basename $pp} `{basename $cp}
                for(f in $fc){
                        a=a/$f
                        if(! test -f $a)
                                a=/dev/null
                        b=b/$f
                        if(! test -f $b)
                                b=/dev/null
                        diff -ur $a $b
                }
                echo '-- '
                echo '9'
        } >$patchfile
        if(~ $#patchdir 0){
                cat $patchfile
                ! ~ $n $m && echo
        }
        if not{
                f=$patchdir/`{cat $patchname}
                mv $patchfile $f
                echo $f
        }
        n=`{echo $n + 1 | bc}
}
exit ''
```

git9/hist.rc

⟨*git9/hist.rc* 76⟩≡

```
#!/bin/rc
rfork ne

. /lib/git/common.rc
gitup

flagfmt='n:num count'; args='files...'
eval `''{auxrc/getflags $*} || exec auxrc/usage
if(~ $#* 0) exec auxrc/usage

fn dodiff {
        while(t=`{read}){
                h=$t(1)
                o=$gitfs/object/`{git/query $h~}
                c=$gitfs/object/$h
                echo 'Hash:' $h
```

```
                echo -n 'Date: '; date '{walk -em $c/msg}
                echo -n 'Author: '; cat $c/author
                echo -n 'Message: '; cat $c/msg
                echo
                for(f in $files){
                        curr=$c/tree/$f
                        prev=$o/tree/$f
                        if(! test -e $curr)
                                curr=/dev/null
                        if(! test -e $prev)
                                prev=/dev/null
                        diff -u $prev $curr
                }
                echo --' '
                echo ⑨
                echo
        }
}

files='{cleanname -d $gitrel $*}
args=(-s $files)
if(! ~ $#num 0)
        args=(-n $num $args)
git/log $args | dodiff
```

git9/import.rc

⟨*git9/import.rc* 77⟩≡

```
#!/bin/rc
rfork ne
. /lib/git/common.rc

diffpath=/tmp/gitimport.$pid.diff
fn sigexit {
        rm -f $diffpath
}

fn apply1 {
        adate='{seconds $adate}
        files='$nl{patch -np1 < $diffpath}
        if(! git/walk -q $files){
                >[1=2] {
                        echo patch would clobber files:
                        git/walk $files
                        exit clobber
                }
        }
        echo applying $msg | sed 1q
        if(! files='$nl{patch -p1 < $diffpath})
                die 'patch failed'
        for(f in $files){
                if(test -e $f)
                        git/add $f
                if not
                        git/add -r $f
        }
        git/walk -fRMA $files
        if(~ $#nocommit 0){
```

```
                if(hash=`{git/save -n $aname -e $amail -N $name -E $email -m $msg -d $adate $parents $files}){
                        echo $hash > $refpath
                        for(f in $files)
                                echo T NOQID 0 $f >> .git/INDEX9
                }
        }
        exit ''
}


fn apply @{
        git/fs
        amail=''
        aname=''
        msg=''
        whoami
        parents='-p'^`{git/query HEAD}
        branch=`{git/branch}
        if(test -e $gitfs/branch/$branch/tree)
                refpath=.git/refs/$branch
        if not if(test -e $gitfs/object/$branch/tree)
                refpath=.git/HEAD
        if not
                die 'invalid branch:' $branch
        awk '
        function doapply(){
                if(aname == "" || amail == "" || date == "" || gotmsg == "")
                        exit("missing headers");
                printf "%s", aname > "/env/aname"
                printf "%s", amail > "/env/amail"
                printf "%s", date > "/env/adate"
                if(system("rc -c apply1") != 0)
                        exit("patch failed");
                close("/env/aname")
                close("/env/amail")
                close("/env/adate")
                close(ENVIRON["diffpath"])
                applied = 1
        }
        BEGIN{
                state="headers"
        }
        state=="diff" && !/^[-+ @]|^diff|^$|^\\ No newline at end of file$/{
                state="footers";
                doapply();
        }
        state=="footers" && /^⓽$/{
                state="headers"
                next
        }
        state=="headers" && /^[fF]rom:/ {
                sub(/^From:[ \t]*/, "", $0);
                aname=$0;
                amail=$0;
                sub(/[ \t]*<.*$/, "", aname);
                sub(/^[^<]*</, "", amail);
                sub(/>[^>]*$/, "", amail);
        }
        state=="headers" && /^[Dd]ate:/{
                sub(/^Date:[ \t]*/, "", $0)
```

```
                date=$0
        }
        state=="headers" && /^Subject:/{
                sub(/^Subject:[ \t]*([Rr][Ee]:)?[ \t]*(\[[^\]]*\][ \t]*)*/, "", $0);
                gotmsg = 1
                print > "/env/msg"
        }
        state=="headers" && /^$/ {
                state="body"
        }
        (state=="headers" || state=="body") && (/^diff / || /^---( |$)/){
                state="diff"
                next
        }
        state=="body" && /^[    ]*$/ {
                empty=1
                next
        }
        state=="body" {
                if(empty)
                        printf "\n" > "/env/msg"
                empty=0
                sub(/[  ]+$/, "")
                print > "/env/msg"
        }
        state=="diff" {
                print > ENVIRON["diffpath"]
        }
        END{
                if(!failed && state == "diff" && doapply() != 0){
                        print "unable to apply patch" > "/fd/2"
                        exit("mispatch")
                }
        }
        '
}

gitup

flagfmt='n:nocommit'; args='file ...'
eval ''{auxrc/getflags $*} || exec auxrc/usage

patches=(/fd/0)
if(! ~ $#* 0)
        patches=`{cleanname -d $gitrel $*}
for(p in $patches){
        # upas serves the decoded header and body separately,
        # so we cat them together when applying a upas message.
        #
        # this allows mime-encoded or line-wrapped patches.
        if(test -d $p && test -f $p/header && test -f $p/body)
                {{cat $p/header; echo; cat $p/body} | apply} || die $status
        if not
                apply < $p
}
exit ''
```

git9/init.rc

git9/merge.rc

git9/pull.rc

git9/push.rc

git9/rebase.rc

git9/revert.rc

git9/rm.rc

git9/conf.c

⟨*function* showconf 80a⟩≡ (81b)

```
static int
showconf(char *cfg, char *sect, char *key)
{
    char *ln, *p;
    Biobuf *f;
    int foundsect, nsect, nkey, found;

    if((f = Bopen(cfg, OREAD)) == nil)
        return 0;

    found = 0;
    nsect = sect ? strlen(sect) : 0;
    nkey = strlen(key);
    foundsect = (sect == nil);
    while((ln = Brdstr(f, '\n', 1)) != nil){
        p = strip(ln);
        if(*p == '[' && sect){
            foundsect = strncmp(sect, ln, nsect) == 0;
        }else if(foundsect && strncmp(p, key, nkey) == 0){
            p = strip(p + nkey);
            if(*p != '=')
                continue;
            p = strip(p + 1);
            print("%s\n", p);
            found = 1;
            if(!showall){
                free(ln);
                goto done;
            }
        }
        free(ln);
    }
done:
    return found;
}
```

⟨*function* usage 80b⟩≡ (81b)

```
void
usage(void)
{
    fprint(2, "usage: %s [-f file] [-r] keys..\n", argv0);
    fprint(2, "\t-f:    use file 'file' (default: .git/config)\n");
```

```
    fprint(2, "\t r:    print repository root\n");
    exits("usage");
}
```

⟨*function* main 81a⟩≡ (81b)

```
void
main(int argc, char **argv)
{
    char repo[512], *p, *s;
    int i, j, nrel;

    ARGBEGIN{
    case 'r':   findroot++;         break;
    case 'a':   showall++;          break;
    default:    usage();            break;
    case 'f':
        if(nfile == nelem(file))
            sysfatal("too many configs");
        file[nfile++]=EARGF(usage());
        break;
    }ARGEND;
```

⟨git9/conf.c 81b⟩≡

```
#include <u.h>
#include <libc.h>
#include <ctype.h>

#include "git.h"

int findroot;
int showall;
int nfile;
char    *file[32];

⟨function showconf 80a⟩


⟨function usage 80b⟩

⟨function main 81a⟩

    gitinit(repo, sizeof(repo), &nrel);
    if(findroot){
        print("%s\n", repo);
        exits(nil);
    }
    if(nfile == 0){
        if((file[nfile++] = smprint("%s/.git/config", repo)) == nil)
            sysfatal("smprint: %r");
        if((p = getenv("home")) != nil)
            if((file[nfile++] = smprint("%s/lib/git/config", p)) == nil)
                sysfatal("smprint: %r");
        file[nfile++] = "/sys/lib/git/config";
    }

    for(i = 0; i < argc; i++){
        if((p = strchr(argv[i], '.')) == nil){
            s = nil;
            p = argv[i];
        }else{
```

```
            *p = 0;
            p++;
            s = smprint("[%s]", argv[i]);
        }
        for(j = 0; j < nfile; j++)
            if(showconf(file[j], s, p))
                break;
    }
    exits(nil);
}
```

git9/delta.c

⟨*function* hash 82a⟩≡ (85)

```
static u64int
hash(void *p, int n)
{
    return murmurhash2(p, n);
}
```

⟨*function* addblk 82b⟩≡ (85)

```
static void
addblk(Dtab *dt, void *buf, int len, int off, u64int h)
{
    int i, sz, probe;
    Dblock *db;

    probe = h % dt->sz;
    while(dt->b[probe].buf != nil){
        if(len == dt->b[probe].len && memcmp(buf, dt->b[probe].buf, len) == 0)
            return;
        probe = (probe + 1) % dt->sz;
    }
    assert(dt->b[probe].buf == nil);
    dt->b[probe].buf = buf;
    dt->b[probe].len = len;
    dt->b[probe].off = off;
    dt->b[probe].hash = h;
    dt->nb++;
    if(dt->sz < 2*dt->nb){
        sz = dt->sz;
        db = dt->b;
        dt->sz *= 2;
        dt->nb = 0;
        dt->b = eamalloc(dt->sz, sizeof(Dblock));
        for(i = 0; i < sz; i++)
            if(db[i].buf != nil)
                addblk(dt, db[i].buf, db[i].len, db[i].off, db[i].hash);
        free(db);
    }
}
```

⟨*function* lookup 82c⟩≡ (85)

```
static Dblock*
lookup(Dtab *dt, uchar *p, int n)
{
    int probe;
    u64int h;
```

```
    h = hash(p, n);
    for(probe = h % dt->sz; dt->b[probe].buf != nil; probe = (probe + 1) % dt->sz){
        if(dt->b[probe].hash != h)
            continue;
        if(n != dt->b[probe].len)
            continue;
        if(memcmp(p, dt->b[probe].buf, n) != 0)
            continue;
        return &dt->b[probe];
    }
    return nil;
}
```

⟨*function* nextblk 83a⟩≡ (85)

```
static int
nextblk(uchar *s, uchar *e)
{
    u32int gh;
    uchar *p;

    if((e - s) < Minchunk)
        return e - s;
    p = s + Minchunk;
    if((e - s) > Maxchunk)
        e = s + Maxchunk;
    gh = 0;
    while(p != e){
        gh = (gh<<1) + geartab[*p++];
        if((gh & Splitmask) == 0)
            break;
    }
    return p - s;
}
```

⟨*function* dtinit 83b⟩≡ (85)

```
void
dtinit(Dtab *dt, Object *obj)
{
    uchar *s, *e;
    u64int h;
    vlong n, o;

    o = 0;
    s = (uchar*)obj->data;
    e = s + obj->size;
    dt->o = ref(obj);
    dt->nb = 0;
    dt->sz = 128;
    dt->b = eamalloc(dt->sz, sizeof(Dblock));
    dt->base = (uchar*)obj->data;
    dt->nbase = obj->size;
    while(s != e){
        n = nextblk(s, e);
        h = hash(s, n);
        addblk(dt, s, n, o, h);
        s += n;
        o += n;
    }
}
```

⟨*function* dtclear 84a⟩≡ (85)

```
void
dtclear(Dtab *dt)
{
    unref(dt->o);
    free(dt->b);
}
```

⟨*function* emitdelta 84b⟩≡ (85)

```
static int
emitdelta(Delta **pd, int *nd, int cpy, int off, int len)
{
    Delta *d;

    *nd += 1;
    *pd = earealloc(*pd, *nd, sizeof(Delta));
    d = &(*pd)[*nd - 1];
    d->cpy = cpy;
    d->off = off;
    d->len = len;
    return len;
}
```

⟨*function* stretch 84c⟩≡ (85)

```
static int
stretch(Dtab *dt, Dblock *b, uchar *s, uchar *e, int n)
{
    uchar *p0, *p, *q, *eb;

    if(b == nil)
        return n;
    p = s + n;
    q = dt->base + b->off + n;
    p0 = p;
    if(dt->nbase < (1<<24)-1)
        eb = dt->base + dt->nbase;
    else
        eb = dt->base + (1<<24)-1;
    while(1){
        if(p == e || q == eb)
            break;
        if(*p != *q)
            break;
        p++;
        q++;
    }
    return n + (p - p0);
}
```

⟨*function* deltify 84d⟩≡ (85)

```
Delta*
deltify(Object *obj, Dtab *dt, int *pnd)
{
    Delta *d;
    Dblock *b;
    uchar *s, *e;
    vlong n, o;

    o = 0;
    d = nil;
```

```
    s = (uchar*)obj->data;
    e = s + obj->size;
    *pnd = 0;
    while(s != e){
        n = nextblk(s, e);
        b = lookup(dt, s, n);
        n = stretch(dt, b, s, e, n);
        if(b != nil)
            emitdelta(&d, pnd, 1, b->off, n);
        else
            emitdelta(&d, pnd, 0, o, n);
        s += n;
        o += n;
    }
    return d;
}
```

⟨git9/delta.c 85⟩≡

```
#include <u.h>
#include <libc.h>

#include "git.h"

enum {
    Minchunk    = 128,
    Maxchunk    = 8192,
    Splitmask   = (1<<8)-1,
};

static u32int geartab[] = {
    0x67ed26b7, 0x32da500c, 0x53d0fee0, 0xce387dc7, 0xcd406d90, 0x2e83a4d4, 0x9fc9a38d, 0xb67259dc,
    0xca6b1722, 0x6d2ea08c, 0x235cea2e, 0x3149bb5f, 0x1beda787, 0x2a6b77d5, 0x2f22d9ac, 0x91fc0544,
    0xe413acfa, 0x5a30ff7a, 0xad6fdde0, 0x444fd0f5, 0x7ad87864, 0x58c5ff05, 0x8d2ec336, 0x2371f853,
    0x550f8572, 0x6aa448dd, 0x7c9ddbcf, 0x95221e14, 0x2a82ec33, 0xcbec5a78, 0xc6795a0d, 0x243995b7,
    0x1c909a2f, 0x4fded51c, 0x635d334b, 0x0e2b9999, 0x2702968d, 0x856de1d5, 0x3325d60e, 0xeb6a7502,
    0xec2a9844, 0x0905835a, 0xa1820375, 0xa4be5cab, 0x96a6c058, 0x2c2ccd70, 0xba40fce3, 0xd794c46b,
    0x8fbae83e, 0xc3aa7899, 0x3d3ff8ed, 0xa0d42b5b, 0x571c0c97, 0xd2811516, 0xf7e7b96c, 0x4fd2fcbd,
    0xe2fdec94, 0x282cc436, 0x78e8e95c, 0x80a3b613, 0xcfbee20c, 0xd4a32d1c, 0x2a12ff13, 0x6af82936,
    0xe5630258, 0x8efa6a98, 0x294fb2d1, 0xdeb57086, 0x5f0fddb3, 0xeceda7ce, 0x4c87305f, 0x3a6d3307,
    0xe22d2942, 0x9d060217, 0x1e42ed02, 0xb6f63b52, 0x4367f39f, 0x055cf262, 0x03a461b2, 0x5ef9e382,
    0x386bc03a, 0x2a1e79c7, 0xf1a0058b, 0xd4d2dea9, 0x56baf37d, 0x5daff6cc, 0xf03a951d, 0xaef7de45,
    0xa8f4581e, 0x3960b555, 0xffbfff6d, 0xbe702a23, 0x8f5b6d6f, 0x061739fb, 0x98696f47, 0x3fd596d4,
    0x151eac6b, 0xa9fcc4f5, 0x69181a12, 0x3ac5a107, 0xb5198fe7, 0x96bcb1da, 0x1b5ddf8e, 0xc757d650,
    0x65865c3a, 0x8fc0a41a, 0x87435536, 0x99eda6f2, 0x41874794, 0x29cff4e8, 0xb70efd9a, 0x3103f6e7,
    0x84d2453b, 0x15a450bd, 0x74f49af1, 0x60f664b1, 0xa1c86935, 0xfdafbce1, 0xe36353e3, 0x5d9ba739,
    0xbc0559ba, 0x708b0054, 0xd41d808c, 0xb2f31723, 0x9027c41f, 0xf136d165, 0xb5374b12, 0x9420a6ac,
    0x273958b6, 0xe6c2fad0, 0xebdc1f21, 0xfb33af8b, 0xc71c25cd, 0xe9a2d8e5, 0xbeb38a50, 0xbceb7cc2,
    0x4e4e73f0, 0xcd6c251d, 0xde4c032c, 0x4b04ac30, 0x725b8b21, 0x4eb8c33b, 0x20d07b75, 0x0567aa63,
    0xb56b2bb7, 0xc1f5fd3a, 0xcafd35ca, 0x470dd4da, 0xfe4f94cd, 0xfb8de424, 0xe8dbcf40, 0xfe50a37a,
    0x62db5b5d, 0xf32f4ab6, 0x2c4a8a51, 0x18473dc0, 0xfe0cbb6e, 0xfe399efd, 0xdf34ecc9, 0x6ccd5055,
    0x46097073, 0x139135c2, 0x721c76f6, 0x1c6a94b4, 0x6eee014d, 0x8a508e02, 0x3da538f5, 0x280d394f,
    0x5248a0c4, 0x3ce94c6c, 0x9a71ad3a, 0x8493dd05, 0xe43f0ab6, 0x18e4ed42, 0x6c5c0e09, 0x42b06ec9,
    0x8d330343, 0xa45b6f59, 0x2a573c0c, 0xd7fd3de6, 0xeedeab68, 0x5c84dafc, 0xbbd1b1a8, 0xa3ce1ad1,
    0x85b70bed, 0xb6add07f, 0xa531309c, 0x8f8ab852, 0x564de332, 0xeac9ed0c, 0x73da402c, 0x3ec52761,
    0x43af2f4d, 0xd6ff45c8, 0x4c367462, 0xd553bd6a, 0x44724855, 0x3b2aa728, 0x56e5eb65, 0xeaf16173,
    0x33fa42ff, 0xd714bb5d, 0xfbd0a3b9, 0xaf517134, 0x9416c8cd, 0x534cf94f, 0x548947c2, 0x34193569,
    0x32f4389a, 0xfe7028bc, 0xed73b1ed, 0x9db95770, 0x468e3922, 0x0440c3cd, 0x60059a62, 0x33504562,
    0x2b229fbd, 0x5174dca5, 0xf7028752, 0xd63c6aa8, 0x31276f38, 0x0646721c, 0xb0191da8, 0xe00e6de0,
    0x9eac1a6e, 0x9f7628a5, 0xed6c06ea, 0x0bb8af15, 0xf119fb12, 0x38693c1c, 0x732bc0fe, 0x84953275,
    0xb82ec888, 0x33a4f1b3, 0x3099835e, 0x028a8782, 0x5fdd51d7, 0xc6c717b3, 0xb06caf71, 0x17c8c111,
```

```
    0x61bad754, 0x9fd03061, 0xe09df1af, 0x3bc9eb73, 0x85878413, 0x9889aaf2, 0x3f5a9e46, 0x42c9f01f,
    0x9984a4f4, 0xd5de43cc, 0xd294daed, 0xbecba2d2, 0xf1f6e72c, 0x5551128a, 0x83af87e2, 0x6f0342ba,
};
```

⟨*function* `hash` 82a⟩

⟨*function* `addblk` 82b⟩

⟨*function* `lookup` 82c⟩

⟨*function* `nextblk` 83a⟩

⟨*function* `dtinit` 83b⟩

⟨*function* `dtclear` 84a⟩

⟨*function* `emitdelta` 84b⟩

⟨*function* `stretch` 84c⟩

⟨*function* `deltify` 84d⟩

git9/fs.c

⟨*constant* `Eperm` 86a⟩≡ (100d)
```
#define Eperm   "permission denied"
```

⟨*constant* `Eexist` 86b⟩≡ (100d)
```
#define Eexist  "file does not exist"
```

⟨*constant* `Enotdir` 86c⟩≡ (100d)
```
#define Enotdir "is not a directory"
```

⟨*constant* `E2long` 86d⟩≡ (100d)
```
#define E2long  "path too long"
```

⟨*constant* `Enodir` 86e⟩≡ (100d)
```
#define Enodir  "not a directory"
```

⟨*constant* `Erepo` 86f⟩≡ (100d)
```
#define Erepo   "unable to read repo"
```

⟨*constant* `Eimpl` 86g⟩≡ (100d)
```
#define Eimpl   "not implemented"
```

⟨*constant* `Egreg` 86h⟩≡ (100d)
```
#define Egreg   "wat"
```

⟨*constant* `Ebadobj` 86i⟩≡ (100d)
```
#define Ebadobj "invalid object"
```

⟨*function* qpath 87a⟩≡ (100d)

```
vlong
qpath(Crumb *p, int idx, vlong id, vlong t)
{
    int h, i;
    vlong pp;
    Cache *c;
    Uqid *u;

    pp = p ? p->qid.path : 0;
    h = (pp*333 + id*7 + t) & (nelem(uqidcache) - 1);
    c = &uqidcache[h];
    u = c->cache;
    for(i=0; i <c->n ; i++){
        if(u->ppath == pp && u->oid == id && u->t == t && u->idx == idx)
            return (u->uqid << 8) | t;
        u++;
    }
    if(c->n == c->max){
        c->max += c->max/2 + 1;
        c->cache = erealloc(c->cache, c->max*sizeof(Uqid));
    }
    nextqid++;
    c->cache[c->n] = (Uqid){nextqid, pp, id, t, idx};
    c->n++;
    return (nextqid << 8) | t;
}
```

⟨*function* crumb 87b⟩≡ (100d)

```
static Crumb*
crumb(Gitaux *aux, int n)
{
    if(n < aux->ncrumb)
        return &aux->crumb[aux->ncrumb - n - 1];
    return nil;
}
```

⟨*function* popcrumb 87c⟩≡ (100d)

```
static void
popcrumb(Gitaux *aux)
{
    Crumb *c;

    if(aux->ncrumb > 1){
        c = crumb(aux, 0);
        free(c->name);
        unref(c->obj);
        aux->ncrumb--;
    }
}
```

⟨*function* branchid 87d⟩≡ (100d)

```
static vlong
branchid(Gitaux *aux, char *path)
{
    int i;

    for(i = 0; branches[i]; i++)
        if(strcmp(path, branches[i]) == 0)
            goto found;
```

```
        branches = realloc(branches, sizeof(char *)*(i + 2));
        branches[i] = estrdup(path);
        branches[i + 1] = nil;

    found:
        if(aux){
            if(aux->refpath)
                free(aux->refpath);
            aux->refpath = estrdup(branches[i]);
        }
        return i;
    }
```

⟨*function* obj2dir 88a⟩≡ (100d)

```
    static void
    obj2dir(Dir *d, Crumb *c, Object *o, char *name)
    {
        d->qid = c->qid;
        d->atime = c->mtime;
        d->mtime = c->mtime;
        d->mode = c->mode;
        d->name = estrdup9p(name);
        d->uid = estrdup9p(username);
        d->gid = estrdup9p(groupname);
        d->muid = estrdup9p(username);
        if(o->type == GBlob || o->type == GTag){
            d->qid.type = 0;
            d->mode &= 0777;
            d->length = o->size;
        }

    }
```

⟨*function* rootgen 88b⟩≡ (100d)

```
    static int
    rootgen(int i, Dir *d, void *p)
    {
        Crumb *c;

        c = crumb(p, 0);
        if (i >= nelem(qroot))
            return -1;
        d->mode = 0555 | DMDIR;
        d->name = estrdup9p(qroot[i]);
        d->qid.vers = 0;
        d->qid.type = strcmp(qroot[i], "ctl") == 0 ? 0 : QTDIR;
        d->qid.path = qpath(nil, i, i, Qroot);
        d->uid = estrdup9p(username);
        d->gid = estrdup9p(groupname);
        d->muid = estrdup9p(username);
        d->mtime = c->mtime;
        return 0;
    }
```

⟨*function* branchgen 88c⟩≡ (100d)

```
    static int
    branchgen(int i, Dir *d, void *p)
    {
        Gitaux *aux;
        Dir *refs;
```

```
    Crumb *c;
    int n;

    aux = p;
    c = crumb(aux, 0);
    refs = nil;
    d->qid.vers = 0;
    d->qid.type = QTDIR;
    d->qid.path = qpath(c, i, branchid(aux, aux->refpath), Qbranch | Internal);
    d->mode = 0555 | DMDIR;
    d->uid = estrdup9p(username);
    d->gid = estrdup9p(groupname);
    d->muid = estrdup9p(username);
    d->mtime = c->mtime;
    d->atime = c->mtime;
    if((n = slurpdir(aux->refpath, &refs)) < 0)
        return -1;
    if(i < n){
        d->name = estrdup9p(refs[i].name);
        free(refs);
        return 0;
    }else{
        free(refs);
        return -1;
    }
}
```

⟨*function* gtreegen 89⟩≡ (100d)

```
static int
gtreegen(int i, Dir *d, void *p)
{
    Object *o, *l, *e;
    Gitaux *aux;
    Crumb *c;
    int m;

    aux = p;
    c = crumb(aux, 0);
    e = c->obj;
    if(i >= e->tree->nent)
        return -1;
    m = e->tree->ent[i].mode;
    if(e->tree->ent[i].ismod)
        o = emptydir();
    else if((o = readobject(e->tree->ent[i].h)) == nil)
        sysfatal("could not read object %H: %r", e->tree->ent[i].h);
    if(e->tree->ent[i].islink)
        if((l = walklink(aux, o->data, o->size, 0, &m)) != nil)
            o = l;
    d->qid.vers = 0;
    d->qid.type = o->type == GTree ? QTDIR : 0;
    d->qid.path = qpath(c, i, o->id, aux->qdir);
    d->mode = m;
    d->atime = c->mtime;
    d->mtime = c->mtime;
    d->uid = estrdup9p(username);
    d->gid = estrdup9p(groupname);
    d->muid = estrdup9p(username);
    d->name = estrdup9p(e->tree->ent[i].name);
    d->length = o->size;
```

```
    return 0;
}
```

⟨*function* gcommitgen 90a⟩≡ (100d)

```
static int
gcommitgen(int i, Dir *d, void *p)
{
    Object *o;
    Crumb *c;

    c = crumb(p, 0);
    o = c->obj;
    d->uid = estrdup9p(username);
    d->gid = estrdup9p(groupname);
    d->muid = estrdup9p(username);
    d->mode = 0444;
    d->atime = o->commit->ctime;
    d->mtime = o->commit->ctime;
    d->qid.type = 0;
    d->qid.vers = 0;

    switch(i){
    case 0:
        d->mode = DMDIR | gitdirmode;
        d->name = estrdup9p("tree");
        d->qid.type = QTDIR;
        d->qid.path = qpath(c, i, o->id, Qtree);
        break;
    case 1:
        d->name = estrdup9p("parent");
        d->qid.path = qpath(c, i, o->id, Qparent);
        break;
    case 2:
        d->name = estrdup9p("msg");
        d->qid.path = qpath(c, i, o->id, Qmsg);
        break;
    case 3:
        d->name = estrdup9p("hash");
        d->qid.path = qpath(c, i, o->id, Qhash);
        break;
    case 4:
        d->name = estrdup9p("author");
        d->qid.path = qpath(c, i, o->id, Qauthor);
        break;
    default:
        free(d->uid);
        free(d->gid);
        free(d->muid);
        return -1;
    }
    return 0;
}
```

⟨*function* objgen 90b⟩≡ (100d)

```
static int
objgen(int i, Dir *d, void *p)
{
    Gitaux *aux;
    Object *o;
    Crumb *c;
```

```
    char name[64];
    Objlist *ols;
    Hash h;

    aux = p;
    c = crumb(aux, 0);
    if(!aux->ols)
        aux->ols = mkols();
    ols = aux->ols;
    o = nil;
    /* We tried to sent it, but it didn't fit */
    if(aux->olslast && ols->idx == i + 1){
        snprint(name, sizeof(name), "%H", aux->olslast->hash);
        obj2dir(d, c, aux->olslast, name);
        return 0;
    }
    while(ols->idx <= i){
        if(olsnext(ols, &h) == -1)
            return -1;
        if((o = readobject(h)) == nil){
            fprint(2, "corrupt object %H\n", h);
            return -1;
        }
    }
    if(o != nil){
        snprint(name, sizeof(name), "%H", o->hash);
        obj2dir(d, c, o, name);
        unref(aux->olslast);
        aux->olslast = ref(o);
        return 0;
    }
    return -1;
}
```

⟨*function* objread 91a⟩≡ (100d)

```
static void
objread(Req *r, Gitaux *aux)
{
    Object *o;

    o = crumb(aux, 0)->obj;
    switch(o->type){
    case GBlob:
        readbuf(r, o->data, o->size);
        break;
    case GTag:
        readbuf(r, o->data, o->size);
        break;
    case GTree:
        dirread9p(r, gtreegen, aux);
        break;
    case GCommit:
        dirread9p(r, gcommitgen, aux);
        break;
    default:
        sysfatal("invalid object type %d", o->type);
    }
}
```

⟨*function* readcommitparent 91b⟩≡ (100d)

```
static void
readcommitparent(Req *r, Object *o)
{
    char *buf, *p, *e;
    int i, n;

    /* 40 bytes per hash, 1 per nl, 1 for terminator */
    n = o->commit->nparent * (40 + 1) + 1;
    buf = emalloc(n);
    p = buf;
    e = buf + n;
    for (i = 0; i < o->commit->nparent; i++)
        p = seprint(p, e, "%H\n", o->commit->parent[i]);
    readbuf(r, buf, p - buf);
    free(buf);
}
```

⟨*function* gitattach 92a⟩≡ (100d)

```
static void
gitattach(Req *r)
{
    Gitaux *aux;
    Dir *d;

    if((d = dirstat(".git")) == nil)
        sysfatal("git/fs: %r");
    if(getwd(gitdir, sizeof(gitdir)) == nil)
        sysfatal("getwd: %r");
    aux = emalloc(sizeof(Gitaux));
    aux->crumb = emalloc(sizeof(Crumb));
    aux->crumb[0].qid = (Qid){Qroot, 0, QTDIR};
    aux->crumb[0].obj = nil;
    aux->crumb[0].mode = DMDIR | 0555;
    aux->crumb[0].mtime = d->mtime;
    aux->crumb[0].name = estrdup("/");
    aux->ncrumb = 1;
    r->ofcall.qid = (Qid){Qroot, 0, QTDIR};
    r->fid->qid = r->ofcall.qid;
    r->fid->aux = aux;
    respond(r, nil);
}
```

⟨*function* walklink 92b⟩≡ (100d)

```
static Object*
walklink(Gitaux *aux, char *link, int nlink, int ndotdot, int *mode)
{
    char *p, *e, *path;
    Object *o, *n;
    int i;

    path = emalloc(nlink + 1);
    memcpy(path, link, nlink);
    cleanname(path);

    o = crumb(aux, ndotdot)->obj;
    assert(o->type == GTree);
    for(p = path; *p; p = e){
        n = nil;
        e = p + strcspn(p, "/");
        if(*e == '/')
```

```
            *e++ = '\0';
        /*
         * cleanname guarantees these show up at the start of the name,
         * which allows trimming them from the end of the trail of crumbs
         * instead of needing to keep track of full parentage.
         */
        if(strcmp(p, "..") == 0)
            n = crumb(aux, ++ndotdot)->obj;
        else if(o->type == GTree)
            for(i = 0; i < o->tree->nent; i++)
                if(strcmp(o->tree->ent[i].name, p) == 0){
                    *mode = o->tree->ent[i].mode;
                    n = readobject(o->tree->ent[i].h);
                    break;
                }
        o = n;
        if(o == nil)
            break;
    }
    free(path);
    for(i = 0; o != nil && i < aux->ncrumb; i++)
        if(crumb(aux, i)->obj == o)
            return nil;
    return o;
}
```

⟨*function* objwalk1 93⟩≡ (100d)

```
static char *
objwalk1(Qid *q, Object *o, Crumb *p, Crumb *c, char *name, vlong qdir, Gitaux *aux)
{
    Object *w, *l;
    char *e;
    int i, m;

    w = nil;
    e = nil;
    if(!o)
        return Eexist;
    switch(o->type){
    case GBlob:
        e = Enotdir;
        break;
    case GTree:
        q->type = 0;
        for(i = 0; i < o->tree->nent; i++){
            if(strcmp(o->tree->ent[i].name, name) != 0)
                continue;
            m = o->tree->ent[i].mode;
            w = readobject(o->tree->ent[i].h);
            if(!w && o->tree->ent[i].ismod)
                w = emptydir();
            if(w && o->tree->ent[i].islink)
                if((l = walklink(aux, w->data, w->size, 1, &m)) != nil)
                    w = l;
            if(!w)
                return Ebadobj;
            q->type = (w->type == GTree) ? QTDIR : 0;
            q->path = qpath(p, i, w->id, qdir);
            c->mode = m;
            c->mode |= (w->type == GTree) ? (DMDIR|0755) : 0644;
```

```
            c->obj = w;
            break;
        }
        if(!w)
            e = Eexist;
        break;
    case  GCommit:
        q->type = 0;
        c->mtime = o->commit->mtime;
        c->mode = 0644;
        assert(qdir == Qcommit || qdir == Qobject || qdir == Qtree || qdir == Qhead || qdir == Qcommitter);
        if(strcmp(name, "msg") == 0)
            q->path = qpath(p, 0, o->id, Qmsg);
        else if(strcmp(name, "parent") == 0)
            q->path = qpath(p, 1, o->id, Qparent);
        else if(strcmp(name, "hash") == 0)
            q->path = qpath(p, 2, o->id, Qhash);
        else if(strcmp(name, "author") == 0)
            q->path = qpath(p, 3, o->id, Qauthor);
        else if(strcmp(name, "committer") == 0)
            q->path = qpath(p, 3, o->id, Qcommitter);
        else if(strcmp(name, "tree") == 0){
            q->type = QTDIR;
            q->path = qpath(p, 4, o->id, Qtree);
            unref(c->obj);
            c->mode = DMDIR | gitdirmode;
            c->obj = readobject(o->commit->tree);
            if(c->obj == nil)
                sysfatal("could not read object %H: %r", o->commit->tree);
        }
        else
            e = Eexist;
        break;
    case GTag:
        e = Eimpl;
        break;
    }
    return e;
}
```

⟨*function* readref 94⟩≡ (100d)

```
static Object *
readref(char *pathstr)
{
    char buf[128], path[128], *p, *e;
    Hash h;
    int n, f;

    snprint(path, sizeof(path), "%s", pathstr);
    while(1){
        if((f = open(path, OREAD)) == -1)
            return nil;
        if((n = readn(f, buf, sizeof(buf) - 1)) == -1)
            return nil;
        close(f);
        buf[n] = 0;
        if(strncmp(buf, "ref:", 4) !=  0)
            break;

        p = buf + 4;
```

```
        while(isspace(*p))
            p++;
        if((e = strchr(p, '\n')) != nil)
            *e = 0;
        snprint(path, sizeof(path), ".git/%s", p);
    }

    if(hparse(&h, buf) == -1)
        return nil;

    return readobject(h);
}
```

⟨*function* gitwalk1 95⟩≡ (100d)

```
static char*
gitwalk1(Fid *fid, char *name, Qid *q)
{
    char path[128];
    Gitaux *aux;
    Crumb *c, *o;
    char *e;
    Dir *d;
    Hash h;

    e = nil;
    aux = fid->aux;

    q->vers = 0;
    if(strcmp(name, "..") == 0){
        popcrumb(aux);
        c = crumb(aux, 0);
        *q = c->qid;
        fid->qid = *q;
        return nil;
    }

    aux->crumb = realloc(aux->crumb, (aux->ncrumb + 1) * sizeof(Crumb));
    aux->ncrumb++;
    c = crumb(aux, 0);
    o = crumb(aux, 1);
    memset(c, 0, sizeof(Crumb));
    c->mode = o->mode;
    c->mtime = o->mtime;
        c->obj = o->obj ? ref(o->obj) : nil;

    switch(QDIR(&fid->qid)){
    case Qroot:
        if(strcmp(name, "HEAD") == 0){
            *q = (Qid){Qhead, 0, QTDIR};
            c->mode = DMDIR | 0555;
            c->obj = readref(".git/HEAD");
        }else if(strcmp(name, "object") == 0){
            *q = (Qid){Qobject, 0, QTDIR};
            c->mode = DMDIR | 0555;
        }else if(strcmp(name, "branch") == 0){
            *q = (Qid){Qbranch, 0, QTDIR};
            aux->refpath = estrdup(".git/refs/");
            c->mode = DMDIR | 0555;
        }else if(strcmp(name, "ctl") == 0){
            *q = (Qid){Qctl, 0, 0};
```

```
        c->mode = 0644;
    }else{
        e = Eexist;
    }
    break;
case Qbranch:
    if(strcmp(aux->refpath, ".git/refs/heads") == 0 && strcmp(name, "HEAD") == 0)
        snprint(path, sizeof(path), ".git/HEAD");
    else
        snprint(path, sizeof(path), "%s/%s", aux->refpath, name);
    q->type = QTDIR;
    d = dirstat(path);
    if(d && d->qid.type == QTDIR)
        q->path = qpath(o, Qbranch, branchid(aux, path), Qbranch);
    else if(d && (c->obj = readref(path)) != nil)
        q->path = qpath(o, Qbranch, c->obj->id, Qcommit);
    else
        e = Eexist;
    if(d != nil)
        c->mode = d->mode & ~0222;
    free(d);
    break;
case Qobject:
    if(c->obj){
        e = objwalk1(q, o->obj, o, c, name, Qobject, aux);
    }else{
        if(hparse(&h, name) == -1)
            return Ebadobj;
        if((c->obj = readobject(h)) == nil)
            return Ebadobj;
        if(c->obj->type == GBlob || c->obj->type == GTag){
            c->mode = 0644;
            q->type = 0;
        }else{
            c->mode = DMDIR | 0755;
            q->type = QTDIR;
        }
        q->path = qpath(o, Qobject, c->obj->id, Qobject);
        q->vers = 0;
    }
    break;
case Qhead:
    e = objwalk1(q, o->obj, o, c, name, Qhead, aux);
    break;
case Qcommit:
    e = objwalk1(q, o->obj, o, c, name, Qcommit, aux);
    break;
case Qtree:
    e = objwalk1(q, o->obj, o, c, name, Qtree, aux);
    break;
case Qparent:
case Qmsg:
case Qcdata:
case Qhash:
case Qauthor:
case Qcommitter:
case Qctl:
    return Enodir;
default:
    return Egreg;
```

```
    }

    c->name = estrdup(name);
    c->qid = *q;
    fid->qid = *q;
    return e;
}
```

⟨*function* gitclone 97a⟩≡ (100d)

```
static char*
gitclone(Fid *o, Fid *n)
{
    Gitaux *aux, *oaux;
    int i;

    oaux = o->aux;
    aux = emalloc(sizeof(Gitaux));
    aux->ncrumb = oaux->ncrumb;
    aux->crumb = eamalloc(oaux->ncrumb, sizeof(Crumb));
    for(i = 0; i < aux->ncrumb; i++){
        aux->crumb[i] = oaux->crumb[i];
        aux->crumb[i].name = estrdup(oaux->crumb[i].name);
        if(aux->crumb[i].obj)
            aux->crumb[i].obj = ref(oaux->crumb[i].obj);
    }
    if(oaux->refpath)
        aux->refpath = strdup(oaux->refpath);
    aux->qdir = oaux->qdir;
    n->aux = aux;
    return nil;
}
```

⟨*function* gitdestroyfid 97b⟩≡ (100d)

```
static void
gitdestroyfid(Fid *f)
{
    Gitaux *aux;
    int i;

    if((aux = f->aux) == nil)
        return;
    for(i = 0; i < aux->ncrumb; i++){
        if(aux->crumb[i].obj)
            unref(aux->crumb[i].obj);
        free(aux->crumb[i].name);
    }
    olsfree(aux->ols);
    free(aux->refpath);
    free(aux->crumb);
    free(aux);
}
```

⟨*function* readctl 97c⟩≡ (100d)

```
static char *
readctl(Req *r)
{
    char data[1024], ref[512], *s, *e;
    int fd, n;

    if((fd = open(".git/HEAD", OREAD)) == -1)
```

```
        return Erepo;
    /* empty HEAD is invalid */
    if((n = readn(fd, ref, sizeof(ref) - 1)) <= 0)
        return Erepo;
    close(fd);

    s = ref;
    ref[n] = 0;
    if(strncmp(s, "ref:", 4) == 0)
        s += 4;
    while(*s == ' ' || *s == '\t')
        s++;
    if((e = strchr(s, '\n')) != nil)
        *e = 0;
    if(strstr(s, "refs/") == s)
        s += strlen("refs/");

    snprint(data, sizeof(data), "branch %s\nrepo %s\n", s, gitdir);
    readstr(r, data);
    return nil;
}
```

⟨*function* gitread 98⟩≡ (100d)

```
static void
gitread(Req *r)
{
    char buf[256], *e;
    Gitaux *aux;
    Object *o;
    Qid *q;

    aux = r->fid->aux;
    q = &r->fid->qid;
    o = crumb(aux, 0)->obj;
    e = nil;

    switch(QDIR(q)){
    case Qroot:
        dirread9p(r, rootgen, aux);
        break;
    case Qbranch:
        if(o)
            objread(r, aux);
        else
            dirread9p(r, branchgen, aux);
        break;
    case Qobject:
        if(o)
            objread(r, aux);
        else
            dirread9p(r, objgen, aux);
        break;
    case Qmsg:
        readbuf(r, o->commit->msg, o->commit->nmsg);
        break;
    case Qparent:
        readcommitparent(r, o);
        break;
    case Qhash:
        snprint(buf, sizeof(buf), "%H\n", o->hash);
```

```
        readstr(r, buf);
        break;
    case Qauthor:
        snprint(buf, sizeof(buf), "%s\n", o->commit->author);
        readstr(r, buf);
        break;
    case Qcommitter:
        snprint(buf, sizeof(buf), "%s\n", o->commit->committer);
        readstr(r, buf);
        break;
    case Qctl:
        e = readctl(r);
        break;
    case Qhead:
        /* Empty repositories have no HEAD */
        if(o == nil)
            r->ofcall.count = 0;
        else
            objread(r, aux);
        break;
    case Qcommit:
    case Qtree:
    case Qcdata:
        objread(r, aux);
        break;
    default:
        e = Egreg;
    }
    respond(r, e);
}
```

⟨*function* gitopen 99⟩≡ (100d)

```
static void
gitopen(Req *r)
{
    Gitaux *aux;
    Crumb *c;

    aux = r->fid->aux;
    c = crumb(aux, 0);
    switch(r->ifcall.mode&3){
    default:
        respond(r, "botched mode");
        break;
    case OWRITE:
        respond(r, Eperm);
        break;
    case OREAD:
    case ORDWR:
        respond(r, nil);
        break;
    case OEXEC:
        if((c->mode & 0111) == 0)
            respond(r, Eperm);
        else
            respond(r, nil);
        break;
    }
}
```

⟨*function* gitstat 100a⟩≡ (100d)

```
static void
gitstat(Req *r)
{
    Gitaux *aux;
    Crumb *c;

    aux = r->fid->aux;
    c = crumb(aux, 0);
    r->d.uid = estrdup9p(username);
    r->d.gid = estrdup9p(groupname);
    r->d.muid = estrdup9p(username);
    r->d.qid = r->fid->qid;
    r->d.mtime = c->mtime;
    r->d.atime = c->mtime;
    r->d.mode = c->mode;
    if(c->obj)
        obj2dir(&r->d, c, c->obj, c->name);
    else
        r->d.name = estrdup9p(c->name);
    respond(r, nil);
}
```

⟨*function* usage (git9/fs.c) 100b⟩≡ (100d)

```
void
usage(void)
{
    fprint(2, "usage: %s [-d]\n", argv0);
    fprint(2, "\t-d:    debug\n");
    exits("usage");
}
```

⟨*function* main (git9/fs.c) 100c⟩≡ (100d)

```
void
main(int argc, char **argv)
{
    char repo[512];
    int nelt;
    Dir *d;

    gitinit(repo, sizeof(repo), &nelt);
    if(chdir(repo) == -1)
        sysfatal("chdir: %r");

    ARGBEGIN{
    case 'd':
        chatty9p++;
        break;
    case 'm':
        mntpt = EARGF(usage());
        break;
    default:
        usage();
        break;
    }ARGEND;
```

⟨git9/fs.c 100d⟩≡

```
#include <u.h>
#include <libc.h>
#include <ctype.h>
```

```
#include <fcall.h>
#include <thread.h>
#include <9p.h>

#include "git.h"

enum {
    Qroot,
    Qhead,
    Qbranch,
    Qcommit,
        Qmsg,
        Qparent,
        Qtree,
        Qcdata,
        Qhash,
        Qauthor,
        Qcommitter,
    Qobject,
    Qctl,
    Qmax,
    Internal=1<<7,
};

typedef struct Gitaux Gitaux;
typedef struct Crumb Crumb;
typedef struct Cache Cache;
typedef struct Uqid Uqid;
struct Crumb {
    char    *name;
    Object  *obj;
    Qid qid;
    int mode;
    vlong   mtime;
};

struct Gitaux {
    int ncrumb;
    Crumb   *crumb;
    char    *refpath;
    int qdir;

    /* For listing object dir */
    Objlist *ols;
    Object  *olslast;
};

struct Uqid {
    vlong   uqid;

    vlong   ppath;
    vlong   oid;
    int t;
    int idx;
};

struct Cache {
    Uqid *cache;
    int n;
    int max;
```

```
};

char *qroot[] = {
    "HEAD",
    "branch",
    "object",
    "ctl",
};
```

⟨*constant* Eperm 86a⟩
⟨*constant* Eexist 86b⟩
⟨*constant* Enotdir 86c⟩
⟨*constant* E2long 86d⟩
⟨*constant* Enodir 86e⟩
⟨*constant* Erepo 86f⟩
⟨*constant* Eimpl 86g⟩
⟨*constant* Egreg 86h⟩
⟨*constant* Ebadobj 86i⟩

```
char    gitdir[512];
char    *username;
char    *groupname;
char    *mntpt = ".git/fs";
char    **branches = nil;
Cache   uqidcache[512];
vlong   nextqid = Qmax;

static Object*  walklink(Gitaux *, char *, int, int, int*);
```

⟨*function* qpath 87a⟩

⟨*function* crumb 87b⟩

⟨*function* popcrumb 87c⟩

⟨*function* branchid 87d⟩

⟨*function* obj2dir 88a⟩

⟨*function* rootgen 88b⟩

⟨*function* branchgen 88c⟩

⟨*function* gtreegen 89⟩

⟨*function* gcommitgen 90a⟩

⟨*function* objgen 90b⟩

⟨*function* objread 91a⟩

⟨*function* readcommitparent 91b⟩

⟨*function* gitattach 92a⟩

⟨*function* walklink 92b⟩

⟨*function* objwalk1 93⟩

⟨*function* readref 94⟩

⟨*function* gitwalk1 95⟩

⟨*function* gitclone 97a⟩

⟨*function* gitdestroyfid 97b⟩

⟨*function* readctl 97c⟩

⟨*function* gitread 98⟩

⟨*function* gitopen 99⟩

⟨*function* gitstat 100a⟩

```
Srv gitsrv = {
    .attach=gitattach,
    .walk1=gitwalk1,
    .clone=gitclone,
    .open=gitopen,
    .read=gitread,
    .stat=gitstat,
    .destroyfid=gitdestroyfid,
};
```

⟨*function* usage (git9/fs.c) 100b⟩

⟨*function* main (git9/fs.c) 100c⟩

```
    if(argc != 0)
        usage();

    if((d = dirstat(".git")) == nil)
        sysfatal("dirstat .git: %r");
    username = strdup(d->uid);
    groupname = strdup(d->gid);
    free(d);

    branches = emalloc(sizeof(char*));
    branches[0] = nil;
    postmountsrv(&gitsrv, nil, mntpt, MCREATE);
    exits(nil);
}
```

git9/get.c

⟨*function* okrefname 103⟩≡ (112)

```
/*
 * Checks the rules for a refname at
 * git/Documentation/protocol-common.txt
 */
int
okrefname(char *s)
{
    if(strcmp(s, "HEAD") == 0)
        return 1;
    if(strncmp(s, "refs/", 5) == 0)
        return okref(s);
    return 0;
```

```
}
```

⟨*function* `resolveremote` 104a⟩≡ (112)

```
int
resolveremote(Hash *h, char *ref)
{
    char buf[128], *s;
    int r, f;

    ref = strip(ref);
    if((r = hparse(h, ref)) != -1)
        return r;
    /* Slightly special handling: translate remote refs to local ones. */
    if(strcmp(ref, "HEAD") == 0){
        snprint(buf, sizeof(buf), ".git/HEAD");
    }else if(strstr(ref, "refs/heads") == ref){
        ref += strlen("refs/heads");
        snprint(buf, sizeof(buf), ".git/refs/remotes/%s/%s", upstream, ref);
    }else if(strstr(ref, "refs/tags") == ref){
        ref += strlen("refs/tags");
        snprint(buf, sizeof(buf), ".git/refs/tags/%s/%s", upstream, ref);
    }else{
        return -1;
    }

    r = -1;
    s = strip(buf);
    if((f = open(s, OREAD)) == -1)
        return -1;
    if(readn(f, buf, sizeof(buf)) >= 40)
        r = hparse(h, buf);
    close(f);

    if(r == -1 && strstr(buf, "ref:") == buf)
        return resolveremote(h, buf + strlen("ref:"));
    return r;
}
```

⟨*function* `rename` 104b⟩≡ (112)

```
int
rename(char *pack, char *idx, Hash h)
{
    char name[128];
    Dir st;

    nulldir(&st);
    st.name = name;
    snprint(name, sizeof(name), "%H.pack", h);
    if(access(name, AEXIST) == 0)
        fprint(2, "warning, pack %s already fetched\n", name);
    else if(dirwstat(pack, &st) == -1)
        return -1;
    snprint(name, sizeof(name), "%H.idx", h);
    if(access(name, AEXIST) == 0)
        fprint(2, "warning, pack %s already indexed\n", name);
    else if(dirwstat(idx, &st) == -1)
        return -1;
    return 0;
}
```

⟨*function* checkhash 105a⟩≡ (112)

```
int
checkhash(int fd, vlong sz, Hash *hcomp)
{
    DigestState *st;
    Hash hexpect;
    char buf[Pktmax];
    vlong n, r;
    int nr;

    if(sz < 28){
        werrstr("undersize packfile");
        return -1;
    }

    st = nil;
    n = 0;
    while(n != sz - 20){
        nr = sizeof(buf);
        if(sz - n - 20 < sizeof(buf))
            nr = sz - n - 20;
        r = readn(fd, buf, nr);
        if(r != nr)
            return -1;
        st = sha1((uchar*)buf, nr, nil, st);
        n += r;
    }
    sha1(nil, 0, hcomp->h, st);
    if(readn(fd, hexpect.h, sizeof(hexpect.h)) != sizeof(hexpect.h))
        sysfatal("truncated packfile");
    if(!hasheq(hcomp, &hexpect)){
        werrstr("bad hash: %H != %H", *hcomp, hexpect);
        return -1;
    }
    return 0;
}
```

⟨*function* mkoutpath 105b⟩≡ (112)

```
int
mkoutpath(char *path)
{
    char s[128];
    char *p;
    int fd;

    snprint(s, sizeof(s), "%s", path);
    for(p=strchr(s+1, '/'); p; p=strchr(p+1, '/')){
        *p = 0;
        if(access(s, AEXIST) != 0){
            fd = create(s, OREAD, DMDIR | 0775);
            if(fd == -1)
                return -1;
            close(fd);
        }
        *p = '/';
    }
    return 0;
}
```

⟨*function* prefixed 105c⟩≡ (112)

```
int
prefixed(char *s, char *pfx)
{
    return strncmp(s, pfx, strlen(pfx)) == 0;
}
```

⟨*function* `branchmatch` 106a⟩≡ (112)

```
int
branchmatch(char *br, char *pat)
{
    char name[128];

    if(prefixed(pat, "refs/heads"))
        snprint(name, sizeof(name), "%s", pat);
    else if(prefixed(pat, "heads/"))
        snprint(name, sizeof(name), "refs/%s", pat);
    else
        snprint(name, sizeof(name), "refs/heads/%s", pat);
    return strcmp(br, name) == 0;
}
```

⟨*function* `fail` 106b⟩≡ (112)

```
void
fail(char *pack, char *idx, char *msg, ...)
{
    char buf[ERRMAX];
    va_list ap;

    va_start(ap, msg);
    snprint(buf, sizeof(buf), msg, ap);
    va_end(ap);

    remove(pack);
    remove(idx);
    fprint(2, "%s", buf);
    exits(buf);
}
```

⟨*function* `enqueueparent` 106c⟩≡ (112)

```
void
enqueueparent(Objq *q, Object *o)
{
    Object *p;
    int i;

    if(o->type != GCommit)
        return;
    for(i = 0; i < o->commit->nparent; i++){
        if((p = readobject(o->commit->parent[i])) == nil)
            continue;
        qput(q, p, 0);
        unref(p);
    }
}
```

⟨*function* `fmtcaps` 106d⟩≡ (112)

```
void
fmtcaps(Conn *c, char *caps, int ncaps)
{
    char *p, *e;
```

```
    p = caps;
    e = caps + ncaps;
    *p = 0;
    if(c->multiack)
        p = seprint(p, e, " multi_ack");
    if(c->sideband64k)
        p = seprint(p, e, " side-band-64k");
    else if(c->sideband)
        p = seprint(p, e, " side-band");
    assert(p != e);
}
```

⟨*function* sbread 107a⟩≡ (112)

```
int
sbread(Conn *c, char *buf, int nbuf, char **pbuf)
{
    int n;

    assert(nbuf >= Pktmax);
    if(!c->sideband && !c->sideband64k){
        *pbuf = buf;
        return readn(c->rfd, buf, nbuf);
    }else{
        *pbuf = buf+1;
        while(1){
            n = readpkt(c, buf, nbuf);
            if(n <= 0)
                return n;
            else if(buf[0] == 1 && n > 1)
                return n - 1;
            else if(buf[0] == 3)
                fprint(2, "error: %s\n", buf+1);
            else if(buf[0] < 1 || buf[0] > 3)
                fprint(2, "unknown sideband(%c:%d) data: %s\n", buf[0], buf[0], buf+1);

        }
    }
}
```

⟨*function* fetchpack 107b⟩≡ (112)

```
int
fetchpack(Conn *c)
{
    char spinner[] = {'|', '/', '-', '\\'};
    char buf[Pktmax], caps[512], *sp[3], *ep;
    char *packtmp, *idxtmp, **ref, *rp;
    Hash h, *have, *want;
    int nref, refsz, first, nsent;
    int i, j, l, n, spin, req, pfd;
    vlong packsz;
    Objset hadobj;
    Object *o;
    Objq haveq;
    Qelt e;

    nref = 0;
    refsz = 16;
    first = 1;
    have = eamalloc(refsz, sizeof(have[0]));
```

```
want = eamalloc(refsz, sizeof(want[0]));
ref = eamalloc(refsz, sizeof(ref[0]));
while(1){
    n = readpkt(c, buf, sizeof(buf));
    if(n == -1)
        return -1;
    if(n == 0)
        break;

    if(first && n > strlen(buf)){
        parsecaps(buf + strlen(buf) + 1, c);
        if(c->symfrom[0] != 0)
            print("symref %s %s\n", c->symfrom, c->symto);
    }
    first = 0;

    if(getfields(buf, sp, nelem(sp), 1, " \t\n\r") < 2)
        sysfatal("invalid ref line");
    if(strstr(sp[1], "^{}"))
        continue;
    if(!okrefname(sp[1]))
        sysfatal("remote side sent invalid ref: %s", sp[1]);
    if(fetchbranch && !branchmatch(sp[1], fetchbranch))
        continue;
    else if(strcmp(sp[1], "HEAD") != 0
    && !prefixed(sp[1], "refs/heads/")
    && !prefixed(sp[1], "refs/tags/"))
        continue;
    if(refsz == nref + 1){
        refsz *= 2;
        have = earealloc(have, refsz, sizeof(have[0]));
        want = earealloc(want, refsz, sizeof(want[0]));
        ref = earealloc(ref, refsz, sizeof(ref[0]));
    }
    if(hparse(&want[nref], sp[0]) == -1)
        sysfatal("invalid hash %s", sp[0]);
    if (resolveremote(&have[nref], sp[1]) == -1)
        memset(&have[nref], 0, sizeof(have[nref]));
    ref[nref] = estrdup(sp[1]);
    nref++;
}
if(listonly){
    flushpkt(c);
    goto showrefs;
}

if(writephase(c) == -1)
    sysfatal("write: %r");
req = 0;
fmtcaps(c, caps, sizeof(caps));
for(i = 0; i < nref; i++){
    if(hasheq(&have[i], &want[i]))
        goto skip;
    for(j = 0; j < i; j++)
        if(hasheq(&want[i], &want[j]))
            goto skip;
    if((o = readobject(want[i])) != nil){
        unref(o);
        continue;
    }
```

```
        if(fmtpkt(c, "want %H%s\n", want[i], caps) == -1)
            sysfatal("could not send want for %H", want[i]);
        caps[0] = 0;
        req = 1;
skip:;
    }
    flushpkt(c);

    nsent = 0;
    qinit(&haveq);
    osinit(&hadobj);
    /*
     * We know we have these objects, and we want to make sure that
     * they end up at the front of the queue. Send the 'have lines'
     * first, and then enqueue their parents for a second round of
     * sends.
     */
    for(i = 0; i < nref; i++){
        if(hasheq(&have[i], &Zhash) || oshas(&hadobj, have[i]))
            continue;
        if((o = readobject(have[i])) == nil)
            sysfatal("missing exected object: %H", have[i]);
        if(fmtpkt(c, "have %H", o->hash) == -1)
            sysfatal("write: %r");
        enqueueparent(&haveq, o);
        osadd(&hadobj, o);
        unref(o);
        nsent++;
    }
    /*
     * The other branches we have probably make sense to send,
     * since often we'll be pulling a new branch with objects
     * that we already have; it's not entirely clear what we
     * want to do here.
     */
    for(i = 0; i < nheads; i++){
        if((o = readobject(heads[i])) == nil)
            sysfatal("missing exected object: %H", have[i]);
        if(fmtpkt(c, "have %H", o->hash) == -1)
            sysfatal("write: %r");
        enqueueparent(&haveq, o);
        osadd(&hadobj, o);
        unref(o);
        nsent++;
    }
    /*
     * While we could short circuit this and check if upstream has
     * acked our objects, for the first 256 haves, this is simple
     * enough.
     *
     * Also, doing multiple rounds of reference discovery breaks
     * when using smart http.
     */
    while(req && qpop(&haveq, &e) && nsent < 256){
        if(oshas(&hadobj, e.o->hash))
            continue;
        if((o = readobject(e.o->hash)) == nil)
            sysfatal("missing object we should have: %H", e.o->hash);
        if(fmtpkt(c, "have %H", o->hash) == -1)
            sysfatal("write: %r");
```

```
    enqueueparent(&haveq, o);
    osadd(&hadobj, o);
    unref(o);
    nsent++;
}
osclear(&hadobj);
qclear(&haveq);
if(!req)
    flushpkt(c);
if(fmtpkt(c, "done\n") == -1)
    sysfatal("write: %r");
if(!req)
    goto showrefs;
if(readphase(c) == -1)
    sysfatal("read: %r");

if((packtmp = smprint(".git/objects/pack/fetch.%d.pack", getpid())) == nil)
    sysfatal("smprint: %r");
if((idxtmp = smprint(".git/objects/pack/fetch.%d.idx", getpid())) == nil)
    sysfatal("smprint: %r");
if(mkoutpath(packtmp) == -1)
    sysfatal("could not create %s: %r", packtmp);
if((pfd = create(packtmp, ORDWR, 0664)) == -1)
    sysfatal("could not create %s: %r", packtmp);

fprint(2, "fetching...  ");
packsz = 0;
if(c->multiack){
    for(i = 0; i < nsent; i++){
        if(readpkt(c, buf, sizeof(buf)) == -1)
            sysfatal("read: %r");
        if(strncmp(buf, "NAK\n", 4) == 0)
            break;
        if(strncmp(buf, "ACK ", 4) == 0){
            if(getfields(buf, sp, nelem(sp), 1, " \t") == 2)
                break;
            continue;
        }
        sysfatal("bad response: '%s'", buf);
    }
}
if(readpkt(c, buf, sizeof(buf)) == -1)
    sysfatal("read: %r");
if(!c->sideband && !c->sideband64k && !c->multiack){
    /*
     * Work around torvalds git bug: we get duplicate have lines
     * somtimes, even though the protocol is supposed to start the
     * pack file immediately.
     *
     * Skip ahead until we read 'PACK' off the wire
     */
    while(1){
        if(readn(c->rfd, buf, 4) != 4)
            sysfatal("fetch packfile: short read");
        if(strncmp(buf, "PACK", 4) == 0)
            break;
        buf[4] = 0;
        l = strtol(buf, &ep, 16);
        if(ep != buf + 4)
            sysfatal("fetch packfile: junk pktline");
```

```
            if(readn(c->rfd, buf, l-4) != l-4)
                sysfatal("fetch packfile: short read");
        }
        if(write(pfd, "PACK", 4) != 4)
            sysfatal("write pack header: %r");
        packsz = 4;
    }
    spin = 0;
    while(1){
        n = sbread(c, buf, sizeof buf, &rp);
        if(n == 0)
            break;
        if(n == -1 || write(pfd, rp, n) != n)
            sysfatal("fetch packfile: %r");
        if(interactive && spin++ % 100 == 0)
            fprint(2, "\b%c", spinner[spin/100 % nelem(spinner)]);
        packsz += n;
    }
    fprint(2, "\n");
    closeconn(c);
    if(seek(pfd, 0, 0) == -1)
        fail(packtmp, idxtmp, "packfile seek: %r");
    if(checkhash(pfd, packsz, &h) == -1)
        fail(packtmp, idxtmp, "corrupt packfile: %r");
    close(pfd);
    if(indexpack(packtmp, idxtmp, h) == -1)
        fail(packtmp, idxtmp, "could not index fetched pack: %r");
    if(rename(packtmp, idxtmp, h) == -1)
        fail(packtmp, idxtmp, "could not rename indexed pack: %r");

showrefs:
    for(i = 0; i < nref; i++){
        print("remote %s %H local %H\n", ref[i], want[i], have[i]);
        free(ref[i]);
    }
    free(ref);
    free(want);
    free(have);
    return 0;
}
```

⟨*function* usage (git9/get.c) 111a⟩≡ (112)

```
void
usage(void)
{
    fprint(2, "usage: %s [-dl] [-b br] [-u upstream] remote\n", argv0);
    fprint(2, "\t-b br: only fetch matching branch 'br'\n");
    fprint(2, "remote:  fetch from this repository\n");
    exits("usage");
}
```

⟨*function* main (git9/get.c) 111b⟩≡ (112)

```
void
main(int argc, char **argv)
{
    char *s;
    Conn c;

    ARGBEGIN{
    case 'b':   fetchbranch=EARGF(usage()); break;
```

```
    case 'u':   upstream=EARGF(usage());    break;
    case 'd':   chattygit++;            break;
    case 'l':   listonly++;         break;
    case 'h':
        s = EARGF(usage());
        if(nheads < nelem(heads))
            if(hparse(&heads[nheads], s) == 0)
                nheads++;
        break;
    default:
        usage();
        break;
    }ARGEND;
```

⟨git9/get.c 112⟩≡

```
#include <u.h>
#include <libc.h>

#include "git.h"

char *fetchbranch;
char *upstream = "origin";
Hash heads[64];
int nheads;
int listonly;
```

⟨*function* okrefname 103⟩

⟨*function* resolveremote 104a⟩

⟨*function* rename 104b⟩

⟨*function* checkhash 105a⟩

⟨*function* mkoutpath 105b⟩

⟨*function* prefixed 105c⟩

⟨*function* branchmatch 106a⟩

⟨*function* fail 106b⟩

⟨*function* enqueueparent 106c⟩

⟨*function* fmtcaps 106d⟩

⟨*function* sbread 107a⟩

⟨*function* fetchpack 107b⟩

⟨*function* usage (git9/get.c) 111a⟩

⟨*function* main (git9/get.c) 111b⟩

```
    gitinit(nil, 0, nil);
    if(argc != 1)
        usage();

    if(gitconnect(&c, argv[0], "upload") == -1)
        sysfatal("could not dial %s: %r", argv[0]);
```

```
    if(fetchpack(&c) == -1)
        sysfatal("fetch failed: %r");
    closeconn(&c);
    exits(nil);
}
```

git9/git.h

⟨*macro* GETBE16 113a⟩≡ (114d)

```
#define GETBE16(b)\
        ((((b)[0] & 0xFFul) <<  8) | \
         (((b)[1] & 0xFFul) <<  0))
```

⟨*macro* GETBE32 113b⟩≡ (114d)

```
#define GETBE32(b)\
        ((((b)[0] & 0xFFul) << 24) | \
         (((b)[1] & 0xFFul) << 16) | \
         (((b)[2] & 0xFFul) <<  8) | \
         (((b)[3] & 0xFFul) <<  0))
```

⟨*macro* GETBE64 113c⟩≡ (114d)

```
#define GETBE64(b)\
        ((((b)[0] & 0xFFull) << 56) | \
         (((b)[1] & 0xFFull) << 48) | \
         (((b)[2] & 0xFFull) << 40) | \
         (((b)[3] & 0xFFull) << 32) | \
         (((b)[4] & 0xFFull) << 24) | \
         (((b)[5] & 0xFFull) << 16) | \
         (((b)[6] & 0xFFull) <<  8) | \
         (((b)[7] & 0xFFull) <<  0))
```

⟨*macro* PUTBE16 113d⟩≡ (114d)

```
#define PUTBE16(b, n)\
    do{ \
        (b)[0] = (n) >> 8; \
        (b)[1] = (n) >> 0; \
    } while(0)
```

⟨*macro* PUTBE32 113e⟩≡ (114d)

```
#define PUTBE32(b, n)\
    do{ \
        (b)[0] = (n) >> 24; \
        (b)[1] = (n) >> 16; \
        (b)[2] = (n) >> 8; \
        (b)[3] = (n) >> 0; \
    } while(0)
```

⟨*macro* PUTBE64 113f⟩≡ (114d)

```
#define PUTBE64(b, n)\
    do{ \
        (b)[0] = (n) >> 56; \
        (b)[1] = (n) >> 48; \
        (b)[2] = (n) >> 40; \
        (b)[3] = (n) >> 32; \
        (b)[4] = (n) >> 24; \
        (b)[5] = (n) >> 16; \
        (b)[6] = (n) >> 8; \
        (b)[7] = (n) >> 0; \
    } while(0)
```

⟨*macro* QDIR 114a⟩≡ (114d)
```
#define QDIR(qid)   ((int)(qid)->path & (0xff))
```

⟨*macro* isblank 114b⟩≡ (114d)
```
#define isblank(c) \
    (((c) != '\n') && isspace(c))
```

⟨*macro* dprint 114c⟩≡ (114d)
```
#define dprint(lvl, ...) \
    if(chattygit >= lvl) _dprint(__VA_ARGS__)
```

⟨git9/git.h 114d⟩≡
```
#include <bio.h>
#include <mp.h>
#include <libsec.h>
#include <flate.h>
#include <regexp.h>

typedef struct Conn Conn;
typedef struct Hash Hash;
typedef struct Delta    Delta;
typedef struct Cinfo    Cinfo;
typedef struct Tinfo    Tinfo;
typedef struct Object   Object;
typedef struct Objset   Objset;
typedef struct Pack Pack;
typedef struct Buf  Buf;
typedef struct Dirent   Dirent;
typedef struct Idxent   Idxent;
typedef struct Objlist  Objlist;
typedef struct Dtab Dtab;
typedef struct Dblock   Dblock;
typedef struct Objq Objq;
typedef struct Qelt Qelt;
typedef struct Idxent   Idxent;

enum {
    Pathmax     = 512,
    Npackcache  = 32,
    Hashsz      = 20,
    Pktmax      = 65536,
    KiB     = 1024,
    MiB     = 1024*KiB,
};

enum {
    GNone   = 0,
    GCommit = 1,
    GTree   = 2,
    GBlob   = 3,
    GTag    = 4,
    GOdelta = 6,
    GRdelta = 7,
};

enum {
    Cloaded = 1 << 0,
    Cidx    = 1 << 1,
    Ccache  = 1 << 2,
    Cexist  = 1 << 3,
```

```
    Cparsed = 1 << 5,
    Cthin   = 1 << 6,
};

enum {
    ConnGit,
    ConnGit9,
    ConnSsh,
    ConnHttp,
};

struct Objlist {
    int idx;

    int fd;
    int state;
    int stage;

    Dir *top;
    int ntop;
    int topidx;
    Dir *loose;
    int nloose;
    int looseidx;
    Dir *pack;
    int npack;
    int packidx;
    int nent;
    int entidx;
};

struct Hash {
    uchar h[20];
};

struct Conn {
    int type;
    int rfd;
    int wfd;

    /* capabilities */
    char    symfrom[256];
    char    symto[256];
    char    multiack;
    char    sideband;
    char    sideband64k;
    char    report;

    /* only used by http */
    int cfd;
    char *url;  /* note, first GET uses a different url */
    char *dir;
    char *direction;
};

struct Dirent {
    char *name;
    int mode;
    Hash h;
    char ismod;
```

```
    char islink;
};

struct Object {
    /* Git data */
    Hash    hash;
    int type;

    /* Cache */
    int id;
    int flag;
    int refs;
    Object  *next;
    Object  *prev;

    /* For indexing */
    vlong   off;
    vlong   len;
    u32int  crc;

    /* Everything below here gets cleared */
    char    *all;
    char    *data;
    /* size excludes header */
    vlong   size;

    /* Significant win on memory use */
    union {
        Cinfo   *commit;
        Tinfo   *tree;
    };
};

struct Tinfo {
    /* Tree */
    Dirent  *ent;
    int nent;
};

struct Cinfo {
    /* Commit */
    Hash    *parent;
    int nparent;
    Hash    tree;
    char    *author;
    char    *committer;
    char    *msg;
    int nmsg;
    vlong   ctime;
    vlong   mtime;
};

struct Objset {
    Object  **obj;
    int nobj;
    int sz;
};

struct Qelt {
    Object  *o;
```

```
    vlong   ctime;
    int color;
};

struct Objq {
    Qelt    *heap;
    int nheap;
    int heapsz;
};

struct Dtab {
    Object  *o;
    uchar   *base;
    int nbase;
    Dblock  *b;
    int nb;
    int sz;
};

struct Dblock {
    uchar   *buf;
    int len;
    int off;
    u64int  hash;
};

struct Delta {
    int cpy;
    int off;
    int len;
};

struct Idxent {
    char    *path;
    Qid qid;
    int mode;
    int order;
    char    state;
};
```

⟨*macro* GETBE16 113a⟩

⟨*macro* GETBE32 113b⟩
⟨*macro* GETBE64 113c⟩

⟨*macro* PUTBE16 113d⟩

⟨*macro* PUTBE32 113e⟩

⟨*macro* PUTBE64 113f⟩

⟨*macro* QDIR 114a⟩
⟨*macro* isblank 114b⟩

```
extern Reprog   *authorpat;
extern Objset   objcache;
extern vlong    cachemax;
extern Hash Zhash;
extern int  chattygit;
extern int  interactive;
```

```
extern int  gitdirmode;

#pragma varargck type "H" Hash
#pragma varargck type "T" int
#pragma varargck type "O" Object*
#pragma varargck type "Q" Qid
int Hfmt(Fmt*);
int Tfmt(Fmt*);
int Ofmt(Fmt*);
int Qfmt(Fmt*);

void gitinit(char*, int, int*);

/* object io */
int resolverefs(Hash **, char *);
int resolveref(Hash *, char *);
int listrefs(Hash **, char ***);
Object  *ancestor(Object *, Object *);
int findtwixt(Hash *, int, Hash *, int, Object ***, int *);
Object  *readobject(Hash);
Object  *clearedobject(Hash, int);
int expandprefix(Hash*, Hash, int);
void    parseobject(Object *);
int indexpack(char *, char *, Hash);
int writepack(int, Hash*, int, Hash*, int, Hash*);
int hasheq(Hash *, Hash *);
Object  *ref(Object *);
void    unref(Object *);
void    cache(Object *);
Object  *emptydir(void);
int entcmp(void*, void*);

/* object sets */
void    osinit(Objset *);
void    osclear(Objset *);
void    osadd(Objset *, Object *);
int oshas(Objset *, Hash);
Object  *osfind(Objset *, Hash);

/* object listing */
Objlist *mkols(void);
int olsnext(Objlist *, Hash *);
void    olsfree(Objlist *);

/* util functions */
⟨macro dprint 114c⟩
void    _dprint(char *, ...);
void    *eamalloc(ulong, ulong);
void    *emalloc(ulong);
void    *earealloc(void *, ulong, ulong);
void    *erealloc(void *, ulong);
char    *estrdup(char *);
int slurpdir(char *, Dir **);
int hparse(Hash *, char *);
int hassuffix(char *, char *);
int swapsuffix(char *, int, char *, char *, char *);
char    *strip(char *);
int showprogress(int, int);
u64int  murmurhash2(void*, usize);
Qid parseqid(char*);
```

```
int charval(int);

/* packing */
void    dtinit(Dtab *, Object*);
void    dtclear(Dtab*);
Delta*  deltify(Object*, Dtab*, int*);

/* proto handling */
int readpkt(Conn*, char*, int);
int writepkt(Conn*, char*, int);
int fmtpkt(Conn*, char*, ...);
int flushpkt(Conn*);
void    initconn(Conn*, int, int);
int gitconnect(Conn *, char *, char *);
int readphase(Conn *);
int writephase(Conn *);
void    closeconn(Conn *);
void    parsecaps(char *, Conn *);
int okref(char*);

/* queues */
void    qinit(Objq*);
void    qclear(Objq*);
void    qput(Objq*, Object*, int);
int qpop(Objq*, Qelt*);
```

git9/log.c

⟨*function* filteradd 119⟩≡ (123c)

```
void
filteradd(Pfilt *pf, char *path)
{
    char *p, *e;
    int i;

    if((e = strchr(path, '/')) != nil)
        p = smprint("%.*s", (int)(e - path), path);
    else
        p = strdup(path);

    while(e != nil && *e == '/')
        e++;
    for(i = 0; i < pf->nsub; i++){
        if(strcmp(pf->sub[i].elt, p) == 0){
            pf->sub[i].show = pf->sub[i].show || (e == nil);
            if(e != nil)
                filteradd(&pf->sub[i], e);
            free(p);
            return;
        }
    }
    pf->sub = earealloc(pf->sub, pf->nsub+1, sizeof(Pfilt));
    pf->sub[pf->nsub].elt = p;
    pf->sub[pf->nsub].show = (e == nil);
    pf->sub[pf->nsub].nsub = 0;
    pf->sub[pf->nsub].sub = nil;
    if(e != nil)
        filteradd(&pf->sub[pf->nsub], e);
    pf->nsub++;
```

```
}
```

⟨*function* `lookup (git9/log.c)` 120a⟩≡ (123c)

```
Hash
lookup(Pfilt *pf, Object *o)
{
    int i;

    if(o == nil)
        return Zhash;
    for(i = 0; i < o->tree->nent; i++)
        if(strcmp(o->tree->ent[i].name, pf->elt) == 0)
            return o->tree->ent[i].h;
    return Zhash;
}
```

⟨*function* `matchesfilter1` 120b⟩≡ (123c)

```
int
matchesfilter1(Pfilt *pf, Object *t, Object *pt)
{
    Object *a, *b;
    Hash ha, hb;
    int i, r;

    if(pf->show)
        return 1;
    if(t != nil){
        if(pt != nil && t->type != pt->type)
            return 1;
        if(t->type != GTree)
            return 0;
    }

    for(i = 0; i < pf->nsub; i++){
        ha = lookup(&pf->sub[i], t);
        hb = lookup(&pf->sub[i], pt);
        if(hasheq(&ha, &hb))
            continue;
        a = readobject(ha);
        b = readobject(hb);
        r = matchesfilter1(&pf->sub[i], a, b);
        unref(a);
        unref(b);
        if(r)
            return 1;
    }
    return 0;
}
```

⟨*function* `matchesfilter` 120c⟩≡ (123c)

```
int
matchesfilter(Object *o)
{
    Object *t, *p, *pt;
    int i, r;

    assert(o->type == GCommit);
    if(pathfilt == nil)
        return 1;
    if((t = readobject(o->commit->tree)) == nil)
```

```
        sysfatal("read %H: %r", o->commit->tree);
    for(i = 0; i < o->commit->nparent; i++){
        if((p = readobject(o->commit->parent[i])) == nil)
            sysfatal("read %H: %r", o->commit->parent[i]);
        if((pt = readobject(p->commit->tree)) == nil)
            sysfatal("read %H: %r", o->commit->tree);
        r = matchesfilter1(pathfilt, t, pt);
        unref(p);
        unref(pt);
        if(r)
            return 1;
    }
    if(o->commit->nparent == 0)
        return matchesfilter1(pathfilt, t, nil);
    return 0;
}
```

⟨*function* nextline 121a⟩≡ (123c)

```
static char*
nextline(char *p, char *e)
{
    for(; p != e; p++)
        if(*p == '\n')
            break;
    return p;
}
```

⟨*function* show 121b⟩≡ (123c)

```
static int
show(Object *o)
{
    //Tm tm;
    char *p, *q, *e;

    assert(o->type == GCommit);
    if(shortlog){
        p = o->commit->msg;
        e = p + o->commit->nmsg;
        q = nextline(p, e);
        Bprint(out, "%H ", o->hash);
        Bwrite(out, p, q - p);
        Bputc(out, '\n');
    }else{
        //9front only
        //tmtime(&tm, o->commit->mtime, tzload("local"));
        Bprint(out, "Hash:\t%H\n", o->hash);
        Bprint(out, "Author:\t%s\n", o->commit->author);
        if(o->commit->committer != nil
        && strcmp(o->commit->author, o->commit->committer) != 0)
            Bprint(out, "Committer:\t%s\n", o->commit->committer);
        //Bprint(out, "Date:\t%τ\n", tmfmt(&tm, "WW MMM D hh:mm:ss z YYYY"));
        Bprint(out, "Date:\t%s", ctime(o->commit->mtime));
        Bprint(out, "\n");
        p = o->commit->msg;
        e = p + o->commit->nmsg;
        for(; p != e; p = q){
            q = nextline(p, e);
            Bputc(out, '\t');
            Bwrite(out, p, q - p);
            Bputc(out, '\n');
```

```
            if(q != e)
                q++;
        }
        Bprint(out, "\n");
    }
    Bflush(out);
    return 1;
}
```

⟨*function* showquery 122a⟩≡ (123c)

```
static void
showquery(char *q)
{
    Object *o;
    Hash *h;
    int n, i;

    if((n = resolverefs(&h, q)) == -1)
        sysfatal("resolve: %r");
    for(i = 0; i < n && (msgcount == -1 || msgcount > 0); i++){
        if((o = readobject(h[i])) == nil)
            sysfatal("read %H: %r", h[i]);
        if(matchesfilter(o)){
            show(o);
            if(msgcount != -1)
                msgcount--;
        }
        unref(o);
    }
    exits(nil);
}
```

⟨*function* showcommits 122b⟩≡ (123c)

```
static void
showcommits(char *c)
{
    Object *o, *p;
    Qelt e;
    int i;
    Hash h;

    if(c == nil)
        c = "HEAD";
    if(resolveref(&h, c) == -1)
        sysfatal("resolve %s: %r", c);
    if((o = readobject(h)) == nil)
        sysfatal("load %H: %r", h);
    if(o->type != GCommit)
        sysfatal("%s: not a commit", c);
    qinit(&objq);
    osinit(&done);
    qput(&objq, o, 0);
    while(qpop(&objq, &e) && (msgcount == -1 || msgcount > 0)){
        if(matchesfilter(e.o)){
            show(e.o);
            if(msgcount != -1)
                msgcount--;
        }
        for(i = 0; i < e.o->commit->nparent; i++){
            if(oshas(&done, e.o->commit->parent[i]))
```

```
                continue;
            if((p = readobject(e.o->commit->parent[i])) == nil)
                sysfatal("load %H: %r", o->commit->parent[i]);
            osadd(&done, p);
            qput(&objq, p, 0);
        }
        unref(e.o);
    }
}
```

⟨*function* usage (git9/log.c) 123a⟩≡ (123c)

```
static void
usage(void)
{
    fprint(2, "usage: %s [-s] [-e expr | -c commit] files..\n", argv0);
    exits("usage");
}
```

⟨*function* main (git9/log.c) 123b⟩≡ (123c)

```
void
main(int argc, char **argv)
{
    char path[1024], repo[1024], *p, *r;
    int i, nrel, nrepo;

    ARGBEGIN{
    case 'e':
        queryexpr = EARGF(usage());
        break;
    case 'c':
        commitid = EARGF(usage());
        break;
    case 's':
        shortlog++;
        break;
    case 'n':
        msgcount = atoi(EARGF(usage()));
        break;
    default:
        usage();
        break;
    }ARGEND;
```

⟨git9/log.c 123c⟩≡

```
#include <u.h>
#include <libc.h>
#include "git.h"

typedef struct Pfilt Pfilt;
struct Pfilt {
    char    *elt;
    int show;
    Pfilt   *sub;
    int nsub;
};

Biobuf  *out;
char    *queryexpr;
char    *commitid;
int shortlog;
```

```
int msgcount = -1;

Objset  done;
Objq    objq;
Pfilt   *pathfilt;
```

⟨*function* `filteradd` 119⟩

⟨*function* `lookup (git9/log.c)` 120a⟩

⟨*function* `matchesfilter1` 120b⟩

⟨*function* `matchesfilter` 120c⟩

⟨*function* `nextline` 121a⟩

⟨*function* `show` 121b⟩

⟨*function* `showquery` 122a⟩

⟨*function* `showcommits` 122b⟩

⟨*function* `usage (git9/log.c)` 123a⟩

⟨*function* `main (git9/log.c)` 123b⟩

```
    gitinit(repo, sizeof(repo), &nrel);
    nrepo = strlen(repo);
    if(argc != 0){
        if(getwd(path, sizeof(path)) == nil)
            sysfatal("getwd: %r");
        if(strncmp(path, repo, nrepo) != 0)
            sysfatal("path shifted??");
        p = path + nrepo;
        pathfilt = emalloc(sizeof(Pfilt));
        for(i = 0; i < argc; i++){
            if(*argv[i] == '/'){
                if(strncmp(argv[i], repo, nrepo) != 0)
                    continue;
                r = smprint("./%s", argv[i]+nrepo);
            }else
                r = smprint("./%s/%s", p, argv[i]);
            cleanname(r);
            filteradd(pathfilt, r);
            free(r);
        }
    }
    if(chdir(repo) == -1)
        sysfatal("chdir: %r");

    //9front-only
    //tmfmtinstall();
    out = Bfdopen(1, OWRITE);
    if(queryexpr != nil)
        showquery(queryexpr);
    else
        showcommits(commitid);
    Bterm(out);
    exits(nil);
```

```
}
```

git9/objset.c

⟨*function* osinit 125a⟩≡ (126b)

```
void
osinit(Objset *s)
{
    s->sz = 16;
    s->nobj = 0;
    s->obj = eamalloc(s->sz, sizeof(Hash));
}
```

⟨*function* osclear 125b⟩≡ (126b)

```
void
osclear(Objset *s)
{
    free(s->obj);
}
```

⟨*function* osadd 125c⟩≡ (126b)

```
void
osadd(Objset *s, Object *o)
{
    u32int probe;
    Object **obj;
    int i, sz;

    probe = GETBE32(o->hash.h) % s->sz;
    while(s->obj[probe]){
        if(hasheq(&s->obj[probe]->hash, &o->hash)){
            s->obj[probe] = o;
            return;
        }
        probe = (probe + 1) % s->sz;
    }
    assert(s->obj[probe] == nil);
    s->obj[probe] = o;
    s->nobj++;
    if(s->sz < 2*s->nobj){
        sz = s->sz;
        obj = s->obj;

        s->sz *= 2;
        s->nobj = 0;
        s->obj = eamalloc(s->sz, sizeof(Hash));
        for(i = 0; i < sz; i++)
            if(obj[i])
                osadd(s, obj[i]);
        free(obj);
    }
}
```

⟨*function* osfind 125d⟩≡ (126b)

```
Object*
osfind(Objset *s, Hash h)
{
    u32int probe;
```

```
    for(probe = GETBE32(h.h) % s->sz; s->obj[probe]; probe = (probe + 1) % s->sz)
        if(hasheq(&s->obj[probe]->hash, &h))
            return s->obj[probe];
    return 0;
}
```

⟨*function* oshas 126a⟩≡ (126b)

```
int
oshas(Objset *s, Hash h)
{
    return osfind(s, h) != nil;
}
```

⟨git9/objset.c 126b⟩≡

```
#include <u.h>
#include <libc.h>

#include "git.h"

⟨function osinit 125a⟩

⟨function osclear 125b⟩

⟨function osadd 125c⟩

⟨function osfind 125d⟩

⟨function oshas 126a⟩
```

git9/ols.c

⟨*function* crackidx 126c⟩≡ (129b)

```
static int
crackidx(char *path, int *np)
{
    int fd;
    char buf[4];

    if((fd = open(path, OREAD)) == -1)
        return -1;
    if(seek(fd, 8 + 255*4, 0) == -1)
        return -1;
    if(readn(fd, buf, sizeof(buf)) != sizeof(buf))
        return -1;
    *np = GETBE32(buf);
    return fd;
}
```

⟨*function* isloosedir 126d⟩≡ (129b)

```
int
isloosedir(char *s)
{
    return strlen(s) == 2 && isxdigit(s[0]) && isxdigit(s[1]);
}
```

⟨*function* endswith 127a⟩≡ (129b)

```
int
endswith(char *n, char *s)
{
    int nn, ns;

    nn = strlen(n);
    ns = strlen(s);
    return nn > ns && strcmp(n + nn - ns, s) == 0;
}
```

⟨*function* olsreadpacked 127b⟩≡ (129b)

```
int
olsreadpacked(Objlist *ols, Hash *h)
{
    char *p;
    int i, j;

    i = ols->packidx;
    j = ols->entidx;

    if(ols->state == Siter)
        goto step;
    for(i = 0; i < ols->npack; i++){
        if(!endswith(ols->pack[i].name, ".idx"))
            continue;
        if((p = smprint(".git/objects/pack/%s", ols->pack[i].name)) == nil)
            sysfatal("smprint: %r");
        ols->fd = crackidx(p, &ols->nent);
        free(p);
        if(ols->fd == -1)
            continue;
        j = 0;
        while(j < ols->nent){
            if(readn(ols->fd, h->h, sizeof(h->h)) != sizeof(h->h))
                continue;
            ols->state = Siter;
            ols->packidx = i;
            ols->entidx = j;
            return 0;
step:
            j++;
        }
        close(ols->fd);
    }
    ols->state = Sinit;
    return -1;
}
```

⟨*function* olsreadloose 127c⟩≡ (129b)

```
int
olsreadloose(Objlist *ols, Hash *h)
{
    char buf[64], *p;
    int i, j, n;

    i = ols->topidx;
    j = ols->looseidx;
    if(ols->state == Siter)
        goto step;
```

```
    for(i = 0; i < ols->ntop; i++){
        if(!isloosedir(ols->top[i].name))
            continue;
        if((p = smprint(".git/objects/%s", ols->top[i].name)) == nil)
            sysfatal("smprint: %r");
        ols->fd = open(p, OREAD);
        free(p);
        if(ols->fd == -1)
            continue;
        while((ols->nloose = dirread(ols->fd, &ols->loose)) > 0){
            j = 0;
            while(j < ols->nloose){
                n = snprint(buf, sizeof(buf), "%s%s", ols->top[i].name, ols->loose[j].name);
                if(n >= sizeof(buf))
                    goto step;
                if(hparse(h, buf) == -1)
                    goto step;
                ols->state = Siter;
                ols->topidx = i;
                ols->looseidx = j;
                return 0;
step:
                j++;
            }
            free(ols->loose);
            ols->loose = nil;
        }
        close(ols->fd);
        ols->fd = -1;
    }
    ols->state = Sinit;
    return -1;
}
```

⟨*function* mkols 128a⟩≡ (129b)

```
Objlist*
mkols(void)
{
    Objlist *ols;

    ols = emalloc(sizeof(Objlist));
    if((ols->ntop = slurpdir(".git/objects", &ols->top)) == -1)
        sysfatal("read top level: %r");
    if((ols->npack = slurpdir(".git/objects/pack", &ols->pack)) == -1)
        ols->pack = nil;
    ols->fd = -1;
    return ols;
}
```

⟨*function* olsfree 128b⟩≡ (129b)

```
void
olsfree(Objlist *ols)
{
    if(ols == nil)
        return;
    if(ols->fd != -1)
        close(ols->fd);
    free(ols->top);
    free(ols->loose);
    free(ols->pack);
```

```
    free(ols);
}
```

⟨*function* `olsnext` 129a⟩≡ (129b)

```
int
olsnext(Objlist *ols, Hash *h)
{
    if(ols->stage == 0){
        if(olsreadloose(ols, h) != -1){
            ols->idx++;
            return 0;
        }
        ols->stage++;
    }
    if(ols->stage == 1){
        if(olsreadpacked(ols, h) != -1){
            ols->idx++;
            return 0;
        }
        ols->stage++;
    }
    return -1;
}
```

⟨`git9/ols.c` 129b⟩≡

```
#include <u.h>
#include <libc.h>
#include <ctype.h>
#include "git.h"

enum {
    Sinit,
    Siter,
};
```

⟨*function* `crackidx` 126c⟩

⟨*function* `isloosedir` 126d⟩

⟨*function* `endswith` 127a⟩

⟨*function* `olsreadpacked` 127b⟩

⟨*function* `olsreadloose` 127c⟩

⟨*function* `mkols` 128a⟩

⟨*function* `olsfree` 128b⟩

⟨*function* `olsnext` 129a⟩

git9/pack.c

⟨*macro* `max` 129c⟩≡ (159b)

```
#define max(x, y) ((x) > (y) ? (x) : (y))
```

⟨*function* clear 130a⟩≡ (159b)

```
static void
clear(Object *o)
{
    if(!o)
        return;

    assert(o->refs == 0);
    assert((o->flag & Ccache) == 0);
    assert(o->flag & Cloaded);
    switch(o->type){
    case GCommit:
        if(!o->commit)
            break;
        free(o->commit->parent);
        free(o->commit->author);
        free(o->commit->committer);
        free(o->commit);
        o->commit = nil;
        break;
    case GTree:
        if(!o->tree)
            break;
        free(o->tree->ent);
        free(o->tree);
        o->tree = nil;
        break;
    default:
        break;
    }

    free(o->all);
    o->all = nil;
    o->data = nil;
    o->flag &= ~(Cloaded|Cparsed);
}
```

⟨*function* unref 130b⟩≡ (159b)

```
void
unref(Object *o)
{
    if(!o)
        return;
    o->refs--;
    if(o->refs == 0)
        clear(o);
}
```

⟨*function* ref 130c⟩≡ (159b)

```
Object*
ref(Object *o)
{
    o->refs++;
    return o;
}
```

⟨*function* cache 130d⟩≡ (159b)

```
void
cache(Object *o)
{
```

```
    Object *p;

    if(o == lruhead)
        return;
    if(o == lrutail)
        lrutail = lrutail->prev;
    if(!(o->flag & Cexist)){
        osadd(&objcache, o);
        o->id = objcache.nobj;
        o->flag |= Cexist;
    }
    if(o->prev != nil)
        o->prev->next = o->next;
    if(o->next != nil)
        o->next->prev = o->prev;
    if(lrutail == o){
        lrutail = o->prev;
        if(lrutail != nil)
            lrutail->next = nil;
    }else if(lrutail == nil)
        lrutail = o;
    if(lruhead)
        lruhead->prev = o;
    o->next = lruhead;
    o->prev = nil;
    lruhead = o;

    if(!(o->flag & Ccache)){
        o->flag |= Ccache;
        ref(o);
        ncache += o->size;
    }
    while(ncache > cachemax && lrutail != lruhead){
        p = lrutail;
        lrutail = p->prev;
        if(lrutail != nil)
            lrutail->next = nil;
        p->flag &= ~Ccache;
        p->prev = nil;
        p->next = nil;
        ncache -= p->size;
        unref(p);
    }
}
```

⟨*function* loadpack 131⟩≡ (159b)

```
static int
loadpack(Packf *pf, char *name)
{
    char buf[128];
    int i, ifd;
    Dir *d;

    memset(pf, 0, sizeof(Packf));
    snprint(buf, sizeof(buf), ".git/objects/pack/%s.idx", name);
    snprint(pf->path, sizeof(pf->path), ".git/objects/pack/%s.pack", name);
    /*
     * if we already have the pack open, just
     * steal the loaded info
     */
```

```
    for(i = 0; i < npackf; i++){
        if(strcmp(pf->path, packf[i].path) == 0){
            pf->pack = packf[i].pack;
            pf->idx = packf[i].idx;
            pf->nidx = packf[i].nidx;
            packf[i].idx = nil;
            packf[i].pack = nil;
            return 0;
        }
    }
    if((ifd = open(buf, OREAD)) == -1)
        return -1;
    if((d = dirfstat(ifd)) == nil){
        close(ifd);
        return -1;
    }
    pf->nidx = d->length;
    pf->idx = emalloc(pf->nidx);
    if(readn(ifd, pf->idx, pf->nidx) != pf->nidx){
        close(ifd);
        free(pf->idx);
        free(d);
        return -1;
    }
    close(ifd);
    free(d);
    return 0;
}
```

⟨*function* refreshpacks 132⟩≡ (159b)

```
static void
refreshpacks(void)
{
    Packf *pf, *new;
    int i, n, l, nnew;
    Dir *d;

    if((n = slurpdir(".git/objects/pack", &d)) == -1)
        return;
    nnew = 0;
    new = eamalloc(n, sizeof(Packf));
    for(i = 0; i < n; i++){
        l = strlen(d[i].name);
        if(l > 4 && strcmp(d[i].name + l - 4, ".idx") != 0)
            continue;
        d[i].name[l - 4] = 0;
        if(loadpack(&new[nnew], d[i].name) != -1)
            nnew++;
    }
    for(i = 0; i < npackf; i++){
        pf = &packf[i];
        free(pf->idx);
        if(pf->pack != nil)
            Bterm(pf->pack);
    }
    free(packf);
    packf = new;
    npackf = nnew;
    free(d);
}
```

⟨*function* openpack 133a⟩≡ (159b)

```
static Biobuf*
openpack(Packf *pf)
{
    vlong t;
    int i, best;

    if(pf->pack != nil){
        pf->refs++;
        return pf->pack;
    }
    /*
     * If we've got more packs open
     * than we want cached, try to
     * free up the oldest ones.
     *
     * If we can't find a slot, this
     * isn't fatal; we can just use
     * another fd.
     */
    while(openpacks >= Npackcache){
        t = (1ull<<62)-1;
        best = -1;
        for(i = 0; i < npackf; i++){
            if(&packf[i] != pf
            && packf[i].pack != nil
            && packf[i].opentm < t
            && packf[i].refs == 0){
                t = packf[i].opentm;
                best = i;
            }
        }
        if(best == -1){
            fprint(2, "no available pack slots\n");
            break;
        }
        Bterm(packf[best].pack);
        packf[best].pack = nil;
        openpacks--;
    }
    openpacks++;
    pf->opentm = nsec();
    pf->refs++;
    if((pf->pack = Bopen(pf->path, OREAD)) == nil)
        return nil;
    return pf->pack;
}
```

⟨*function* closepack 133b⟩≡ (159b)

```
static void
closepack(Packf *pf)
{
    pf->refs--;
}
```

⟨*function* crc32 133c⟩≡ (159b)

```
static u32int
crc32(u32int crc, char *b, int nb)
{
    static u32int crctab[256] = {
```

```
        0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f, 0xe963a535,
        0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988, 0x09b64c2b, 0x7eb17cbd,
        0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de, 0x1adad47d,
        0x6ddde4eb, 0xf4d4b551, 0x83d385c7, 0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec,
        0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4,
        0xa2677172, 0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0x42b2986c,
        0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59, 0x26d930ac,
        0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f,
        0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924, 0x2f6f7c87, 0x58684c11, 0xc1611dab,
        0xb6662d3d, 0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f,
        0x9fbfe4a5, 0xe8b8d433, 0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb,
        0x086d3d2d, 0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
        0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea,
        0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65, 0x4db26158, 0x3ab551ce,
        0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a,
        0x346ed9fc, 0xad678846, 0xda60b8d0, 0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9,
        0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409,
        0xce61e49f, 0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
        0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a, 0xead54739,
        0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8,
        0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1, 0xf00f9344, 0x8708a3d2, 0x1e01f268,
        0x6906c2fe, 0xf762575d, 0x806567cb, 0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0,
        0x10da7a5a, 0x67dd4acc, 0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8,
        0xa1d1937e, 0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
        0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55, 0x316e8eef,
        0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236, 0xcc0c7795, 0xbb0b4703,
        0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7,
        0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d, 0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a,
        0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae,
        0x0cb61b38, 0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
        0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777, 0x88085ae6,
        0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45,
        0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7, 0x4969474d,
        0x3e6e77db, 0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5,
        0x47b2cf7f, 0x30b5ffe9, 0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605,
        0xcdd70693, 0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
        0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
    };
    int i;

    crc ^=  0xFFFFFFFF;
    for(i = 0; i < nb; i++)
        crc = (crc >> 8) ^ crctab[(crc ^ b[i]) & 0xFF];
    return crc ^ 0xFFFFFFFF;
}
```

⟨*function* bappend 134⟩≡ (159b)

```
int
bappend(void *p, void *src, int len)
{
    Buf *b = p;
    char *n;

    while(b->len + len >= b->sz){
        b->sz = b->sz*2 + 64;
        n = realloc(b->data, b->sz);
        if(n == nil)
            return -1;
        b->data = n;
    }
```

```
    memmove(b->data + b->len, src, len);
    b->len += len;
    return len;
}
```

⟨*function* breadc 135a⟩≡ (159b)

```
int
breadc(void *p)
{
    return Bgetc(p);
}
```

⟨*function* bdecompress 135b⟩≡ (159b)

```
int
bdecompress(Buf *d, Biobuf *b, vlong *csz)
{
    vlong o;

    o = Boffset(b);
    if(inflatezlib(d, bappend, b, breadc) == -1 || d->data == nil){
        free(d->data);
        return -1;
    }
    if (csz)
        *csz = Boffset(b) - o;
    return d->len;
}
```

⟨*function* decompress 135c⟩≡ (159b)

```
int
decompress(void **p, Biobuf *b, vlong *csz)
{
    Buf d = {.len=0, .data=nil, .sz=0};

    if(bdecompress(&d, b, csz) == -1){
        free(d.data);
        return -1;
    }
    *p = d.data;
    return d.len;
}
```

⟨*function* readvint 135d⟩≡ (159b)

```
static vlong
readvint(char *p, char *ep, char **pp)
{
    int s, c;
    vlong n;

    s = 0;
    n = 0;
    do {
        if(p == ep)
            return -1;
        c = *p++;
        n |= (vlong)(c & 0x7f) << s;
        s += 7;
    } while (c & 0x80 && s < 63);
    *pp = p;
    return n;
}
```

⟨*function* applydelta 136⟩≡ (159b)

```
static int
applydelta(Object *dst, Object *base, char *d, int nd)
{
    char *r, *b, *ed, *er;
    vlong o, l,  n, nr, c;

    ed = d + nd;
    b = base->data;
    n = readvint(d, ed, &d);
    if(n == -1 || n >= 1LL << 31 || n != base->size){
        werrstr("mismatched source size");
        return -1;
    }

    nr = readvint(d, ed, &d);
    if(nr == -1 || nr >= 1LL << 31){
        werrstr("invalid pack: %r");
        return -1;
    }
    r = emalloc(nr + 64);
    n = snprint(r, 64, "%T %lld", base->type, nr) + 1;
    dst->all = r;
    dst->type = base->type;
    dst->data = r + n;
    dst->size = nr;
    er = dst->data + nr;
    r = dst->data;

    while(d != ed){
        c = *d++;
        /* copy from base */
        if(c & 0x80){
            o = 0;
            l = 0;
            /* Offset in base */
            if(d != ed && (c & 0x01)) o |= (*d++ & 0xff) <<  0;
            if(d != ed && (c & 0x02)) o |= (*d++ & 0xff) <<  8;
            if(d != ed && (c & 0x04)) o |= (*d++ & 0xff) << 16;
            if(d != ed && (c & 0x08)) o |= (*d++ & 0xff) << 24;

            /* Length to copy */
            if(d != ed && (c & 0x10)) l |= (*d++ & 0xff) <<  0;
            if(d != ed && (c & 0x20)) l |= (*d++ & 0xff) <<  8;
            if(d != ed && (c & 0x40)) l |= (*d++ & 0xff) << 16;
            if(l == 0) l = 0x10000;

            if(o < 0 || l < 0 || l > er - r || o + l > base->size){
                werrstr("garbled delta: out of bounds copy");
                return -1;
            }
            memmove(r, b + o, l);
            r += l;
        /* inline data */
        }else{
            if(c > ed - d || c > er - r){
                werrstr("garbled delta: write past object");
                return -1;
            }
            memmove(r, d, c);
```

```
            d += c;
            r += c;
        }
    }
    if(r != er){
        werrstr("truncated delta");
        return -1;
    }

    return nr;
}
```

⟨*function* `readrdelta` 137a⟩≡ (159b)

```
static int
readrdelta(Biobuf *f, Object *o, int nd, int flag)
{
    Object *b;
    Hash h;
    char *d;
    int n;

    d = nil;
    if(Bread(f, h.h, sizeof(h.h)) != sizeof(h.h))
        goto error;
    if(hasheq(&o->hash, &h))
        goto error;
    if((n = decompress(&d, f, nil)) == -1)
        goto error;
    o->len = Boffset(f) - o->off;
    if(d == nil || n != nd)
        goto error;
    if((b = readidxobject(f, h, flag|Cthin)) == nil)
        goto error;
    if(applydelta(o, b, d, n) == -1)
        goto error;
    free(d);
    return 0;
error:
    free(d);
    return -1;
}
```

⟨*function* `readodelta` 137b⟩≡ (159b)

```
static int
readodelta(Biobuf *f, Object *o, vlong nd, vlong p, int flag)
{
    Object b;
    char *d;
    vlong r;
    int c, n;

    d = nil;
    if((c = Bgetc(f)) == -1)
        return -1;
    r = c & 0x7f;
    while(c & 0x80 && r < (1ULL<<56)){
        if((c = Bgetc(f)) == -1)
            return -1;
        r = ((r + 1)<<7) | (c & 0x7f);
    }
```

```
    if(r > p || r >= (1ULL<<56)){
        werrstr("junk offset -%lld (from %lld)", r, p);
        goto error;
    }
    if((n = decompress(&d, f, nil)) == -1)
        goto error;
    o->len = Boffset(f) - o->off;
    if(d == nil || n != nd)
        goto error;
    if(Bseek(f, p - r, 0) == -1)
        goto error;
    memset(&b, 0, sizeof(Object));
    if(readpacked(f, &b, flag) == -1)
        goto error;
    if(applydelta(o, &b, d, nd) == -1)
        goto error;
    clear(&b);
    free(d);
    return 0;
error:
    free(d);
    return -1;
}
```

⟨*function* readpacked 138⟩≡ (159b)

```
static int
readpacked(Biobuf *f, Object *o, int flag)
{
    int c, s, n, t;
    vlong l, p;
    Buf b;

    p = Boffset(f);
    c = Bgetc(f);
    if(c == -1)
        return -1;
    l = c & 0xf;
    s = 4;
    t = (c >> 4) & 0x7;
    if(!t){
        werrstr("unknown type for byte %x at %lld", c, p);
        return -1;
    }
    while(c & 0x80){
        if((c = Bgetc(f)) == -1)
            return -1;
        l |= (vlong)(c & 0x7f) << s;
        s += 7;
    }
    if(l >= (1ULL << 32)){
        werrstr("object too big");
        return -1;
    }
    switch(t){
    default:
        werrstr("invalid object at %lld", Boffset(f));
        return -1;
    case GCommit:
    case GTree:
```

```
    case GTag:
    case GBlob:
        b.sz = 64 + l;
        b.data = emalloc(b.sz);
        n = snprint(b.data, 64, "%T %lld", t, l) + 1;
        b.len = n;
        if(bdecompress(&b, f, nil) == -1){
            free(b.data);
            return -1;
        }
        o->len = Boffset(f) - o->off;
        o->type = t;
        o->all = b.data;
        o->data = b.data + n;
        o->size = b.len - n;
        break;
    case GOdelta:
        if(readodelta(f, o, l, p, flag) == -1)
            return -1;
        break;
    case GRdelta:
        if(readrdelta(f, o, l, flag) == -1)
            return -1;
        break;
    }
    o->flag |= Cloaded|flag;
    return 0;
}
```

⟨*function* readloose 139⟩≡ (159b)

```
static int
readloose(Biobuf *f, Object *o, int flag)
{
    struct { char *tag; int type; } *p, types[] = {
        {"blob", GBlob},
        {"tree", GTree},
        {"commit", GCommit},
        {"tag", GTag},
        {nil},
    };
    char *d, *s, *e, *ed;
    vlong sz, n;
    int l;

    n = decompress(&d, f, nil);
    if(n == -1)
        return -1;

    s = d;
    ed = d + n;
    o->type = GNone;
    for(p = types; p->tag; p++){
        l = strlen(p->tag);
        if(strncmp(s, p->tag, l) == 0){
            s += l;
            o->type = p->type;
            while(s != ed && !isspace(*s))
                s++;
            break;
        }
```

```
    }
    if(o->type == GNone){
        free(o->data);
        return -1;
    }
    sz = strtol(s, &e, 0);
    if(e == s || *e++ != 0){
        werrstr("malformed object header");
        goto error;
    }
    if(sz != n - (e - d)){
        werrstr("mismatched sizes");
        goto error;
    }
    o->size = sz;
    o->data = e;
    o->all = d;
    o->flag |= Cloaded|flag;
    return 0;

error:
    free(d);
    return -1;
}
```

⟨*function* hashcmp 140a⟩≡ (159b)

```
int
hashcmp(uchar *a, uchar *b, uint nbit)
{
    int x, y, i, r;

    i = nbit/8;
    r = memcmp(a, b, i);
    if(r != 0 || nbit % 8 == 0)
        return r;
    x = a[i] >> (8 - nbit % 8);
    y = b[i] >> (8 - nbit % 8);
    return x - y;
}
```

⟨*function* searchindex 140b⟩≡ (159b)

```
vlong
searchindex(char *idx, int nidx, Hash h, int npfx, Hash *hret)
{
    uint lo, hi, hidx;
    vlong o, oo, nent;
    int i, r;
    void *s;

    o = 8;
    if(nidx < 8 + 256*4)
        return -1;
    if(npfx < 8)
        return -1;
    /*
     * Read the fanout table. The fanout table
     * contains 256 entries, corresponsding to
     * the first byte of the hash. Each entry
     * is a 4 byte big endian integer, containing
     * the total number of entries with a leading
```

```
 * byte <= the table index, allowing us to
 * rapidly do a binary search on them.
 */
if (h.h[0] == 0){
    lo = 0;
    hi = GETBE32(idx + o);
} else {
    o += h.h[0]*4 - 4;
    lo = GETBE32(idx + o);
    hi = GETBE32(idx + o + 4);
}
if(hi == lo)
    goto notfound;
nent = GETBE32(idx + 8 + 255*4);

/*
 * Now that we know the range of hashes that the
 * entry may exist in, search them
 */
r = -1;
s = nil;
hidx = -1;
o = 8 + 256*4;
while(lo < hi){
    hidx = (hi + lo)/2;
    s = idx + o + hidx*sizeof(h.h);
    r = hashcmp(h.h, s, npfx);
    if(r < 0)
        hi = hidx;
    else if(r > 0)
        lo = hidx + 1;
    else
        break;
}
if(r != 0)
    goto notfound;
if(hret != nil)
    memcpy(hret, s, sizeof(Hash));

/*
 * We found the entry. If it's 32 bits, then we
 * can just return the oset, otherwise the 32
 * bit entry contains the oset to the 64 bit
 * entry.
 */
oo = 8;         /* Header */
oo += 256*4;        /* Fanout table */
oo += Hashsz*nent;  /* Hashes */
oo += 4*nent;       /* Checksums */
oo += 4*hidx;       /* Offset offset */
if(oo < 0 || oo + 4 > nidx)
    goto err;
i = GETBE32(idx + oo);
o = i & 0xffffffffULL;
/*
 * Large offsets (i.e. 64-bit) are encoded as an index
 * into the next table with the MSB bit set.
 */
if(o & (1ull << 31)){
    o &= 0x7fffffffULL;
```

```
        oo = 8;             /* Header */
        oo += 256*4;            /* Fanout table */
        oo += Hashsz*nent;      /* Hashes */
        oo += 4*nent;           /* Checksums */
        oo += 4*nent;           /* 32-bit Offsets */
        oo += 8*o;          /* 64-bit Offset offset */
        if(oo < 0 || oo + 8 >= nidx)
            goto err;
        o = GETBE64(idx + oo);
    }
    return o;

err:
    werrstr("out of bounds read");
    return -1;
notfound:
    werrstr("not present");
    return -1;
}
```

⟨*function* scanword 142⟩≡ (159b)

```
/*
 * Scans for non-empty word, copying it into buf.
 * Strips off word, leading, and trailing space
 * from input.
 *
 * Returns -1 on empty string or error, leaving
 * input unmodified.
 */
static int
scanword(char **str, int *nstr, char *buf, int nbuf)
{
    char *p;
    int n, r;

    r = -1;
    p = *str;
    n = *nstr;
    while(n && isblank(*p)){
        n--;
        p++;
    }

    for(; n && *p && !isspace(*p); p++, n--){
        r = 0;
        *buf++ = *p;
        nbuf--;
        if(nbuf == 0)
            return -1;
    }
    while(n && isblank(*p)){
        n--;
        p++;
    }
    *buf = 0;
    *str = p;
    *nstr = n;
    return r;
}
```

⟨*function* nextline (git9/pack.c) 143a⟩≡ (159b)

```
static void
nextline(char **str, int *nstr)
{
    char *s;

    if((s = strchr(*str, '\n')) != nil){
        *nstr -= s - *str + 1;
        *str = s + 1;
    }
}
```

⟨*function* parseauthor 143b⟩≡ (159b)

```
static int
parseauthor(char **str, int *nstr, char **name, vlong *time)
{
    char buf[512];
    Resub m[4];
    vlong tz;
    char *p;
    int n, nm;

    if((p = strchr(*str, '\n')) == nil)
        sysfatal("malformed author line");
    n = p - *str;
    if(n >= sizeof(buf))
        sysfatal("overlong author line");
    memset(m, 0, sizeof(m));
    memcpy(buf, *str, n);
    buf[n] = 0;
    *str = p;
    *nstr -= n;

    if(!regexec(authorpat, buf, m, nelem(m)))
        sysfatal("invalid author line %s", buf);
    nm = m[1].e.ep - m[1].s.sp;
    *name = emalloc(nm + 1);
    memcpy(*name, m[1].s.sp, nm);
    buf[nm] = 0;

    nm = m[3].e.ep - m[3].s.sp;
    memcpy(buf, m[3].s.sp, nm);
    buf[nm] = 0;
    tz = atoll(buf);

    nm = m[2].e.ep - m[2].s.sp;
    memcpy(buf, m[2].s.sp, nm);
    buf[nm] = 0;
    *time = atoll(buf) + 3600*(tz/100) + 60*(tz%100);

    return 0;
}
```

⟨*function* parsecommit 143c⟩≡ (159b)

```
static void
parsecommit(Object *o)
{
    char *p, *t, buf[512];
    int np;
```

```
    p = o->data;
    np = o->size;
    o->commit = emalloc(sizeof(Cinfo));
    while(1){
        if(scanword(&p, &np, buf, sizeof(buf)) == -1)
            break;
        if(strcmp(buf, "tree") == 0){
            if(scanword(&p, &np, buf, sizeof(buf)) == -1)
                sysfatal("invalid commit: tree missing");
            if(hparse(&o->commit->tree, buf) == -1)
                sysfatal("invalid commit: garbled tree");
        }else if(strcmp(buf, "parent") == 0){
            if(scanword(&p, &np, buf, sizeof(buf)) == -1)
                sysfatal("invalid commit: missing parent");
            o->commit->parent = realloc(o->commit->parent, ++o->commit->nparent * sizeof(Hash));
            if(!o->commit->parent)
                sysfatal("unable to malloc: %r");
            if(hparse(&o->commit->parent[o->commit->nparent - 1], buf) == -1)
                sysfatal("invalid commit: garbled parent");
        }else if(strcmp(buf, "author") == 0){
            parseauthor(&p, &np, &o->commit->author, &o->commit->mtime);
        }else if(strcmp(buf, "committer") == 0){
            parseauthor(&p, &np, &o->commit->committer, &o->commit->ctime);
        }else if(strcmp(buf, "gpgsig") == 0){
            /* just drop it */
            if((t = strstr(p, "-----END PGP SIGNATURE-----")) == nil)
            if((t = strstr(p, "-----END SSH SIGNATURE-----")) == nil)
            if((t = strstr(p, "-----END SIGNED MESSAGE-----")) == nil)
                sysfatal("malformed gpg signature");
            np -= t - p;
            p = t;
        }
        nextline(&p, &np);
    }
    while (np && isspace(*p)) {
        p++;
        np--;
    }
    o->commit->msg = p;
    o->commit->nmsg = np;
}
```

⟨*function* parsetree 144⟩≡ (159b)

```
static void
parsetree(Object *o)
{
    int m, a, entsz, nent;
    Dirent *t, *ent;
    char *p, *ep;

    p = o->data;
    ep = p + o->size;

    nent = 0;
    entsz = 16;
    ent = eamalloc(entsz, sizeof(Dirent));
    o->tree = emalloc(sizeof(Tinfo));
    while(p != ep){
        if(nent == entsz){
            entsz *= 2;
```

```
            ent = earealloc(ent, entsz, sizeof(Dirent));
        }
        t = &ent[nent++];
        m = strtol(p, &p, 8);
        if(*p != ' ')
            sysfatal("malformed tree %H: *p=(%d) %c", o->hash, *p, *p);
        p++;
        /*
         * only the stored permissions for the user
         * are relevant; git fills group and world
         * bits with whatever -- so to serve with
         * useful permissions, replicate the mode
         * of the git repo dir.
         */
        t->mode = gitdirmode;
        t->ismod = 0;
        t->islink = 0;
        if(m & 0777){
            a = (m & 0777)>>6;
            t->mode &= ((a<<6)|(a<<3)|a);
        }
        if(m == 0160000){ /* module */
            t->mode |= DMDIR;
            t->ismod = 1;
        }else if(m == 0120000){ /* symlink */
            t->mode = 0;
            t->islink = 1;
        }
        if(m & 0040000) /* dir */
            t->mode |= DMDIR;
        t->name = p;
        p = memchr(p, 0, ep - p);
        if(p == nil || *p++ != 0 ||  ep - p < sizeof(t->h.h))
            sysfatal("malformed tree %H, remaining %d (%s)", o->hash, (int)(ep - p), p);
        memcpy(t->h.h, p, sizeof(t->h.h));
        p += sizeof(t->h.h);
    }
    o->tree->ent = ent;
    o->tree->nent = nent;
}
```

⟨*function* parsetag 145a⟩≡ (159b)

```
static void
parsetag(Object *)
{
}
```

⟨*function* parseobject 145b⟩≡ (159b)

```
void
parseobject(Object *o)
{
    if(o->flag & Cparsed)
        return;
    switch(o->type){
    case GTree: parsetree(o);   break;
    case GCommit:   parsecommit(o); break;
    case GTag:  parsetag(o);    break;
    default:    break;
    }
    o->flag |= Cparsed;
}
```

⟨*function* readidxobject 146⟩≡ (159b)

```
static Object*
readidxobject(Biobuf *idx, Hash h, int flag)
{
    char path[Pathmax], hbuf[41];
    Object *obj, *new;
    int i, r, retried;
    Biobuf *f;
    vlong o;

    if((obj = osfind(&objcache, h)) != nil){
        if(flag & Cidx){
            /*
             * If we're indexing, we need to be careful
             * to only return objects within this pack,
             * so if the objects exist outside the pack,
             * we don't index the wrong copy.
             */
            if(!(obj->flag & Cidx))
                return nil;
            if(obj->flag & Cloaded)
                return obj;
            o = Boffset(idx);
            if(Bseek(idx, obj->off, 0) == -1)
                return nil;
            if(readpacked(idx, obj, flag) == -1)
                return nil;
            if(Bseek(idx, o, 0) == -1)
                sysfatal("could not restore offset");
            cache(obj);
            return obj;
        }
        if(obj->flag & Cloaded)
            return obj;
    }
    if(flag & Cthin)
        flag &= ~Cidx;
    if(flag & Cidx)
        return nil;
    new = nil;
    if(obj == nil){
        new = emalloc(sizeof(Object));
        new->id = objcache.nobj + 1;
        new->hash = h;
        obj = new;
    }

    o = -1;
    retried = 0;
retry:
    for(i = 0; i < npackf; i++){
        o = searchindex(packf[i].idx, packf[i].nidx, h, SHA1dlen*8, nil);
        if(o != -1){
            if((f = openpack(&packf[i])) == nil)
                goto error;
            if((r = Bseek(f, o, 0)) != -1)
                r = readpacked(f, obj, flag);
            closepack(&packf[i]);
            if(r == -1)
                goto error;
```

```
            parseobject(obj);
            cache(obj);
            return obj;
        }
    }

    snprint(hbuf, sizeof(hbuf), "%H", h);
    snprint(path, sizeof(path), ".git/objects/%c%c/%s", hbuf[0], hbuf[1], hbuf + 2);
    if((f = Bopen(path, OREAD)) != nil){
        if(readloose(f, obj, flag) == -1)
            goto errorf;
        Bterm(f);
        parseobject(obj);
        cache(obj);
        return obj;
    }

    if(o == -1){
        if(retried)
            goto error;
        retried = 1;
        refreshpacks();
        goto retry;
    }
errorf:
    Bterm(f);
error:
    free(new);
    return nil;
}
```

⟨*function* expandprefix 147⟩≡ (159b)

```
int
expandprefix(Hash *rh, Hash h, int npfx)
{
    int i, fd, ndir;
    char buf[128];
    Dir *d;

    refreshpacks();
    if(npfx < 8 || npfx % 4 != 0)
        return -1;
    for(i = 0; i < npackf; i++)
        if(searchindex(packf[i].idx, packf[i].nidx, h, npfx, rh) != -1)
            return 0;
    sprint(buf, ".git/objects/%x", h.h[0]);
    if((fd = open(buf, OREAD)) == -1)
        return -1;
    ndir = dirreadall(fd, &d);
    close(fd);
    if(ndir == -1)
        return -1;
    for(i = 0; i < ndir; i++){
        snprint(buf, sizeof(buf), "%x%s", h.h[0], d[i].name);
        if(hparse(rh, buf) == 0 && hashcmp(h.h, rh->h, npfx) == 0){
            free(d);
            return 0;
        }

    }
```

```
    free(d);
    return -1;
}
```

⟨*function* `readobject` 148a⟩≡ (159b)

```
/*
 * Loads and returns a cached object.
 */
Object*
readobject(Hash h)
{
    Object *o;

    if((o = readidxobject(nil, h, 0)) == nil)
        return nil;
    parseobject(o);
    ref(o);
    return o;
}
```

⟨*function* `clearedobject` 148b⟩≡ (159b)

```
/*
 * Creates and returns a cached, cleared object
 * that will get loaded some other time. Useful
 * for performance if need to mark that a blob
 * exists, but we don't care about its contents.
 *
 * The refcount of the returned object is 0, so
 * it doesn't need to be unrefed.
 */
Object*
clearedobject(Hash h, int type)
{
    Object *o;

    if((o = osfind(&objcache, h)) != nil)
        return o;

    o = emalloc(sizeof(Object));
    o->hash = h;
    o->type = type;
    osadd(&objcache, o);
    o->id = objcache.nobj;
    o->flag |= Cexist;
    return o;
}
```

⟨*function* `objcmp` 148c⟩≡ (159b)

```
int
objcmp(void *pa, void *pb)
{
    Object *a, *b;

    a = *(Object**)pa;
    b = *(Object**)pb;
    return memcmp(a->hash.h, b->hash.h, sizeof(a->hash.h));
}
```

⟨*function* hwrite 149a⟩≡ (159b)

```
static int
hwrite(Biobuf *b, void *buf, int len, DigestState **st)
{
    *st = sha1(buf, len, nil, *st);
    return Bwrite(b, buf, len);
}
```

⟨*function* objectcrc 149b⟩≡ (159b)

```
static u32int
objectcrc(Biobuf *f, Object *o)
{
    char buf[8096];
    int n, r;

    o->crc = 0;
    Bseek(f, o->off, 0);
    for(n = o->len; n > 0; n -= r){
        r = Bread(f, buf, n > sizeof(buf) ? sizeof(buf) : n);
        if(r == -1)
            return -1;
        if(r == 0)
            return 0;
        o->crc = crc32(o->crc, buf, r);
    }
    return 0;
}
```

⟨*function* indexpack 149c⟩≡ (159b)

```
int
indexpack(char *pack, char *idx, Hash ph)
{
    char hdr[4*3], buf[8];
    int nobj, npct, nvalid, nbig;
    int i, n, pct;
    Object *o, **obj;
    DigestState *st;
    char *valid;
    Biobuf *f;
    Hash h;
    int c;

    if((f = Bopen(pack, OREAD)) == nil)
        return -1;
    if(Bread(f, hdr, sizeof(hdr)) != sizeof(hdr)){
        werrstr("short read on header");
        return -1;
    }
    if(memcmp(hdr, "PACK\0\0\0\2", 8) != 0){
        werrstr("invalid header");
        return -1;
    }

    pct = 0;
    npct = 0;
    nvalid = 0;
    nobj = GETBE32(hdr + 8);
    obj = eamalloc(nobj, sizeof(Object*));
    valid = eamalloc(nobj, sizeof(char));
    if(interactive)
```

```
    fprint(2, "indexing %d objects:   0%%", nobj);
while(nvalid != nobj){
    n = 0;
    for(i = 0; i < nobj; i++){
        if(valid[i]){
            n++;
            continue;
        }
        pct = showprogress((npct*100)/nobj, pct);
        if(obj[i] == nil){
            o = emalloc(sizeof(Object));
            o->off = Boffset(f);
            obj[i] = o;
        }
        o = obj[i];
        /*
         * We can seek around when packing delta chains.
         * Be extra careful while we don't know where all
         * the objects start.
         */
        Bseek(f, o->off, 0);
        if(readpacked(f, o, Cidx) == -1)
            continue;
        sha1((uchar*)o->all, o->size + strlen(o->all) + 1, o->hash.h, nil);
        valid[i] = 1;
        cache(o);
        npct++;
        n++;
        if(objectcrc(f, o) == -1)
            return -1;
    }
    if(n == nvalid)
        sysfatal("fix point reached too early: %d/%d: %r", nvalid, nobj);
    nvalid = n;
}
if(interactive)
    fprint(2, "\b\b\b\b100%%\n");
Bterm(f);

st = nil;
qsort(obj, nobj, sizeof(Object*), objcmp);
if((f = Bopen(idx, OWRITE)) == nil)
    return -1;
if(hwrite(f, "\xfftOc\x00\x00\x00\x02", 8, &st) != 8)
    goto error;
/* fanout table */
c = 0;
for(i = 0; i < 256; i++){
    while(c < nobj && (obj[c]->hash.h[0] & 0xff) <= i)
        c++;
    PUTBE32(buf, c);
    if(hwrite(f, buf, 4, &st) == -1)
        goto error;
}
for(i = 0; i < nobj; i++){
    o = obj[i];
    if(hwrite(f, o->hash.h, sizeof(o->hash.h), &st) == -1)
        goto error;
}
```

```
    for(i = 0; i < nobj; i++){
        PUTBE32(buf, obj[i]->crc);
        if(hwrite(f, buf, 4, &st) == -1)
            goto error;
    }

    nbig = 0;
    for(i = 0; i < nobj; i++){
        if(obj[i]->off < (1ull<<31))
            PUTBE32(buf, obj[i]->off);
        else{
            PUTBE32(buf, (1ull << 31) | nbig);
            nbig++;
        }
        if(hwrite(f, buf, 4, &st) == -1)
            goto error;
    }
    for(i = 0; i < nobj; i++){
        if(obj[i]->off >= (1ull<<31)){
            PUTBE64(buf, obj[i]->off);
            if(hwrite(f, buf, 8, &st) == -1)
                goto error;
        }
    }
    if(hwrite(f, ph.h, sizeof(ph.h), &st) == -1)
        goto error;
    sha1(nil, 0, h.h, st);
    Bwrite(f, h.h, sizeof(h.h));

    free(obj);
    free(valid);
    Bterm(f);
    return 0;

error:
    free(obj);
    free(valid);
    Bterm(f);
    return -1;
}
```

⟨*function* deltaordercmp 151⟩≡ (159b)

```
static int
deltaordercmp(void *pa, void *pb)
{
    Meta *a, *b;
    vlong cmp;

    a = *(Meta**)pa;
    b = *(Meta**)pb;
    if(a->obj->type != b->obj->type)
        return a->obj->type - b->obj->type;
    cmp = (b->path - a->path);
    if(cmp != 0)
        return (cmp < 0) ? -1 : 1;
    cmp = a->mtime - b->mtime;
    if(cmp != 0)
        return (cmp < 0) ? -1 : 1;
    return memcmp(a->obj->hash.h, b->obj->hash.h, sizeof(a->obj->hash.h));
}
```

⟨*function* writeordercmp 152a⟩≡ (159b)

```
static int
writeordercmp(void *pa, void *pb)
{
    Meta *a, *b, *ahd, *bhd;

    a = *(Meta**)pa;
    b = *(Meta**)pb;
    ahd = (a->head == nil) ? a : a->head;
    bhd = (b->head == nil) ? b : b->head;
    if(ahd->mtime != bhd->mtime)
        return bhd->mtime - ahd->mtime;
    if(ahd != bhd)
        return (uintptr)bhd - (uintptr)ahd;
    if(a->nchain != b->nchain)
        return a->nchain - b->nchain;
    return a->mtime - b->mtime;
}
```

⟨*function* addmeta 152b⟩≡ (159b)

```
static void
addmeta(Metavec *v, Objset *has, Object *o, vlong pathid, vlong mtime)
{
    Meta *m;

    if(oshas(has, o->hash))
        return;
    osadd(has, o);
    if(v == nil)
        return;
    m = emalloc(sizeof(Meta));
    m->obj = o;
    m->path = pathid;
    m->mtime = mtime;

    if(v->nmeta == v->metasz){
        v->metasz = 2*v->metasz;
        v->meta = earealloc(v->meta, v->metasz, sizeof(Meta*));
    }
    v->meta[v->nmeta++] = m;
}
```

⟨*function* freemeta 152c⟩≡ (159b)

```
static void
freemeta(Meta *m)
{
    free(m->delta);
    free(m);
}
```

⟨*function* loadtree 152d⟩≡ (159b)

```
static int
loadtree(Metavec *v, Objset *has, Hash tree, char *dpath, vlong mtime)
{
    Object *t, *o;
    Dirent *e;
    vlong dh, eh;
    int i, k, r;
    char *p;
```

```
    if(oshas(has, tree))
        return 0;
    if((t = readobject(tree)) == nil)
        return -1;
    if(t->type != GTree){
        fprint(2, "load: %H: not tree\n", t->hash);
        unref(t);
        return 0;
    }
    dh = murmurhash2(dpath, strlen(dpath));
    addmeta(v, has, t, dh, mtime);
    for(i = 0; i < t->tree->nent; i++){
        e = &t->tree->ent[i];
        if(oshas(has, e->h))
            continue;
        if(e->ismod)
            continue;
        k = (e->mode & DMDIR) ? GTree : GBlob;
        o = clearedobject(e->h, k);
        if(k == GTree){
            p = smprint("%s/%s", dpath, e->name);
            r = loadtree(v, has, e->h, p, mtime);
            free(p);
            if(r == -1)
                return -1;
        }else{
            eh = murmurhash2(e->name, strlen(e->name));
            addmeta(v, has, o, dh^eh, mtime);
        }
    }
    unref(t);
    return 0;
}
```

⟨*function* `loadcommit` 153a⟩≡ (159b)

```
static int
loadcommit(Metavec *v, Objset *has, Hash h)
{
    Object *c;
    int r;

    if(osfind(has, h))
        return 0;
    if((c = readobject(h)) == nil)
        return -1;
    if(c->type != GCommit){
        fprint(2, "load: %H: not commit\n", c->hash);
        unref(c);
        return 0;
    }
    addmeta(v, has, c, 0, c->commit->ctime);
    r = loadtree(v, has, c->commit->tree, "", c->commit->ctime);
    unref(c);
    return r;
}
```

⟨*function* `readmeta` 153b⟩≡ (159b)

```
static int
readmeta(Hash *theirs, int ntheirs, Hash *ours, int nours, Meta ***m)
{
```

```
    Object **obj;
    Objset has;
    int i, nobj;
    Metavec v;

    *m = nil;
    osinit(&has);
    v.nmeta = 0;
    v.metasz = 64;
    v.meta = eamalloc(v.metasz, sizeof(Meta*));
    if(findtwixt(theirs, ntheirs, ours, nours, &obj, &nobj) == -1)
        sysfatal("load twixt: %r");

    if(nobj == 0)
        return 0;
    for(i = 0; i < nours; i++)
        if(!hasheq(&ours[i], &Zhash))
            if(loadcommit(nil, &has, ours[i]) == -1)
                goto out;
    for(i = 0; i < nobj; i++)
        if(loadcommit(&v, &has, obj[i]->hash) == -1)
            goto out;
    osclear(&has);
    *m = v.meta;
    return v.nmeta;
out:
    osclear(&has);
    free(v.meta);
    return -1;
}
```

⟨*function* `deltasz` 154a⟩≡ (159b)

```
static int
deltasz(Delta *d, int nd)
{
    int i, sz;
    sz = 32;
    for(i = 0; i < nd; i++)
        sz += d[i].cpy ? 7 : d[i].len + 1;
    return sz;
}
```

⟨*function* `pickdeltas` 154b⟩≡ (159b)

```
static void
pickdeltas(Meta **meta, int nmeta)
{
    Meta *m, *p;
    Object *o;
    Delta *d;
    int i, j, nd, sz, pct, best;

    pct = 0;
    dprint(1, "picking deltas\n");
    if(interactive)
        fprint(2, "deltifying %d objects:   0%%", nmeta);
    qsort(meta, nmeta, sizeof(Meta*), deltaordercmp);
    for(i = 0; i < nmeta; i++){
        m = meta[i];
        pct = showprogress((i*100) / nmeta, pct);
        m->delta = nil;
```

```
            m->ndelta = 0;
            if(m->obj->type == GCommit || m->obj->type == GTag)
                continue;
            if((o = readobject(m->obj->hash)) == nil)
                sysfatal("readobject %H: %r", m->obj->hash);
            dtinit(&m->dtab, o);
            if(i >= 11)
                dtclear(&meta[i-11]->dtab);
            best = o->size;
            for(j = max(0, i - 10); j < i; j++){
                p = meta[j];
                /* long chains make unpacking slow */
                if(p->nchain >= 128 || p->obj->type != o->type)
                    continue;
                d = deltify(o, &p->dtab, &nd);
                sz = deltasz(d, nd);
                if(sz + 32 < best){
                    /*
                     * if we already picked a best delta,
                     * replace it.
                     */
                    free(m->delta);
                    best = sz;
                    m->delta = d;
                    m->ndelta = nd;
                    m->nchain = p->nchain + 1;
                    m->prev = p;
                    m->head = p->head;
                    if(m->head == nil)
                        m->head = p;
                }else
                    free(d);
            }
            unref(o);
        }
        for(i = max(0, nmeta - 10); i < nmeta; i++)
            dtclear(&meta[i]->dtab);
        if(interactive)
            fprint(2, "\b\b\b\b100%%\n");
    }
```

⟨*function* compread 155a⟩≡ (159b)

```
    static int
    compread(void *p, void *dst, int n)
    {
        Buf *b;

        b = p;
        if(n > b->sz - b->off)
            n = b->sz - b->off;
        memcpy(dst, b->data + b->off, n);
        b->off += n;
        return n;
    }
```

⟨*function* compwrite 155b⟩≡ (159b)

```
    static int
    compwrite(void *p, void *buf, int n)
    {
        return hwrite(((Compout *)p)->bfd, buf, n, &((Compout*)p)->st);
    }
```

⟨*function* hcompress 156a⟩≡ (159b)

```
static int
hcompress(Biobuf *bfd, void *buf, int sz, DigestState **st)
{
    int r;
    Buf b ={
        .off=0,
        .data=buf,
        .sz=sz,
    };
    Compout o = {
        .bfd = bfd,
        .st = *st,
    };

    r = deflatezlib(&o, compwrite, &b, compread, 6, 0);
    *st = o.st;
    return r;
}
```

⟨*function* append 156b⟩≡ (159b)

```
static void
append(char **p, int *len, int *sz, void *seg, int nseg)
{
    if(*len + nseg >= *sz){
        while(*len + nseg >= *sz)
            *sz += *sz/2;
        *p = erealloc(*p, *sz);
    }
    memcpy(*p + *len, seg, nseg);
    *len += nseg;
}
```

⟨*function* encodedelta 156c⟩≡ (159b)

```
static int
encodedelta(Meta *m, Object *o, Object *b, void **pp)
{
    char *p, *bp, buf[16];
    int len, sz, n, i, j;
    Delta *d;

    sz = 128;
    len = 0;
    p = emalloc(sz);

    /* base object size */
    buf[0] = b->size & 0x7f;
    n = b->size >> 7;
    for(i = 1; n > 0; i++){
        buf[i - 1] |= 0x80;
        buf[i] = n & 0x7f;
        n >>= 7;
    }
    append(&p, &len, &sz, buf, i);

    /* target object size */
    buf[0] = o->size & 0x7f;
    n = o->size >> 7;
    for(i = 1; n > 0; i++){
        buf[i - 1] |= 0x80;
```

```
        buf[i] = n & 0x7f;
        n >>= 7;
    }
    append(&p, &len, &sz, buf, i);
    for(j = 0; j < m->ndelta; j++){
        d = &m->delta[j];
        if(d->cpy){
            n = d->off;
            bp = buf + 1;
            buf[0] = 0x81;
            buf[1] = 0x00;
            for(i = 0; i < sizeof(buf); i++) {
                buf[0] |= 1<<i;
                *bp++ = n & 0xff;
                n >>= 8;
                if(n == 0)
                    break;
            }

            n = d->len;
            if(n != 0x10000) {
                buf[0] |= 0x1<<4;
                for(i = 0; i < sizeof(buf)-4 && n > 0; i++){
                    buf[0] |= 1<<(i + 4);
                    *bp++ = n & 0xff;
                    n >>= 8;
                }
            }
            append(&p, &len, &sz, buf, bp - buf);
        }else{
            n = 0;
            while(n != d->len){
                buf[0] = (d->len - n < 127) ? d->len - n : 127;
                append(&p, &len, &sz, buf, 1);
                append(&p, &len, &sz, o->data + d->off + n, buf[0]);
                n += buf[0];
            }
        }
    }
    *pp = p;
    return len;
}
```

⟨*function* packhdr 157⟩≡ (159b)

```
static int
packhdr(char *hdr, int ty, int len)
{
    int i;

    hdr[0] = ty << 4;
    hdr[0] |= len & 0xf;
    len >>= 4;
    for(i = 1; len != 0; i++){
        hdr[i-1] |= 0x80;
        hdr[i] = len & 0x7f;
        len >>= 7;
    }
    return i;
}
```

⟨*function* packoff 158a⟩≡ (159b)

```
static int
packoff(char *hdr, vlong off)
{
    int i, j;
    char rbuf[8];

    rbuf[0] = off & 0x7f;
    for(i = 1; (off >>= 7) != 0; i++)
        rbuf[i] = (--off & 0x7f)|0x80;

    j = 0;
    while(i > 0)
        hdr[j++] = rbuf[--i];
    return j;
}
```

⟨*function* genpack 158b⟩≡ (159b)

```
static int
genpack(int fd, Meta **meta, int nmeta, Hash *h, int odelta)
{
    int i, nh, nd, res, pct, ret;
    DigestState *st;
    Biobuf *bfd;
    Meta *m;
    Object *o, *po, *b;
    char *p, buf[32];

    st = nil;
    ret = -1;
    pct = 0;
    dprint(1, "generating pack\n");
    if((fd = dup(fd, -1)) == -1)
        return -1;
    if((bfd = Bfdopen(fd, OWRITE)) == nil)
        return -1;
    if(hwrite(bfd, "PACK", 4, &st) == -1)
        return -1;
    PUTBE32(buf, 2);
    if(hwrite(bfd, buf, 4, &st) == -1)
        return -1;
    PUTBE32(buf, nmeta);
    if(hwrite(bfd, buf, 4, &st) == -1)
        return -1;
    qsort(meta, nmeta, sizeof(Meta*), writeordercmp);
    if(interactive)
        fprint(2, "writing %d objects:   0%%", nmeta);
    for(i = 0; i < nmeta; i++){
        pct = showprogress((i*100)/nmeta, pct);
        m = meta[i];
        m->off = Boffset(bfd);
        if(m->off == -1)
            goto error;
        if((o = readobject(m->obj->hash)) == nil)
            return -1;
        if(m->delta == nil){
            nh = packhdr(buf, o->type, o->size);
            if(hwrite(bfd, buf, nh, &st) == -1)
                goto error;
            if(hcompress(bfd, o->data, o->size, &st) == -1)
```

```
                goto error;
            }else{
                if((b = readobject(m->prev->obj->hash)) == nil)
                    goto error;
                nd = encodedelta(m, o, b, &p);
                unref(b);
                if(odelta && m->prev->off != 0){
                    nh = 0;
                    nh += packhdr(buf, GOdelta, nd);
                    nh += packoff(buf+nh, m->off - m->prev->off);
                    if(hwrite(bfd, buf, nh, &st) == -1)
                        goto error;
                }else{
                    nh = packhdr(buf, GRdelta, nd);
                    po = m->prev->obj;
                    if(hwrite(bfd, buf, nh, &st) == -1)
                        goto error;
                    if(hwrite(bfd, po->hash.h, sizeof(po->hash.h), &st) == -1)
                        goto error;
                }
                res = hcompress(bfd, p, nd, &st);
                free(p);
                if(res == -1)
                    goto error;
            }
            unref(o);
        }
        if(interactive)
            fprint(2, "\b\b\b\b100%%\n");
        sha1(nil, 0, h->h, st);
        if(Bwrite(bfd, h->h, sizeof(h->h)) == -1)
            goto error;
        ret = 0;
    error:
        if(Bterm(bfd) == -1)
            return -1;
        return ret;
    }
```

⟨*function* `writepack` 159a⟩≡ (159b)

```
    int
    writepack(int fd, Hash *theirs, int ntheirs, Hash *ours, int nours, Hash *h)
    {
        Meta **meta;
        int i, r, nmeta;

        if((nmeta = readmeta(theirs, ntheirs, ours, nours, &meta)) == -1)
            return -1;
        pickdeltas(meta, nmeta);
        r = genpack(fd, meta, nmeta, h, 0);
        for(i = 0; i < nmeta; i++)
            freemeta(meta[i]);
        free(meta);
        return r;
    }
```

⟨`git9/pack.c` 159b⟩≡

```
    #include <u.h>
    #include <libc.h>
    #include <ctype.h>
```

```
#include "git.h"

typedef struct Buf  Buf;
typedef struct Metavec  Metavec;
typedef struct Meta Meta;
typedef struct Compout  Compout;
typedef struct Packf    Packf;
```

⟨*macro* max 129c⟩

```
struct Metavec {
    Meta    **meta;
    int nmeta;
    int metasz;
};

struct Meta {
    Object  *obj;
    vlong   path;
    vlong   mtime;

    /* The best delta we picked */
    Meta    *head;
    Meta    *prev;
    Delta   *delta;
    int ndelta;
    int nchain;

    /* Only used for delta window */
    Dtab    dtab;

    /* Only used for writing offset deltas */
    vlong   off;
};

struct Compout {
    Biobuf *bfd;
    DigestState *st;
};

struct Buf {
    int len;
    int sz;
    int off;
    char *data;
};

struct Packf {
    char    path[128];
    char    *idx;
    vlong   nidx;

    int refs;
    Biobuf  *pack;
    vlong   opentm;
};

static int  readpacked(Biobuf *, Object *, int);
static Object   *readidxobject(Biobuf *, Hash, int);
```

```
Objset objcache;
Object *lruhead;
Object *lrutail;
vlong   ncache;
vlong   cachemax = 128*MiB;
Packf   *packf;
int npackf;
int openpacks;
```

⟨*function* `clear` 130a⟩

⟨*function* `unref` 130b⟩

⟨*function* `ref` 130c⟩

⟨*function* `cache` 130d⟩

⟨*function* `loadpack` 131⟩

⟨*function* `refreshpacks` 132⟩

⟨*function* `openpack` 133a⟩

⟨*function* `closepack` 133b⟩

⟨*function* `crc32` 133c⟩

⟨*function* `bappend` 134⟩

⟨*function* `breadc` 135a⟩

⟨*function* `bdecompress` 135b⟩

⟨*function* `decompress` 135c⟩

⟨*function* `readvint` 135d⟩

⟨*function* `applydelta` 136⟩

⟨*function* `readrdelta` 137a⟩

⟨*function* `readodelta` 137b⟩

⟨*function* `readpacked` 138⟩

⟨*function* `readloose` 139⟩

⟨*function* `hashcmp` 140a⟩

⟨*function* `searchindex` 140b⟩

⟨*function* `scanword` 142⟩

⟨*function* `nextline (git9/pack.c)` 143a⟩

⟨*function* `parseauthor` 143b⟩

⟨*function* `parsecommit` 143c⟩

⟨*function* `parsetree` 144⟩

⟨*function* `parsetag` 145a⟩

⟨*function* `parseobject` 145b⟩

⟨*function* `readidxobject` 146⟩

⟨*function* `expandprefix` 147⟩

⟨*function* `readobject` 148a⟩

⟨*function* `clearedobject` 148b⟩

⟨*function* `objcmp` 148c⟩

⟨*function* `hwrite` 149a⟩

⟨*function* `objectcrc` 149b⟩

⟨*function* `indexpack` 149c⟩

⟨*function* `deltaordercmp` 151⟩

⟨*function* `writeordercmp` 152a⟩

⟨*function* `addmeta` 152b⟩

⟨*function* `freemeta` 152c⟩

⟨*function* `loadtree` 152d⟩

⟨*function* `loadcommit` 153a⟩

⟨*function* `readmeta` 153b⟩

⟨*function* `deltasz` 154a⟩

⟨*function* `pickdeltas` 154b⟩

⟨*function* `compread` 155a⟩

⟨*function* `compwrite` 155b⟩

⟨*function* `hcompress` 156a⟩

⟨*function* `append` 156b⟩

⟨*function* `encodedelta` 156c⟩

⟨*function* `packhdr` 157⟩

⟨*function* `packoff` 158a⟩

⟨*function* `genpack` 158b⟩

⟨*function* `writepack` 159a⟩

git9/proto.c

⟨*constant* Useragent 163a⟩≡ (172c)
```
#define Useragent   "useragent git/2.24.1"
```

⟨*constant* Contenthdr 163b⟩≡ (172c)
```
#define Contenthdr  "headers Content-Type: application/x-git-%s-pack-request"
```

⟨*constant* Accepthdr 163c⟩≡ (172c)
```
#define Accepthdr   "headers Accept: application/x-git-%s-pack-result"
```

⟨*function* matchcap 163d⟩≡ (172c)
```
char *
matchcap(char *s, char *cap, int full)
{
    if(strncmp(s, cap, strlen(cap)) == 0)
        if(!full || strlen(s) == strlen(cap))
            return s + strlen(cap);
    return nil;
}
```

⟨*function* parsecaps 163e⟩≡ (172c)
```
void
parsecaps(char *caps, Conn *c)
{
    char *p, *n, *s, *t;

    for(p = caps; p != nil; p = n){
        n = strchr(p, ' ');
        if(n != nil)
            *n++ = 0;
        if(matchcap(p, "report-status", 1) != nil)
            c->report = 1;
        if(matchcap(p, "multi_ack", 1) != nil)
            c->multiack = 1;
        else if(matchcap(p, "side-band", 1) != nil)
            c->sideband = 1;
        else if(matchcap(p, "side-band-64k", 1) != nil)
            c->sideband64k = 1;
        else if((s = matchcap(p, "symref=", 0)) != nil){
            if((t = strchr(s, ':')) == nil)
                continue;
            *t++ = '\0';
            snprint(c->symfrom, sizeof(c->symfrom), s);
            snprint(c->symto, sizeof(c->symto), t);
        }
    }
}
```

⟨*function* tracepkt 163f⟩≡ (172c)
```
void
tracepkt(int v, char *pfx, char *b, int n)
{
    char *f;
    int o, i;

    if(chattygit < v)
        return;
    o = 0;
    f = emalloc(n*4 + 1);
```

```
    for(i = 0; i < n; i++){
        if(isprint(b[i])){
            f[o++] = b[i];
            continue;
        }
        f[o++] = '\\';
        switch(b[i]){
        case '\\':  f[o++] = '\\';  break;
        case '\n':  f[o++] = 'n';   break;
        case '\r':  f[o++] = 'r';   break;
        case '\v':  f[o++] = 'v';   break;
        case '\0':  f[o++] = '0';   break;
        default:
            f[o++] = 'x';
            f[o++] = "0123456789abcdef"[(b[i]>>4)&0xf];
            f[o++] = "0123456789abcdef"[(b[i]>>0)&0xf];
            break;
        }
    }
    f[o] = '\0';
    fprint(2, "%s %04x:\t%s\n", pfx, n, f);
    free(f);
}
```

⟨*function* readpkt 164⟩≡ (172c)

```
int
readpkt(Conn *c, char *buf, int nbuf)
{
    char len[5];
    char *e;
    int n;

    if(readn(c->rfd, len, 4) != 4){
        werrstr("pktline: short read from transport");
        return -1;
    }
    len[4] = 0;
    n = strtol(len, &e, 16);
    if(n == 0){
        dprint(1, "=r=> 0000\n");
        return 0;
    }
    if(e != len + 4 || n <= 4)
        sysfatal("pktline: bad length '%s'", len);
    n  -= 4;
    if(n >= nbuf)
        abort();//sysfatal("pktline: undersize buffer");
    if(readn(c->rfd, buf, n) != n)
        return -1;
    if(n > 4 && strncmp(buf, "ERR ", 4) == 0){
        if((e = strrchr(buf, '\n')) != nil)
            *e = '\0';
        werrstr("%s", buf + 4);
        return -1;
    }
    buf[n] = 0;
    tracepkt(1, "=r=>", buf, n);
    return n;
}
```

⟨*function* `writepkt` 165a⟩≡ (172c)

```
int
writepkt(Conn *c, char *buf, int nbuf)
{
    char len[5];


    snprint(len, sizeof(len), "%04x", nbuf + 4);
    if(write(c->wfd, len, 4) != 4)
        return -1;
    if(write(c->wfd, buf, nbuf) != nbuf)
        return -1;
    tracepkt(1, "<=w=", buf, nbuf);
    return 0;
}
```

⟨*function* `fmtpkt` 165b⟩≡ (172c)

```
int
fmtpkt(Conn *c, char *fmt, ...)
{
    char pkt[Pktmax];
    va_list ap;
    int n;

    va_start(ap, fmt);
    n = vsnprint(pkt, sizeof(pkt), fmt, ap);
    n = writepkt(c, pkt, n);
    va_end(ap);
    return n;
}
```

⟨*function* `flushpkt` 165c⟩≡ (172c)

```
int
flushpkt(Conn *c)
{
    dprint(1, "<=w= 0000\n");
    return write(c->wfd, "0000", 4);
}
```

⟨*function* `grab` 165d⟩≡ (172c)

```
static void
grab(char *dst, int n, char *p, char *e)
{
    int l;

    l = e - p;
    if(l >= n)
        sysfatal("overlong component");
    memcpy(dst, p, l);
    dst[l] = 0;
}
```

⟨*function* `parseuri` 165e⟩≡ (172c)

```
static int
parseuri(char *uri, char *proto, char *host, char *port, char *path)
{
    char *s, *p, *q;
    int hasport;

    print("uri: \"%s\"\n", uri);
```

```
    p = strstr(uri, "://");
    if(p == nil)
        snprint(proto, Nproto, "ssh");
    else if(strncmp(uri, "git+", 4) == 0)
        grab(proto, Nproto, uri + 4, p);
    else
        grab(proto, Nproto, uri, p);
    *port = 0;
    hasport = 1;
    if(strcmp(proto, "git") == 0)
        snprint(port, Nport, "9418");
    else if(strncmp(proto, "https", 5) == 0)
        snprint(port, Nport, "443");
    else if(strncmp(proto, "http", 4) == 0)
        snprint(port, Nport, "80");
    else if(strncmp(proto, "hjgit", 5) == 0)
        snprint(port, Nport, "17021");
    else if(strncmp(proto, "gits", 5) == 0)
        snprint(port, Nport, "9419");
    else
        hasport = 0;
    s = (p != nil) ? p + 3 : uri;
    p = nil;
    if(!hasport){
        p = strstr(s, ":");
        if(p != nil)
            p++;
    }
    if(p == nil)
        p = strstr(s, "/");
    if(p == nil || strlen(p) == 1){
        werrstr("missing path");
        return -1;
    }

    q = memchr(s, ':', p - s);
    if(q){
        grab(host, Nhost, s, q);
        grab(port, Nport, q + 1, p);
    }else{
        grab(host, Nhost, s, p);
    }

    snprint(path, Npath, "%s", p);
    return 0;
}
```

⟨*function* webclone 166⟩≡ (172c)

```
static int
webclone(Conn *c, char *url)
{
    char buf[16];
    int n, conn;

    if((c->cfd = open("/mnt/web/clone", ORDWR)) < 0)
        goto err;
    if((n = read(c->cfd, buf, sizeof(buf)-1)) == -1)
        goto err;
    buf[n] = 0;
```

```
    conn = atoi(buf);

    /* github will behave differently based on useragent */
    if(write(c->cfd, Useragent, sizeof(Useragent)) == -1)
        return -1;
    dprint(1, "open url %s\n", url);
    if(fprint(c->cfd, "url %s", url) == -1)
        goto err;
    free(c->dir);
    c->dir = smprint("/mnt/web/%d", conn);
    return 0;
err:
    if(c->cfd != -1)
        close(c->cfd);
    return -1;
}
```

⟨*function* webopen 167a⟩≡ (172c)

```
static int
webopen(Conn *c, char *file, int mode)
{
    char path[128];
    int fd;

    snprint(path, sizeof(path), "%s/%s", c->dir, file);
    if((fd = open(path, mode)) == -1)
        return -1;
    return fd;
}
```

⟨*function* issmarthttp 167b⟩≡ (172c)

```
static int
issmarthttp(Conn *c, char *direction)
{
    char buf[Pktmax+1], svc[128];
    int fd, n;

    if((fd = webopen(c, "contenttype", OREAD)) == -1)
        return -1;
    n = readn(fd, buf, sizeof(buf) - 1);
    close(fd);
    if(n == -1)
        return -1;
    buf[n] = '\0';
    snprint(svc, sizeof(svc), "application/x-git-%s-pack-advertisement", direction);
    if(strcmp(svc, buf) != 0){
        werrstr("dumb http protocol not supported");
        return -1;
    }

    if((n = readpkt(c, buf, sizeof(buf))) == -1)
        sysfatal("http read: %r");
    buf[n] = 0;
    snprint(svc, sizeof(svc), "# service=git-%s-pack\n", direction);
    if(strncmp(svc, buf, n) != 0){
        werrstr("invalid initial packet line");
        return -1;
    }
    if(readpkt(c, buf, sizeof(buf)) != 0){
        werrstr("protocol garble: expected flushpkt");
```

```
        return -1;
    }
    return 0;
}
```

⟨*function* dialhttp 168a⟩≡ (172c)

```
static int
dialhttp(Conn *c, char *host, char *port, char *path, char *direction)
{
    char *geturl, *suff, *hsep, *psep, *isep;

    suff = "";
    hsep = "";
    psep = "";
    isep = "";
    if(port && strlen(port) != 0)
        hsep = ":";
    if(path && path[0] != '/')
        psep = "/";
    if(path && path[0] && path[strlen(path)-1] != '/')
        isep = "/";
    memset(c, 0, sizeof(*c));
    geturl = smprint("https://%s%s%s%s%s%s%sinfo/refs?service=git-%s-pack",
        host, hsep, port, psep, path, suff, isep, direction);
    c->type = ConnHttp;
    c->url = smprint("https://%s%s%s%s%s%s%sgit-%s-pack",
        host, hsep, port, psep, path, suff, isep, direction);
    c->cfd = webclone(c, geturl);
    free(geturl);
    if(c->cfd == -1)
        return -1;
    c->rfd = webopen(c, "body", OREAD);
    c->wfd = -1;
    if(c->rfd == -1)
        return -1;
    if(issmarthttp(c, direction) == -1)
        return -1;
    c->direction = estrdup(direction);
    return 0;
}
```

⟨*function* dialssh 168b⟩≡ (172c)

```
static int
dialssh(Conn *c, char *host, char *, char *path, char *direction)
{
    int pid, pfd[2];
    char cmd[64];

    if(pipe(pfd) == -1)
        sysfatal("unable to open pipe: %r");
    pid = fork();
    if(pid == -1)
        sysfatal("unable to fork");
    if(pid == 0){
        close(pfd[1]);
        dup(pfd[0], 0);
        dup(pfd[0], 1);
        snprint(cmd, sizeof(cmd), "git-%s-pack", direction);
        dprint(1, "exec ssh '%s' '%s' %s\n", host, cmd, path);
        execl("/bin/ssh", "ssh", host, cmd, path, nil);
```

```
        sysfatal("exec: %r");
    }
    close(pfd[0]);
    c->type = ConnSsh;
    c->rfd = pfd[1];
    c->wfd = dup(pfd[1], -1);
    return 0;
}
```

⟨*function* `githandshake` 169a⟩≡ (172c)

```
static int
githandshake(Conn *c, char *host, char *path, char *direction)
{
    char *p, *e, cmd[512];

    p = cmd;
    e = cmd + sizeof(cmd);
    p = seprint(p, e - 1, "git-%s-pack %s", direction, path);
    if(host != nil)
        p = seprint(p + 1, e, "host=%s", host);
    if(writepkt(c, cmd, p - cmd + 1) == -1){
        fprint(2, "failed to write message\n");
        closeconn(c);
        return -1;
    }
    return 0;
}
```

⟨*function* `dialhjgit` 169b⟩≡ (172c)

```
static int
dialhjgit(Conn *c, char *host, char *port, char *path, char *direction, int auth)
{
    char *ds;
    int pid, pfd[2];

    if((ds = netmkaddr(host, "tcp", port)) == nil)
        return -1;
    if(pipe(pfd) == -1)
        sysfatal("unable to open pipe: %r");
    pid = fork();
    if(pid == -1)
        sysfatal("unable to fork");
    if(pid == 0){
        close(pfd[1]);
        dup(pfd[0], 0);
        dup(pfd[0], 1);
        dprint(1, "exec tlsclient -a %s\n", ds);
        if(auth)
            execl("/bin/tlsclient", "tlsclient", "-a", ds, nil);
        else
            execl("/bin/tlsclient", "tlsclient", ds, nil);
        sysfatal("exec: %r");
    }
    close(pfd[0]);
    c->type = ConnGit9;
    c->rfd = pfd[1];
    c->wfd = dup(pfd[1], -1);
    return githandshake(c, host, path, direction);
}
```

⟨*function* initconn 170a⟩≡ (172c)
```
void
initconn(Conn *c, int rd, int wr)
{
    c->type = ConnGit;
    c->rfd = rd;
    c->wfd = wr;
}
```

⟨*function* dialgit 170b⟩≡ (172c)
```
static int
dialgit(Conn *c, char *host, char *port, char *path, char *direction)
{
    char *ds;
    int fd;

    if((ds = netmkaddr(host, "tcp", port)) == nil)
        return -1;
    dprint(1, "dial %s git-%s-pack %s\n", ds, direction, path);
    fd = dial(ds, nil, nil, nil);
    if(fd == -1)
        return -1;
    c->type = ConnGit;
    c->rfd = fd;
    c->wfd = dup(fd, -1);
    return githandshake(c, host, path, direction);
}
```

⟨*function* servelocal 170c⟩≡ (172c)
```
static int
servelocal(Conn *c, char *path, char *direction)
{
    int pid, pfd[2];

    if(pipe(pfd) == -1)
        sysfatal("unable to open pipe: %r");
    pid = fork();
    if(pid == -1)
        sysfatal("unable to fork");
    if(pid == 0){
        close(pfd[1]);
        dup(pfd[0], 0);
        dup(pfd[0], 1);
        execl("/bin/git/serve", "serve", "-w", nil);
        sysfatal("exec: %r");
    }
    close(pfd[0]);
    c->type = ConnGit;
    c->rfd = pfd[1];
    c->wfd = dup(pfd[1], -1);
    return githandshake(c, nil, path, direction);
}
```

⟨*function* localrepo 170d⟩≡ (172c)
```
static int
localrepo(char *uri, char *path, int npath)
{
    int fd;
    snprint(path, npath, "%s/.git/../", uri);
    fd = open(path, OREAD);
```

```
    if(fd < 0)
        return -1;
    if(fd2path(fd, path, npath) != 0){
        close(fd);
        return -1;
    }
    close(fd);
    return 0;
}
```

⟨*function* gitconnect 171a⟩≡ (172c)

```
int
gitconnect(Conn *c, char *uri, char *direction)
{
    char proto[Nproto], host[Nhost], port[Nport], path[Npath];

    memset(c, 0, sizeof(Conn));
    c->rfd = c->wfd = c->cfd = -1;

    if(localrepo(uri, path, sizeof(path)) == 0)
        return servelocal(c, path, direction);

    if(parseuri(uri, proto, host, port, path) == -1){
        werrstr("bad uri %s", uri);
        return -1;
    }
    if(strcmp(proto, "ssh") == 0)
        return dialssh(c, host, port, path, direction);
    else if(strcmp(proto, "git") == 0)
        return dialgit(c, host, port, path, direction);
    else if(strcmp(proto, "hjgit") == 0)
        return dialhjgit(c, host, port, path, direction, 1);
    else if(strcmp(proto, "gits") == 0)
        return dialhjgit(c, host, port, path, direction, 0);
    else if(strcmp(proto, "http") == 0 || strcmp(proto, "https") == 0)
        return dialhttp(c, host, port, path, direction);
    werrstr("unknown protocol %s", proto);
    return -1;
}
```

⟨*function* writephase 171b⟩≡ (172c)

```
int
writephase(Conn *c)
{
    char hdr[128];
    int n;

    dprint(1, "start write phase\n");
    if(c->type != ConnHttp)
        return 0;

    if(c->wfd != -1)
        close(c->wfd);
    if(c->cfd != -1)
        close(c->cfd);
    if((c->cfd = webclone(c, c->url)) == -1)
        return -1;
    n = snprint(hdr, sizeof(hdr), Contenthdr, c->direction);
    if(write(c->cfd, hdr, n) == -1)
        return -1;
```

```
    n = snprint(hdr, sizeof(hdr), Accepthdr, c->direction);
    if(write(c->cfd, hdr, n) == -1)
        return -1;
    if((c->wfd = webopen(c, "postbody", OWRITE)) == -1)
        return -1;
    c->rfd = -1;
    return 0;
}
```

⟨*function* readphase 172a⟩≡ (172c)

```
int
readphase(Conn *c)
{
    dprint(1, "start read phase\n");
    if(c->type != ConnHttp)
        return 0;
    if(close(c->wfd) == -1)
        return -1;
    if((c->rfd = webopen(c, "body", OREAD)) == -1)
        return -1;
    c->wfd = -1;
    return 0;
}
```

⟨*function* closeconn 172b⟩≡ (172c)

```
void
closeconn(Conn *c)
{
    close(c->rfd);
    close(c->wfd);
    switch(c->type){
    case ConnGit:
        break;
    case ConnGit9:
    case ConnSsh:
        free(wait());
        break;
    case ConnHttp:
        close(c->cfd);
        break;
    }
}
```

⟨git9/proto.c 172c⟩≡

```
#include <u.h>
#include <libc.h>
#include <ctype.h>

#include "git.h"

⟨constant Useragent 163a⟩
⟨constant Contenthdr 163b⟩
⟨constant Accepthdr 163c⟩

enum {
    Nproto  = 16,
    Nport   = 16,
    Nhost   = 256,
    Npath   = 128,
    Nbranch = 32,
```

```
};
```

⟨*function* `matchcap` 163d⟩

⟨*function* `parsecaps` 163e⟩

⟨*function* `tracepkt` 163f⟩

⟨*function* `readpkt` 164⟩

⟨*function* `writepkt` 165a⟩

⟨*function* `fmtpkt` 165b⟩

⟨*function* `flushpkt` 165c⟩

⟨*function* `grab` 165d⟩

⟨*function* `parseuri` 165e⟩

⟨*function* `webclone` 166⟩

⟨*function* `webopen` 167a⟩

⟨*function* `issmarthttp` 167b⟩

⟨*function* `dialhttp` 168a⟩

⟨*function* `dialssh` 168b⟩

⟨*function* `githandshake` 169a⟩

⟨*function* `dialhjgit` 169b⟩

⟨*function* `initconn` 170a⟩

⟨*function* `dialgit` 170b⟩

⟨*function* `servelocal` 170c⟩

⟨*function* `localrepo` 170d⟩

⟨*function* `gitconnect` 171a⟩

⟨*function* `writephase` 171b⟩

⟨*function* `readphase` 172a⟩

⟨*function* `closeconn` 172b⟩

git9/query.c

⟨*function* `Pfmt` 173⟩≡ (176c)

```
int
Pfmt(Fmt *f)
{
    int i, n;
```

```
    n = 0;
    for(i = 0; i < npath; i++)
        n += fmtprint(f, "%s/", path[i]);
    return n;
}
```

⟨*function* showdir 174a⟩≡ (176c)

```
void
showdir(Hash dh, char *dname, char m)
{
    Dirent *p, *e;
    Object *d;


    path[npath++] = dname;
    if((d = readobject(dh)) == nil)
        sysfatal("bad hash %H", dh);
    assert(d->type == GTree);
    p = d->tree->ent;
    e = p + d->tree->nent;
    for(; p != e; p++){
        if(p->ismod)
            continue;
        if(p->mode & DMDIR)
            showdir(p->h, p->name, m);
        else
            print("%c %P%s\n", m, p->name);
    }
    print("%c %P\n", m);
    unref(d);
    npath--;
}
```

⟨*function* show (git9/query.c) 174b⟩≡ (176c)

```
void
show(Dirent *e, char m)
{
    if(e->mode & DMDIR)
        showdir(e->h, e->name, m);
    else
        print("%c %P%s\n", m, e->name);
}
```

⟨*function* difftrees 174c⟩≡ (176c)

```
void
difftrees(Object *a, Object *b)
{
    Dirent *ap, *bp, *ae, *be;
    int c;

    ap = ae = nil;
    bp = be = nil;
    if(a != nil){
        if(a->type != GTree)
            return;
        ap = a->tree->ent;
        ae = ap + a->tree->nent;
    }
    if(b != nil){
        if(b->type != GTree)
```

```
            return;
        bp = b->tree->ent;
        be = bp + b->tree->nent;
    }
    while(ap != ae && bp != be){
        c = entcmp(ap, bp);
        if(c == 0){
            if(ap->mode == bp->mode && hasheq(&ap->h, &bp->h))
                goto next;
            if(ap->mode != bp->mode)
                print("! %P%s\n", ap->name);
            else if(!(ap->mode & DMDIR) || !(bp->mode & DMDIR))
                print("@ %P%s\n", ap->name);
            if((ap->mode & DMDIR) && (bp->mode & DMDIR)){
                if(npath >= nelem(path))
                    sysfatal("path too deep");
                path[npath++] = ap->name;
                if((a = readobject(ap->h)) == nil)
                    sysfatal("bad hash %H", ap->h);
                if((b = readobject(bp->h)) == nil)
                    sysfatal("bad hash %H", bp->h);
                difftrees(a, b);
                unref(a);
                unref(b);
                npath--;
            }
next:
            ap++;
            bp++;
        }else if(c < 0) {
            show(ap, '-');
            ap++;
        }else if(c > 0){
            show(bp, '+');
            bp++;
        }
    }
    for(; ap != ae; ap++)
        show(ap, '-');
    for(; bp != be; bp++)
        show(bp, '+');
}
```

⟨*function* `diffcommits` 175⟩≡ (176c)

```
void
diffcommits(Hash ah, Hash bh)
{
    Object *a, *b, *at, *bt;

    at = nil;
    bt = nil;
    if(!hasheq(&ah, &Zhash) && (a = readobject(ah)) != nil){
        if(a->type != GCommit)
            sysfatal("not commit: %H", ah);
        if((at = readobject(a->commit->tree)) == nil)
            sysfatal("bad hash %H", a->commit->tree);
        unref(a);
    }
    if(!hasheq(&bh, &Zhash) && (b = readobject(bh)) != nil){
        if(b->type != GCommit)
```

```
            sysfatal("not commit: %H", ah);
        if((bt = readobject(b->commit->tree)) == nil)
            sysfatal("bad hash %H", b->commit->tree);
        unref(b);
    }
    difftrees(at, bt);
    unref(at);
    unref(bt);
}
```

⟨*function* usage (git9/query.c) 176a⟩≡ (176c)

```
void
usage(void)
{
    fprint(2, "usage: %s [-pcr] query\n", argv0);
    exits("usage");
}
```

⟨*function* main (git9/query.c) 176b⟩≡ (176c)

```
void
main(int argc, char **argv)
{
    char *query, repo[512];
    char *p, *e, *objpfx;
    int i, j, n, nrel;
    Hash *h;

    ARGBEGIN{
    case 'd':   chattygit++;    break;
    case 'p':   fullpath++; break;
    case 'c':   changes++;  break;
    case 'r':   reverse ^= 1;   break;
    default:    usage();    break;
    }ARGEND;
```

⟨git9/query.c 176c⟩≡

```
#include <u.h>
#include <libc.h>

#include "git.h"

#pragma varargck    type    "P" void

int fullpath;
int changes;
int reverse;
char *path[128];
int npath;
```

⟨*function* Pfmt 173⟩

⟨*function* showdir 174a⟩

⟨*function* show (git9/query.c) 174b⟩

⟨*function* difftrees 174c⟩

⟨*function* diffcommits 175⟩

⟨*function* usage (git9/query.c) 176a⟩

⟨*function* `main (git9/query.c)` 176b⟩

```
    fmtinstall('P', Pfmt);
    if(argc == 0)
        usage();
    gitinit(repo, sizeof(repo), &nrel);
    if(chdir(repo) == -1)
        sysfatal("chdir: %r");
    if((objpfx = smprint("%s/.git/fs/object/", repo)) == nil)
        sysfatal("smprint: %r");
    for(i = 0, n = 0; i < argc; i++)
        n += strlen(argv[i]) + 1;
    query = emalloc(n+1);
    p = query;
    e = query + n;
    for(i = 0; i < argc; i++)
        p = seprint(p, e, "%s ", argv[i]);
    n = resolverefs(&h, query);
    free(query);
    if(n == -1)
        sysfatal("resolve: %r");
    if(changes){
        for(i = 1; i < n; i++)
            diffcommits(h[0], h[i]);
    }else{
        p = (fullpath ? objpfx : "");
        for(j = 0; j < n; j++)
            print("%s%H\n", p, h[reverse ? n - 1 - j : j]);
    }
    exits(nil);
}
```

git9/ref.c

⟨*function* `eatspace` 177a⟩≡ (186b)

```
void
eatspace(Eval *ev)
{
    while(isspace(ev->p[0]))
        ev->p++;
}
```

⟨*function* `push` 177b⟩≡ (186b)

```
void
push(Eval *ev, Object *o)
{
    if(ev->nstk == ev->stksz){
        ev->stksz = 2*ev->stksz + 1;
        ev->stk = erealloc(ev->stk, ev->stksz*sizeof(Object*));
    }
    ev->stk[ev->nstk++] = o;
}
```

⟨*function* `pop` 177c⟩≡ (186b)

```
Object*
pop(Eval *ev)
{
```

```
    if(ev->nstk == 0)
        sysfatal("stack underflow");
    return ev->stk[--ev->nstk];
}
```

⟨*function* peek 178a⟩≡ (186b)
```
Object*
peek(Eval *ev)
{
    if(ev->nstk == 0)
        sysfatal("stack underflow");
    return ev->stk[ev->nstk - 1];
}
```

⟨*function* isword 178b⟩≡ (186b)
```
int
isword(char e)
{
    return isalnum(e) || e == '/' || e == '-' || e == '_' || e == '.';
}
```

⟨*function* word 178c⟩≡ (186b)
```
int
word(Eval *ev, char *b, int nb)
{
    char *p, *e;
    int n;

    p = ev->p;
    for(e = p; isword(*e) && strncmp(e, "..", 2) != 0; e++)
        /* nothing */;
    /* 1 for nul terminator */
    n = e - p + 1;
    if(n >= nb)
        n = nb;
    snprint(b, n, "%s", p);
    ev->p = e;
    return n > 0;
}
```

⟨*function* take 178d⟩≡ (186b)
```
int
take(Eval *ev, char *m)
{
    int l;

    l = strlen(m);
    if(strncmp(ev->p, m, l) != 0)
        return 0;
    ev->p += l;
    return 1;
}
```

⟨*function* paint 178e⟩≡ (186b)
```
static int
paint(Hash *head, int nhead, Hash *tail, int ntail, Object ***res, int *nres, int mode)
{
    Qelt e;
    Objq objq;
    Objset keep, drop, skip;
```

```
Object *o, *c, **range;
int i, nskip, nrange;

osinit(&keep);
osinit(&drop);
osinit(&skip);
qinit(&objq);
range = nil;
nrange = 0;
nskip = 0;

for(i = 0; i < nhead; i++){
    if(hasheq(&head[i], &Zhash))
        continue;
    if((o = readobject(head[i])) == nil){
        fprint(2, "warning: %H does not point at commit\n", head[i]);
        werrstr("read head %H: %r", head[i]);
        return -1;
    }
    if(o->type != GCommit){
        fprint(2, "warning: %H does not point at commit\n", o->hash);
        unref(o);
        continue;
    }
    dprint(1, "init: keep %H\n", o->hash);
    qput(&objq, o, Keep);
    unref(o);
}
for(i = 0; i < ntail; i++){
    if(hasheq(&tail[i], &Zhash))
        continue;
    if((o = readobject(tail[i])) == nil){
        werrstr("read tail %H: %r", tail[i]);
        return -1;
    }
    if(o->type != GCommit){
        fprint(2, "warning: %H does not point at commit\n", o->hash);
        unref(o);
        continue;
    }
    dprint(1, "init: drop %H\n", o->hash);
    qput(&objq, o, Drop);
    unref(o);
}

dprint(1, "finding twixt commits\n");
while(objq.nheap != 0 && qpop(&objq, &e)){
    if(e.color == Skip)
        nskip--;
    if(oshas(&skip, e.o->hash))
        continue;
    switch(e.color){
    case Keep:
        if(oshas(&keep, e.o->hash))
            continue;
        if(oshas(&drop, e.o->hash))
            e.color = Skip;
        else if(mode == Range){
            range = earealloc(range, nrange+1, sizeof(Object*));
            range[nrange++] = e.o;
```

```
        }
        osadd(&keep, e.o);
        break;
    case Drop:
        if(oshas(&drop, e.o->hash))
            continue;
        if(oshas(&keep, e.o->hash))
            e.color = Skip;
        osadd(&drop, e.o);
        break;
    case Skip:
        osadd(&skip, e.o);
        break;
    }
    o = readobject(e.o->hash);
    if(o->type != GCommit){
        werrstr("not a commit: %H", o->hash);
        goto error;
    }
    for(i = 0; i < o->commit->nparent; i++){
        if((c = readobject(e.o->commit->parent[i])) == nil)
            goto error;
        if(c->type != GCommit){
            fprint(2, "warning: %H does not point at commit\n", c->hash);
            unref(c);
            continue;
        }
        dprint(2, "\tenqueue: %s %H\n", colors[e.color], c->hash);
        qput(&objq, c, e.color);
        unref(c);
        if(e.color == Skip)
            nskip++;
    }
    unref(o);
}
switch(mode){
case Lca:
    dprint(1, "found ancestor\n");
    o = nil;
    for(i = 0; i < keep.sz; i++){
        o = keep.obj[i];
        if(o != nil && oshas(&drop, o->hash) && !oshas(&skip, o->hash))
            break;
    }
    if(i == keep.sz){
        *nres = 0;
        *res = nil;
    }else{
        *nres = 1;
        *res = eamalloc(1, sizeof(Object*));
        (*res)[0] = o;
    }
    break;
case Twixt:
    dprint(1, "found twixt\n");
    *res = eamalloc(keep.nobj, sizeof(Object*));
    *nres = 0;
    for(i = 0; i < keep.sz; i++){
        o = keep.obj[i];
        if(o != nil && !oshas(&drop, o->hash) && !oshas(&skip, o->hash)){
```

```
                (*res)[*nres] = o;
                (*nres)++;
            }
        }
        break;
    case Range:
        dprint(1, "found range\n");
        *res = eamalloc(nrange, sizeof(Object*));
        *nres = 0;
        for(i = nrange - 1; i >= 0; i--){
            o = range[i];
            if(!oshas(&drop, o->hash) && !oshas(&skip, o->hash)){
                (*res)[*nres] = o;
                (*nres)++;
            }
        }
        free(range);
        break;
    }
    osclear(&keep);
    osclear(&drop);
    osclear(&skip);
    return 0;
error:
    dprint(1, "paint error: %r\n");
    free(objq.heap);
    free(range);
    return -1;
}
```

⟨*function* `findtwixt` 181a⟩≡ (186b)

```
int
findtwixt(Hash *head, int nhead, Hash *tail, int ntail, Object ***res, int *nres)
{
    return paint(head, nhead, tail, ntail, res, nres, Twixt);
}
```

⟨*function* `ancestor` 181b⟩≡ (186b)

```
Object*
ancestor(Object *a, Object *b)
{
    Object **o, *r;
    int n;

    if(paint(&a->hash, 1, &b->hash, 1, &o, &n, Lca) == -1 || n == 0)
        return nil;
    r = ref(o[0]);
    free(o);
    return r;
}
```

⟨*function* `lca` 181c⟩≡ (186b)

```
int
lca(Eval *ev)
{
    Object *a, *b, **o;
    int n;

    if(ev->nstk < 2){
        werrstr("ancestor needs 2 objects");
```

```
        return -1;
    }
    n = 0;
    b = pop(ev);
    a = pop(ev);
    paint(&a->hash, 1, &b->hash, 1, &o, &n, Lca);
    if(n == 0)
        return -1;
    push(ev, *o);
    free(o);
    return 0;
}
```

⟨*function* parent 182a⟩≡ (186b)

```
static int
parent(Eval *ev)
{
    Object *o, *p;

    o = pop(ev);
    if(o->type != GCommit){
        werrstr("not a commit: %H", o->hash);
        return -1;
    }
    /* Special case: first commit has no parent. */
    if(o->commit->nparent == 0)
        p = emptydir();
    else if ((p = readobject(o->commit->parent[0])) == nil){
        werrstr("no parent for %H", o->hash);
        return -1;
    }

    push(ev, p);
    return 0;
}
```

⟨*function* range 182b⟩≡ (186b)

```
static int
range(Eval *ev)
{
    Object *a, *b, **o;
    int i, n;

    b = pop(ev);
    a = pop(ev);
    if(hasheq(&b->hash, &Zhash))
        b = &zcommit;
    if(hasheq(&a->hash, &Zhash))
        a = &zcommit;
    if(a->type != GCommit || b->type != GCommit){
        werrstr("non-commit object in range");
        return -1;
    }

    if(paint(&b->hash, 1, &a->hash, 1, &o, &n, Range) == -1)
        return -1;
    for(i = 0; i < n; i++)
        push(ev, o[i]);
    free(o);
    return 0;
}
```

⟨*function* matchpfx 183a⟩≡ (186b)

```
static int
matchpfx(Hash *h, char *ref)
{
    int i, c;
    Hash pfx;
    char *p;

    memset(&pfx, 0, sizeof(Hash));
    for(i = 0, p = ref; *p; p++, i++){
        if((c = charval(*p)) == -1)
            return -1;
        pfx.h[i/2] |= c;
        if((i & 1) == 0)
            pfx.h[i/2] <<= 4;
    }
    return expandprefix(h, pfx, i*4);
}
```

⟨*function* readref (git9/ref.c) 183b⟩≡ (186b)

```
int
readref(Hash *h, char *ref)
{
    static char *try[] = {"", "refs/", "refs/heads/", "refs/remotes/", "refs/tags/", nil};
    char buf[256], s[256], **pfx;
    int r, f, n;

    if((r = hparse(h, ref)) != -1)
        return r;
    if(strcmp(ref, "HEAD") == 0){
        snprint(buf, sizeof(buf), ".git/HEAD");
        if((f = open(buf, OREAD)) == -1)
            return -1;
        if((n = readn(f, s, sizeof(s) - 1))== -1)
            return -1;
        s[n] = 0;
        strip(s);
        r = hparse(h, s);
        goto found;
    }
    for(pfx = try; *pfx; pfx++){
        snprint(buf, sizeof(buf), ".git/%s%s", *pfx, ref);
        if((f = open(buf, OREAD)) == -1)
            continue;
        if((n = readn(f, s, sizeof(s) - 1)) == -1)
            continue;
        s[n] = 0;
        strip(s);
        r = hparse(h, s);
        close(f);
        goto found;
    }
    if((r = matchpfx(h, ref)) != -1)
        return r;
    return -1;

found:
    if(r == -1 && strncmp(s, "ref: ", 5) == 0)
        r = readref(h, s + 5);
    return r;
```

```
}
```

⟨*function* evalpostfix 184a⟩≡ (186b)

```
int
evalpostfix(Eval *ev)
{
    char name[256];
    Object *o;
    Hash h;

    eatspace(ev);
    if(!word(ev, name, sizeof(name))){
        werrstr("expected name in expression");
        return -1;
    }
    if(readref(&h, name) == -1){
        werrstr("invalid ref %s", name);
        return -1;
    }
    if(hasheq(&h, &Zhash))
        o = &zcommit;
    else if((o = readobject(h)) == nil){
        werrstr("invalid ref %s (hash %H)", name, h);
        return -1;
    }
    push(ev, o);

    while(1){
        eatspace(ev);
        switch(ev->p[0]){
        case '^':
        case '~':
            ev->p++;
            if(parent(ev) == -1)
                return -1;
            break;
        case '@':
            ev->p++;
            if(lca(ev) == -1)
                return -1;
            break;
        default:
            goto done;
            break;
        }
    }
done:
    return 0;
}
```

⟨*function* evalexpr 184b⟩≡ (186b)

```
int
evalexpr(Eval *ev, char *ref)
{
    memset(ev, 0, sizeof(*ev));
    ev->str = ref;
    ev->p = ref;

    while(1){
        if(evalpostfix(ev) == -1)
```

```
            return -1;
        if(ev->p[0] == '\0')
            return 0;
        else if(take(ev, ":") || take(ev, "..")){
            if(evalpostfix(ev) == -1)
                return -1;
            if(ev->p[0] != '\0'){
                werrstr("junk at end of expression");
                return -1;
            }
            return range(ev);
        }
    }
}
```

⟨*function* resolverefs 185a⟩≡ (186b)

```
int
resolverefs(Hash **r, char *ref)
{
    Eval ev;
    Hash *h;
    int i;

    if(evalexpr(&ev, ref) == -1){
        free(ev.stk);
        return -1;
    }
    h = eamalloc(ev.nstk, sizeof(Hash));
    for(i = 0; i < ev.nstk; i++)
        h[i] = ev.stk[i]->hash;
    *r = h;
    free(ev.stk);
    return ev.nstk;
}
```

⟨*function* resolveref 185b⟩≡ (186b)

```
int
resolveref(Hash *r, char *ref)
{
    Eval ev;

    if(evalexpr(&ev, ref) == -1){
        free(ev.stk);
        return -1;
    }
    if(ev.nstk != 1){
        werrstr("ambiguous ref expr");
        free(ev.stk);
        return -1;
    }
    *r = ev.stk[0]->hash;
    free(ev.stk);
    return 0;
}
```

⟨*function* readrefdir 185c⟩≡ (186b)

```
int
readrefdir(Hash **refs, char ***names, int *nrefs, char *dpath, char *dname)
{
    Dir *d, *e, *dir;
```

```
    char *path, *name, *sep;
    int ndir;

    if((ndir = slurpdir(dpath, &dir)) == -1)
        return -1;
    sep = (*dname == '\0') ? "" : "/";
    e = dir + ndir;
    for(d = dir; d != e; d++){
        path = smprint("%s/%s", dpath, d->name);
        name = smprint("%s%s%s", dname, sep, d->name);
        if(d->mode & DMDIR) {
            if(readrefdir(refs, names, nrefs, path, name) == -1)
                goto noref;
        }else{
            *refs = erealloc(*refs, (*nrefs + 1)*sizeof(Hash));
            *names = erealloc(*names, (*nrefs + 1)*sizeof(char*));
            if(resolveref(&(*refs)[*nrefs], name) == -1)
                goto noref;
            (*names)[*nrefs] = name;
            *nrefs += 1;
            goto next;
        }
noref:      free(name);
next:       free(path);
    }
    free(dir);
    return 0;
}
```

⟨*function* `listrefs` 186a⟩≡ (186b)

```
int
listrefs(Hash **refs, char ***names)
{
    int nrefs;

    *refs = nil;
    *names = nil;
    nrefs = 0;
    if(readrefdir(refs, names, &nrefs, ".git/refs", "") == -1){
        free(*refs);
        return -1;
    }
    return nrefs;
}
```

⟨git9/ref.c 186b⟩≡

```
#include <u.h>
#include <libc.h>
#include <ctype.h>

#include "git.h"

typedef struct Eval Eval;

enum {
    Blank,
    Keep,
    Drop,
    Skip,
};
```

```
enum {
    Lca,
    Twixt,
    Range,
};

struct Eval {
    char    *str;
    char    *p;
    Object  **stk;
    int nstk;
    int stksz;
};

static char *colors[] = {
[Keep] "keep",
[Drop] "drop",
[Blank] "blank",
[Skip] "skip",
};

static Object zcommit = {
    .type=GCommit
};
```

⟨*function* eatspace 177a⟩

⟨*function* push 177b⟩

⟨*function* pop 177c⟩

⟨*function* peek 178a⟩

⟨*function* isword 178b⟩

⟨*function* word 178c⟩

⟨*function* take 178d⟩

⟨*function* paint 178e⟩

⟨*function* findtwixt 181a⟩

⟨*function* ancestor 181b⟩

⟨*function* lca 181c⟩

⟨*function* parent 182a⟩

⟨*function* range 182b⟩

⟨*function* matchpfx 183a⟩

⟨*function* readref (git9/ref.c) 183b⟩

⟨*function* evalpostfix 184a⟩

⟨*function* evalexpr 184b⟩

⟨*function* `resolverefs` 185a⟩

⟨*function* `resolveref` 185b⟩

⟨*function* `readrefdir` 185c⟩

⟨*function* `listrefs` 186a⟩

git9/repack.c

⟨*macro* `TMPPATH` 188a⟩≡ (189a)
```
#define TMPPATH(suff) (".git/objects/pack/repack."suff)
```

⟨*function* `cleanup` 188b⟩≡ (189a)
```
int
cleanup(Hash h)
{
    char newpfx[42], dpath[256], fpath[256];
    int i, j, nd;
    Dir *d;

    snprint(newpfx, sizeof(newpfx), "%H.", h);
    for(i = 0; i < 256; i++){
        snprint(dpath, sizeof(dpath), ".git/objects/%02x", i);
        if((nd = slurpdir(dpath, &d)) == -1)
            continue;
        for(j = 0; j < nd; j++){
            snprint(fpath, sizeof(fpath), ".git/objects/%02x/%s", i, d[j].name);
            remove(fpath);
        }
        remove(dpath);
        free(d);
    }
    snprint(dpath, sizeof(dpath), ".git/objects/pack");
    if((nd = slurpdir(dpath, &d)) == -1)
        return -1;
    for(i = 0; i < nd; i++){
        if(strncmp(d[i].name, newpfx, strlen(newpfx)) == 0)
            continue;
        snprint(fpath, sizeof(fpath), ".git/objects/pack/%s", d[i].name);
        remove(fpath);
    }
    return 0;
}
```

⟨*function* `usage (git9/repack.c)` 188c⟩≡ (189a)
```
void
usage(void)
{
    fprint(2, "usage: %s [-d]\n", argv0);
    exits("usage");
}
```

⟨*function* `main (git9/repack.c)` 188d⟩≡ (189a)
```
void
main(int argc, char **argv)
{
    char path[128], **names;
    int fd, nrefs;
```

```
    Hash *refs, h;
    Dir rn;

    ARGBEGIN{
    case 'd':
        chattygit++;
        break;
    default:
        usage();
    }ARGEND;
```

⟨git9/repack.c 189a⟩≡

```
#include <u.h>
#include <libc.h>

#include "git.h"

⟨macro TMPPATH 188a⟩

⟨function cleanup 188b⟩

⟨function usage (git9/repack.c) 188c⟩

⟨function main (git9/repack.c) 188d⟩

    gitinit(nil, 0, nil);
    refs = nil;
    if((nrefs = listrefs(&refs, &names)) == -1)
        sysfatal("load refs: %r");
    if((fd = create(TMPPATH("pack.tmp"), OWRITE, 0644)) == -1)
        sysfatal("open %s: %r", TMPPATH("pack.tmp"));
    if(writepack(fd, refs, nrefs, nil, 0, &h) == -1)
        sysfatal("writepack: %r");
    if(indexpack(TMPPATH("pack.tmp"), TMPPATH("idx.tmp"), h) == -1)
        sysfatal("indexpack: %r");
    close(fd);

    nulldir(&rn);
    rn.name = path;
    snprint(path, sizeof(path), "%H.pack", h);
    if(dirwstat(TMPPATH("pack.tmp"), &rn) == -1)
        sysfatal("rename pack: %r");
    snprint(path, sizeof(path), "%H.idx", h);
    if(dirwstat(TMPPATH("idx.tmp"), &rn) == -1)
        sysfatal("rename pack: %r");
    if(cleanup(h) == -1)
        sysfatal("cleanup: %r");
    exits(nil);
}
```

git9/save.c

⟨function gitmode 189b⟩≡ (196)

```
int
gitmode(Dirent *e)
{
    if(e->islink)
        return 0120000;
    else if(e->ismod)
```

```
        return 0160000;
    else if(e->mode & DMDIR)
        return 0040000;
    else if(e->mode & 0100)
        return 0100755;
    else
        return 0100644;
}
```

⟨*function* namecmp 190a⟩≡ (196)

```
int
namecmp(void *pa, void *pb)
{
    return strcmp(*(char**)pa, *(char**)pb);
}
```

⟨*function* idxcmp 190b⟩≡ (196)

```
int
idxcmp(void *pa, void *pb)
{
    Idxent *a, *b;
    int c;

    a = (Idxent*)pa;
    b = (Idxent*)pb;
    if((c = strcmp(a->path, b->path)) != 0)
        return c;
    assert(a->order != b->order);
    return a-> order < b->order ? -1 : 1;
}
```

⟨*function* bwrite 190c⟩≡ (196)

```
static int
bwrite(void *p, void *buf, int nbuf)
{
    return Bwrite(p, buf, nbuf);
}
```

⟨*function* objbytes 190d⟩≡ (196)

```
static int
objbytes(void *p, void *buf, int nbuf)
{
    Objbuf *b;
    int r, n, o;
    char *s;

    b = p;
    n = 0;
    if(b->off < b->nhdr){
        r = b->nhdr - b->off;
        r = (nbuf < r) ? nbuf : r;
        memcpy(buf, b->hdr, r);
        b->off += r;
        nbuf -= r;
        n += r;
    }
    if(b->off < b->ndat + b->nhdr){
        s = buf;
        o = b->off - b->nhdr;
        r = b->ndat - o;
```

```
        r = (nbuf < r) ? nbuf : r;
        memcpy(s + n, b->dat + o, r);
        b->off += r;
        n += r;
    }
    return n;
}
```

⟨*function* writeobj 191a⟩≡ (196)

```
void
writeobj(Hash *h, char *hdr, int nhdr, char *dat, int ndat)
{
    Objbuf b = {.off=0, .hdr=hdr, .nhdr=nhdr, .dat=dat, .ndat=ndat};
    char s[64], o[256];
    SHA1state *st;
    Biobuf *f;
    int fd;

    st = sha1((uchar*)hdr, nhdr, nil, nil);
    st = sha1((uchar*)dat, ndat, nil, st);
    sha1(nil, 0, h->h, st);

    snprint(s, sizeof(s), "%H", *h);
    fd = create(".git/objects", OREAD, DMDIR|0755);
    close(fd);
    snprint(o, sizeof(o), ".git/objects/%c%c", s[0], s[1]);
    fd = create(o, OREAD, DMDIR | 0755);
    close(fd);
    snprint(o, sizeof(o), ".git/objects/%c%c/%s", s[0], s[1], s + 2);
    if(readobject(*h) == nil){
        if((f = Bopen(o, OWRITE)) == nil)
            sysfatal("could not open %s: %r", o);
        if(deflatezlib(f, bwrite, &b, objbytes, 9, 0) == -1)
            sysfatal("could not write %s: %r", o);
        Bterm(f);
    }
}
```

⟨*function* writetree 191b⟩≡ (196)

```
int
writetree(Dirent *ent, int nent, Hash *h)
{
    char *t, *txt, *etxt, hdr[128];
    int nhdr, n;
    Dirent *d, *p;

    t = emalloc((16+256+20) * nent);
    txt = t;
    etxt = t + (16+256+20) * nent;

    /* sqeeze out deleted entries */
    n = 0;
    p = ent;
    for(d = ent; d != ent + nent; d++)
        if(d->name)
            p[n++] = *d;
    nent = n;

    qsort(ent, nent, sizeof(Dirent), entcmp);
    for(d = ent; d != ent + nent; d++){
```

```
        if(strlen(d->name) >= 255)
            sysfatal("overly long filename: %s", d->name);
        t = seprint(t, etxt, "%o %s", gitmode(d), d->name) + 1;
        memcpy(t, d->h.h, sizeof(d->h.h));
        t += sizeof(d->h.h);
    }
    nhdr = snprint(hdr, sizeof(hdr), "%T %lld", GTree, (vlong)(t - txt)) + 1;
    writeobj(h, hdr, nhdr, txt, t - txt);
    free(txt);
    return nent;
}
```

⟨*function* blobify 192a⟩≡ (196)

```
void
blobify(Dir *d, char *path, int *mode, Hash *bh)
{
    char h[64], *buf;
    int f, nh;

    if((d->mode & DMDIR) != 0)
        sysfatal("not file: %s", path);
    *mode = d->mode;
    nh = snprint(h, sizeof(h), "%T %lld", GBlob, d->length) + 1;
    if((f = open(path, OREAD)) == -1)
        sysfatal("could not open %s: %r", path);
    buf = emalloc(d->length);
    if(readn(f, buf, d->length) != d->length)
        sysfatal("could not read blob %s: %r", path);
    writeobj(bh, h, nh, buf, d->length);
    free(buf);
    close(f);
}
```

⟨*function* tracked 192b⟩≡ (196)

```
int
tracked(char *path)
{
    int r, lo, hi, mid;

    lo = 0;
    hi = nidx-1;
    while(lo <= hi){
        mid = (hi + lo) / 2;
        r = strcmp(path, idx[mid].path);
        if(r < 0)
            hi = mid-1;
        else if(r > 0)
            lo = mid+1;
        else
            return idx[mid].state != 'R';
    }
    return 0;
}
```

⟨*function* dirent 192c⟩≡ (196)

```
Dirent*
dirent(Dirent **ent, int *nent, char *name)
{
    Dirent *d;
```

```
    assert(strchr(name, '/') == nil);

    for(d = *ent; d != *ent + *nent; d++)
        if(d->name && strcmp(d->name, name) == 0)
            return d;
    *nent += 1;
    *ent = erealloc(*ent, *nent * sizeof(Dirent));
    d = *ent + (*nent - 1);
    memset(d, 0, sizeof(*d));
    d->name = estrdup(name);
    return d;
}
```

⟨*function* treeify 193⟩≡ (196)

```
int
treeify(Object *t, char **path, char **epath, int off, Hash *h)
{
    int nent, ne, slash, isdir;
    char *s, **p, **ep;
    Dirent *e, *ent;
    Object *o;
    Dir *d;

    nent = t->tree->nent;
    ent = eamalloc(nent, sizeof(*ent));
    memcpy(ent, t->tree->ent, nent*sizeof(*ent));
    for(p = path; p != epath; p = ep){
        s = *p;

        /*
         * paths have been normalized already,
         * no leading or double-slashes allowed.
         */
        assert(off <= strlen(s));
        assert(off == 0 || s[off-1] == '/');
        assert(s[off] != '\0' && s[off] != '/');

        /* get next path element length (from off until '/' or nul) */
        for(ne = 1; s[off+ne] != '\0' && s[off+ne] != '/'; ne++)
            ;

        /* truncate at '/' or nul */
        slash = s[off + ne];
        s[off + ne] = '\0';

        /* skip over children (having s as prefix) */
        for(ep = p + 1; slash && ep != epath; ep++){
            if(strncmp(s, *ep, off + ne) != 0)
                break;
            if((*ep)[off+ne] != '\0' && (*ep)[off+ne] != '/')
                break;
        }

        d = dirstat(s);
        e = dirent(&ent, &nent, s + off);
        if(e->islink)
            sysfatal("symlinks may not be modified: %s", s);
        if(e->ismod)
            sysfatal("submodules may not be modified: %s", s);
```

```
    s[off + ne] = slash;

    isdir = d != nil && (d->mode & DMDIR) != 0;
    /*
     * exist? slash? dir?   track?
     * n      _      _      _      -> remove: file gone
     * y      n      n      y      -> blob: tracked non-dir
     * y      n      y      n      -> remove: file untracked
     * y      n      y      n      -> remove: file -> dir
     * y      n      y      y      -> remove: file -> dir
     * y      n      y      n      -> untracked dir, cli junk
     * y      y      y      n      -> recurse
     * y      y      y      y      -> recurse
     */
    if(d == nil || !slash && isdir && tracked(s)){
        /*
         * if a tracked file is removed or turned
         * into a dir, we want to delete it. We
         * only want to change files passed in, and
         * not ones along the way, so ignore files
         * that have a '/'.
         */
        e->name = nil;
        s[off + ne] = slash;
        continue;
    } else if(slash && isdir){
        /*
         * If we have a list of entries that go into
         * a directory, create a tree node for this
         * entry, and recurse down.
         */
        e->mode = DMDIR | 0755;
        o = readobject(e->h);
        if(o == nil || o->type != GTree)
            o = emptydir();
        /*
         * if after processing deletions, a tree is empty,
         * mark it for removal from the parent.
         *
         * Note, it is still written to the object store,
         * but this is fine -- and ensures that an empty
         * repository will continue to work.
         */
        if(treeify(o, p, ep, off + ne + 1, &e->h) == 0)
            e->name = nil;
    }else if(!slash && !isdir){
        /*
         * If the file was explicitly passed in and is
         * not a dir, we want to either remove it or
         * track it, depending on the state of the index.
         */
        if(tracked(s) && !isdir)
            blobify(d, s, &e->mode, &e->h);
        else
            e->name = nil;
    }
    free(d);
}
if(nent == 0)
    sysfatal("%.*s: refusing to update empty directory", off, *path);
```

```
    nent = writetree(ent, nent, h);
    free(ent);
    return nent;
}
```

⟨*function* mkcommit 195a⟩≡ (196)

```
void
mkcommit(Hash *c, vlong date, Hash tree)
{
    char *s, h[64];
    int ns, nh, i;
    Fmt f;

    fmtstrinit(&f);
    fmtprint(&f, "tree %H\n", tree);
    for(i = 0; i < nparents; i++)
        fmtprint(&f, "parent %H\n", parents[i]);
    fmtprint(&f, "author %s <%s> %lld +0000\n", authorname, authoremail, date);
    fmtprint(&f, "committer %s <%s> %lld +0000\n", committername, committeremail, date);
    fmtprint(&f, "\n");
    fmtprint(&f, "%s", commitmsg);
    s = fmtstrflush(&f);

    ns = strlen(s);
    nh = snprint(h, sizeof(h), "%T %d", GCommit, ns) + 1;
    writeobj(c, h, nh, s, ns);
    free(s);
}
```

⟨*function* findroot 195b⟩≡ (196)

```
Object*
findroot(void)
{
    Object *t, *c;
    Hash h;

    if(resolveref(&h, "HEAD") == -1)
        return emptydir();
    if((c = readobject(h)) == nil || c->type != GCommit)
        sysfatal("could not read HEAD %H", h);
    if((t = readobject(c->commit->tree)) == nil)
        sysfatal("could not read tree for commit %H", h);
    return t;
}
```

⟨*function* usage (git9/save.c) 195c⟩≡ (196)

```
void
usage(void)
{
    fprint(2, "usage: %s -n name -e email -m message -d date [files...]\n", argv0);
    exits("usage");
}
```

⟨*function* main (git9/save.c) 195d⟩≡ (196)

```
void
main(int argc, char **argv)
{
    char *ln, *dstr, *parts[4], cwd[1024];
    int i, r, line, ncwd;
    Hash th, ch;
```

```
    vlong date;
    Biobuf *f;
    Object *t;

    gitinit(nil, 0, nil);
    if(access(".git", AEXIST) != 0)
        sysfatal("could not find git repo: %r");
    if(getwd(cwd, sizeof(cwd)) == nil)
        sysfatal("getcwd: %r");
    dstr = nil;
    date = time(nil);
    ncwd = strlen(cwd);

    ARGBEGIN{
    case 'm':
        commitmsg = EARGF(usage());
        break;
    case 'n':
        authorname = EARGF(usage());
        break;
    case 'e':
        authoremail = EARGF(usage());
        break;
    case 'N':
        committername = EARGF(usage());
        break;
    case 'E':
        committeremail = EARGF(usage());
        break;
    case 'd':
        dstr = EARGF(usage());
        break;
    case 'p':
        if(nparents >= Maxparents)
            sysfatal("too many parents");
        if(resolveref(&parents[nparents++], EARGF(usage())) == -1)
            sysfatal("invalid parent: %r");
        break;
    default:
        usage();
        break;
    }ARGEND;
```

⟨git9/save.c 196⟩≡

```
#include <u.h>
#include <libc.h>
#include "git.h"

typedef struct Objbuf Objbuf;
struct Objbuf {
    int off;
    char *hdr;
    int nhdr;
    char *dat;
    int ndat;
};

enum {
    Maxparents = 16,
};
```

```
char    *authorname;
char    *authoremail;
char    *committername;
char    *committeremail;
char    *commitmsg;
Hash    parents[Maxparents];
int nparents;
Idxent  *idx;
int idxsz;
int nidx;
```

⟨*function* `gitmode` 189b⟩

⟨*function* `namecmp` 190a⟩

⟨*function* `idxcmp` 190b⟩

⟨*function* `bwrite` 190c⟩

⟨*function* `objbytes` 190d⟩

⟨*function* `writeobj` 191a⟩

⟨*function* `writetree` 191b⟩

⟨*function* `blobify` 192a⟩

⟨*function* `tracked` 192b⟩

⟨*function* `dirent` 192c⟩

⟨*function* `treeify` 193⟩

⟨*function* `mkcommit` 195a⟩

⟨*function* `findroot` 195b⟩

⟨*function* `usage (git9/save.c)` 195c⟩

⟨*function* `main (git9/save.c)` 195d⟩

```
    if(commitmsg == nil)
        sysfatal("missing message");
    if(authorname == nil)
        sysfatal("missing name");
    if(authoremail == nil)
        sysfatal("missing email");
    if((committername == nil) != (committeremail == nil))
        sysfatal("partially specified committer");
    if(committername == nil && committeremail == nil){
        committername = authorname;
        committeremail = authoremail;
    }
    if(dstr){
        date=strtoll(dstr, &dstr, 10);
        if(strlen(dstr) != 0)
            sysfatal("could not parse date %s", dstr);
    }
```

```
    for(i = 0; i < argc; i++){
        cleanname(argv[i]);
        if(*argv[i] == '/' && strncmp(argv[i], cwd, ncwd) == 0)
            argv[i] += ncwd;
        while(*argv[i] == '/')
            argv[i]++;
    }
    qsort(argv, argc, sizeof(*argv), namecmp);

    t = findroot();
    nidx = 0;
    idxsz = 32;
    idx = emalloc(idxsz*sizeof(Idxent));
    if((f = Bopen(".git/INDEX9", OREAD)) == nil)
        sysfatal("open index: %r");
    line = 0;
    while((ln = Brdstr(f, '\n', 1)) != nil){
        line++;
        if(ln[0] == 0 || ln[0] == '\n')
            continue;
        if(getfields(ln, parts, nelem(parts), 0, " \t") != nelem(parts))
            sysfatal(".git/INDEX9:%d: corrupt index", line);
        if(nidx == idxsz){
            idxsz += idxsz/2;
            idx = realloc(idx, idxsz*sizeof(Idxent));
        }
        cleanname(parts[3]);
        idx[nidx].state = *parts[0];
        idx[nidx].qid = parseqid(parts[1]);
        idx[nidx].mode = strtol(parts[2], nil, 8);
        idx[nidx].path = strdup(parts[3]);
        idx[nidx].order = nidx;
        nidx++;
        free(ln);
    }
    Bterm(f);
    qsort(idx, nidx, sizeof(Idxent), idxcmp);
    r = treeify(t, argv, argv + argc, 0, &th);
    if(r == -1)
        sysfatal("could not commit: %r");
    mkcommit(&ch, date, th);
    print("%H\n", ch);
    exits(nil);
}
```

git9/send.c

⟨*function* `findref` 198a⟩≡ (203a)

```
int
findref(char **r, int nr, char *ref)
{
    int i;
    for(i = 0; i < nr; i++)
        if(strcmp(r[i], ref) == 0)
            return i;
    return -1;
}
```

⟨*function* `findkey` 198b⟩≡ (203a)

```
int
findkey(Map *m, int nm, char *ref)
{
    int i;
    for(i = 0; i < nm; i++)
        if(strcmp(m[i].ref, ref) == 0)
            return i;
    return -1;
}
```

⟨*function* readours 199a⟩≡ (203a)

```
int
readours(Hash **tailp, char ***refp)
{
    int nu, i, idx;
    char *r, *pfx, **ref;
    Hash *tail;

    if(sendall)
        return listrefs(tailp, refp);
    nu = 0;
    tail = eamalloc((nremoved + nbranch), sizeof(Hash));
    ref = eamalloc((nremoved + nbranch), sizeof(char*));
    for(i = 0; i < nbranch; i++){
        ref[nu] = estrdup(branch[i]);
        if(resolveref(&tail[nu], branch[i]) == -1)
            sysfatal("broken branch %s", branch[i]);
        nu++;
    }
    for(i = 0; i < nremoved; i++){
        pfx = "refs/heads/";
        if(strstr(removed[i], "heads/") == removed[i])
            pfx = "refs/";
        if(strstr(removed[i], "refs/heads/") == removed[i])
            pfx = "";
        if((r = smprint("%s%s", pfx, removed[i])) == nil)
            sysfatal("smprint: %r");
        if((idx = findref(ref, nu, r)) == -1)
            idx = nu++;
        else
            free(ref[idx]);
        assert(idx < nremoved + nbranch);
        memcpy(&tail[idx], &Zhash, sizeof(Hash));
        ref[idx] = r;
    }
    dprint(1, "nu: %d\n", nu);
    for(i = 0; i < nu; i++)
        dprint(1, "update: %H %s\n", tail[i], ref[i]);
    *tailp = tail;
    *refp = ref;
    return nu;
}
```

⟨*function* sendpack 199b⟩≡ (203a)

```
int
sendpack(Conn *c)
{
    int i, n, idx, nsp, send, first;
    int nours, ntheirs, nmap;
    char buf[Pktmax], *sp[3];
```

```
Hash h, *theirs, *ours;
Object *a, *b, *p, *o;
char **refs;
Map *map, *m;

first = 1;
nours = readours(&ours, &refs);
theirs = nil;
ntheirs = 0;
nmap = nours;
map = eamalloc(nmap, sizeof(Map));
for(i = 0; i < nmap; i++){
    map[i].theirs = Zhash;
    map[i].ours = ours[i];
    map[i].ref = refs[i];
}
while(1){
    n = readpkt(c, buf, sizeof(buf));
    if(n == -1)
        return -1;
    if(n == 0)
        break;
    if(first && n > strlen(buf))
        parsecaps(buf + strlen(buf) + 1, c);
    first = 0;

    if(getfields(buf, sp, nelem(sp), 1, " \t\r\n") != 2)
        sysfatal("invalid ref line %.*s", utfnlen(buf, n), buf);
    theirs = earealloc(theirs, ntheirs+1, sizeof(Hash));
    if(hparse(&theirs[ntheirs], sp[0]) == -1)
        sysfatal("invalid hash %s", sp[0]);
    if((idx = findkey(map, nmap, sp[1])) != -1)
        map[idx].theirs = theirs[ntheirs];
    /*
     * we only keep their ref if we can read the object to add it
     * to our reachability; otherwise, discard it; we only care
     * that we don't have it, so we can tell whether we need to
     * bail out of pushing.
     */
    if((o = readobject(theirs[ntheirs])) != nil){
        ntheirs++;
        unref(o);
    }
}

if(writephase(c) == -1)
    return -1;
send = 0;
if(force)
    send=1;
for(i = 0; i < nmap; i++){
    m = &map[i];
    a = readobject(m->theirs);
    if(hasheq(&m->ours, &Zhash))
        b = nil;
    else
        b = readobject(m->ours);
    p = nil;
    if(a != nil && b != nil)
        p = ancestor(a, b);
```

```
    if(!force
    && !hasheq(&m->theirs, &Zhash)
    && !hasheq(&m->ours, &Zhash)
    && (a == nil || p != a)){
        fprint(2, "remote has diverged\n");
        werrstr("remote diverged");
        flushpkt(c);
        return -1;
    }
    unref(a);
    unref(b);
    unref(p);
    if(hasheq(&m->theirs, &m->ours)){
        print("uptodate %s\n", m->ref);
        continue;
    }
    print("update %s %H %H\n", m->ref, m->theirs, m->ours);
    n = snprint(buf, sizeof(buf), "%H %H %s", m->theirs, m->ours, m->ref);

    /*
     * Workaround for github.
     *
     * Github will accept the pack but fail to update the references
     * if we don't have capabilities advertised. Report-status seems
     * harmless to add, so we add it.
     *
     * Github doesn't advertise any capabilities, so we can't check
     * for compatibility. We just need to add it blindly.
     */
    if(i == 0 && c->report){
        buf[n++] = '\0';
        n += snprint(buf + n, sizeof(buf) - n, " report-status");
    }
    if(writepkt(c, buf, n) == -1)
        sysfatal("unable to send update pkt");
    send = 1;
}
flushpkt(c);
if(!send){
    fprint(2, "nothing to send\n");
    return 0;
}

if(writepack(c->wfd, ours, nours, theirs, ntheirs, &h) == -1)
    return -1;
if(!c->report)
    return 0;

if(readphase(c) == -1)
    return -1;
/* We asked for a status report, may as well use it. */
while((n = readpkt(c, buf, sizeof(buf))) > 0){
    buf[n] = 0;
    if(chattygit)
        fprint(2, "done sending pack, status %s\n", buf);
    nsp = getfields(buf, sp, nelem(sp), 1, " \t\n\r");
    if(nsp < 2)
        continue;
    if(nsp < 3)
        sp[2] = "";
```

```
        /*
         * Only report errors; successes will be reported by
         * surrounding scripts.
         */
        if(strcmp(sp[0], "unpack") == 0 && strcmp(sp[1], "ok") != 0)
            werrstr("unpack %s", sp[1]);
        else if(strcmp(sp[0], "ng") == 0)
            werrstr("failed update: %s %s", sp[1], sp[2]);
        else
            continue;
        return -1;
    }
    return 0;
}
```

⟨*function* `usage (git9/send.c)` 202a⟩≡ (203a)

```
void
usage(void)
{
    fprint(2, "usage: %s remote [reponame]\n", argv0);
    exits("usage");
}
```

⟨*function* `main (git9/send.c)` 202b⟩≡ (203a)

```
void
main(int argc, char **argv)
{
    char *br;
    Conn c;

    ARGBEGIN{
    default:
        usage();
        break;
    case 'd':
        chattygit++;
        break;
    case 'f':
        force++;
        break;
    case 'r':
        if(nremoved == nelem(removed))
            sysfatal("too many deleted branches");
        removed[nremoved++] = EARGF(usage());
        break;
    case 'a':
        sendall++;
        break;
    case 'b':
        br = EARGF(usage());
        if(strncmp(br, "refs/heads/", strlen("refs/heads/")) == 0)
            br = smprint("%s", br);
        else if(strncmp(br, "heads/", strlen("heads/")) == 0)
            br = smprint("refs/%s", br);
        else
            br = smprint("refs/heads/%s", br);
        branch = erealloc(branch, (nbranch + 1)*sizeof(char*));
        branch[nbranch] = br;
        nbranch++;
        break;
    }ARGEND;
```

⟨git9/send.c 203a⟩≡
```
#include <u.h>
#include <libc.h>

#include "git.h"

typedef struct Map  Map;

struct Map {
    char    *ref;
    Hash    ours;
    Hash    theirs;
};

int sendall;
int force;
int nbranch;
char **branch;
char *removed[128];
int nremoved;
int npacked;
int nsent;

⟨function findref 198a⟩

⟨function findkey 198b⟩

⟨function readours 199a⟩

⟨function sendpack 199b⟩

⟨function usage (git9/send.c) 202a⟩

⟨function main (git9/send.c) 202b⟩

    gitinit(nil, 0, nil);
    if(argc != 1)
        usage();
    if(gitconnect(&c, argv[0], "receive") == -1)
        sysfatal("git connect: %s: %r", argv[0]);
    if(sendpack(&c) == -1)
        sysfatal("send failed: %r");
    closeconn(&c);
    exits(nil);
}
```

git9/serve.c

⟨function fail (git9/serve.c) 203b⟩≡ (213c)
```
_Noreturn static void
fail(Conn *c, char *fmt, ...)
{
    char msg[ERRMAX];
    va_list ap;

    va_start(ap, fmt);
    vsnprint(msg, sizeof(msg), fmt, ap);
    va_end(ap);
    fmtpkt(c, "ERR %s\n", msg);
```

```
    sysfatal("%s", msg);
}
```

⟨*function* gethead 204a⟩≡ (213c)

```
char*
gethead(Hash *h, char *ref, int nref)
{
    int fd, n;
    char *s;

    if((fd = open(".git/HEAD", OREAD)) == -1)
        return nil;
    if((n = readn(fd, ref, nref-1)) == -1)
        return nil;
    ref[n] = 0;
    strip(ref);
    if(strncmp(ref, "ref: ", 5) != 0)
        return nil;
    s = ref+5;
    if(resolveref(h, s) == -1)
        return nil;
    return s;
}
```

⟨*function* showrefs 204b⟩≡ (213c)

```
int
showrefs(Conn *c)
{
    char **names, *s, buf[256];
    int i, ret, nrefs;
    Hash head, *refs;

    ret = -1;
    nrefs = 0;
    refs = nil;
    names = nil;

    if((s = gethead(&head,  buf, sizeof(buf))) == nil)
        memset(&head, 0, sizeof(Hash));
    if(fmtpkt(c, "%H HEAD%csymref=HEAD:%s no-thin\n", head, 0, s) == -1)
        goto error;
    if((nrefs = listrefs(&refs, &names)) == -1)
        fail(c, "listrefs: %r");
    for(i = 0; i < nrefs; i++){
        if(strncmp(names[i], "heads/", strlen("heads/")) != 0)
            continue;
        if(fmtpkt(c, "%H refs/%s\n", refs[i], names[i]) == -1)
            goto error;
    }
    if(flushpkt(c) == -1)
        goto error;
    ret = 0;
error:
    for(i = 0; i < nrefs; i++)
        free(names[i]);
    free(names);
    free(refs);
    return ret;
}
```

⟨*function* servnegotiate 205⟩≡ (213c)

```
int
servnegotiate(Conn *c, Hash **head, int *nhead, Hash **tail, int *ntail)
{
    char pkt[Pktmax];
    int n, acked;
    Object *o;
    Hash h;

    if(showrefs(c) == -1)
        return -1;

    *head = nil;
    *tail = nil;
    *nhead = 0;
    *ntail = 0;
    while(1){
        if((n = readpkt(c, pkt, sizeof(pkt))) == -1)
            goto error;
        if(n == 0)
            break;
        if(strncmp(pkt, "want ", 5) != 0){
            werrstr(" protocol garble %s", pkt);
            goto error;
        }
        if(hparse(&h, &pkt[5]) == -1){
            werrstr(" garbled want");
            goto error;
        }
        if((o = readobject(h)) == nil){
            werrstr("requested nonexistent object");
            goto error;
        }
        unref(o);
        *head = erealloc(*head, (*nhead + 1)*sizeof(Hash));
        (*head)[*nhead] = h;
        *nhead += 1;
    }

    acked = 0;
    while(1){
        if((n = readpkt(c, pkt, sizeof(pkt))) == -1)
            goto error;
        if(strncmp(pkt, "done", 4) == 0)
            break;
        if(n == 0){
            if(!acked && fmtpkt(c, "NAK") == -1)
                    goto error;
        }
        if(strncmp(pkt, "have ", 5) != 0){
            werrstr(" protocol garble %s", pkt);
            goto error;
        }
        if(hparse(&h, &pkt[5]) == -1){
            werrstr(" garbled have");
            goto error;
        }
        if((o = readobject(h)) == nil)
            continue;
        if(!acked){
```

```
            if(fmtpkt(c, "ACK %H", h) == -1)
                goto error;
            acked = 1;
        }
        unref(o);
        *tail = erealloc(*tail, (*ntail + 1)*sizeof(Hash));
        (*tail)[*ntail] = h;
        *ntail += 1;
    }
    if(!acked && fmtpkt(c, "NAK\n") == -1)
        goto error;
    return 0;
error:
    fmtpkt(c, "ERR %r\n");
    free(*head);
    free(*tail);
    return -1;
}
```

⟨*function* servpack 206a⟩≡ (213c)

```
int
servpack(Conn *c)
{
    Hash *head, *tail, h;
    int nhead, ntail;

    dprint(1, "negotiating pack\n");
    if(servnegotiate(c, &head, &nhead, &tail, &ntail) == -1)
        fail(c, "negotiate: %r");
    dprint(1, "writing pack\n");
    if(writepack(c->wfd, head, nhead, tail, ntail, &h) == -1)
        fail(c, "send: %r");
    return 0;
}
```

⟨*function* validref 206b⟩≡ (213c)

```
int
validref(char *s)
{
    cleanname(s);
    if(strncmp(s, "refs/", 5) != 0)
        return 0;
    return okref(s);
}
```

⟨*function* recvnegotiate 206c⟩≡ (213c)

```
int
recvnegotiate(Conn *c, Hash **cur, Hash **upd, char ***ref, int *nupd)
{
    char pkt[Pktmax], refpath[512], *sp[4];
    Hash old, new;
    int n, i;

    if(showrefs(c) == -1)
        return -1;
    *cur = nil;
    *upd = nil;
    *ref = nil;
    *nupd = 0;
    while(1){
```

```
        if((n = readpkt(c, pkt, sizeof(pkt))) == -1)
            goto error;
        if(n == 0)
            break;
        if(getfields(pkt, sp, nelem(sp), 1, " \t\n\r") != 3){
            fmtpkt(c, "ERR  protocol garble %s\n", pkt);
            goto error;
        }
        if(hparse(&old, sp[0]) == -1){
            fmtpkt(c, "ERR bad old hash %s\n", sp[0]);
            goto error;
        }
        if(hparse(&new, sp[1]) == -1){
            fmtpkt(c, "ERR bad new hash %s\n", sp[1]);
            goto error;
        }
        if(!validref(sp[2])){
            fmtpkt(c, "ERR invalid ref %s\n", sp[2]);
            goto error;
        }
        *cur = erealloc(*cur, (*nupd + 1)*sizeof(Hash));
        *upd = erealloc(*upd, (*nupd + 1)*sizeof(Hash));
        *ref = erealloc(*ref, (*nupd + 1)*sizeof(Hash));
        (*cur)[*nupd] = old;
        (*upd)[*nupd] = new;
        (*ref)[*nupd] = estrdup(sp[2]);
        n = snprint(refpath, sizeof(refpath), ".git/%s", sp[2]);
        if(n >= sizeof(refpath)-1){
            fmtpkt(c, "ERR invalid ref %s\n", sp[2]);
            goto error;
        }
        if(access(refpath, AWRITE) == -1
        && access(refpath, AEXIST) == 0){
            fmtpkt(c, "ERR read-only ref %s\n", sp[2]);
            goto error;
        }
        *nupd += 1;
    }
    return 0;
error:
    free(*cur);
    free(*upd);
    for(i = 0; i < *nupd; i++)
        free((*ref)[i]);
    free(*ref);
    return -1;
}
```

⟨*function* rename (git9/serve.c) 207⟩≡ (213c)

```
int
rename(char *pack, char *idx, Hash h)
{
    char name[128], path[196];
    Dir st;

    nulldir(&st);
    st.name = name;
    snprint(name, sizeof(name), "%H.pack", h);
    snprint(path, sizeof(path), ".git/objects/pack/%s", name);
    if(access(path, AEXIST) == 0)
```

```
        fprint(2, "warning, pack %s already pushed\n", name);
    else if(dirwstat(pack, &st) == -1)
        return -1;
    snprint(name, sizeof(name), "%H.idx", h);
    snprint(path, sizeof(path), ".git/objects/pack/%s", name);
    if(access(path, AEXIST) == 0)
        fprint(2, "warning, pack %s already indexed\n", name);
    else if(dirwstat(idx, &st) == -1)
        return -1;
    return 0;
}
```

⟨*function* checkhash (git9/serve.c) 208a⟩≡ (213c)

```
int
checkhash(int fd, vlong sz, Hash *hcomp)
{
    DigestState *st;
    Hash hexpect;
    char buf[Pktmax];
    vlong n, r;
    int nr;

    if(sz < 28){
        werrstr("undersize packfile");
        return -1;
    }

    st = nil;
    n = 0;
    if(seek(fd, 0, 0) == -1)
        sysfatal("packfile seek: %r");
    while(n != sz - 20){
        nr = sizeof(buf);
        if(sz - n - 20 < sizeof(buf))
            nr = sz - n - 20;
        r = readn(fd, buf, nr);
        if(r != nr){
            werrstr("short read");
            return -1;
        }
        st = sha1((uchar*)buf, nr, nil, st);
        n += r;
    }
    sha1(nil, 0, hcomp->h, st);
    if(readn(fd, hexpect.h, sizeof(hexpect.h)) != sizeof(hexpect.h))
        sysfatal("truncated packfile");
    if(!hasheq(hcomp, &hexpect)){
        werrstr("bad hash: %H != %H", *hcomp, hexpect);
        return -1;
    }
    return 0;
}
```

⟨*function* mkdir 208b⟩≡ (213c)

```
int
mkdir(char *dir)
{
    char buf[ERRMAX];
    int f;
```

```
    if(access(dir, AEXIST) == 0)
        return 0;
    if((f = create(dir, OREAD, DMDIR | 0755)) == -1){
        rerrstr(buf, sizeof(buf));
        if(strstr(buf, "exist") == nil)
            return -1;
    }
    close(f);
    return 0;
}
```

⟨*function* mkpath 209a⟩≡ (213c)

```
int
mkpath(char *path)
{
    char *p;

    p = path;
    /*
     * we assume that the path has been
     * checked with validref(), so there
     * are no double '/' or odd characters.
     */
    while(1){
        p = strchr(p, '/');
        if(p == nil)
            return 0;
        *p = 0;
        if(mkdir(path) == -1)
            return -1;
        *p++ = '/';
    }
}
```

⟨*function* updatepack 209b⟩≡ (213c)

```
int
updatepack(Conn *c)
{
    char buf[Pktmax], packtmp[128], idxtmp[128], ebuf[ERRMAX];
    int n, pfd, packsz;
    Hash h;

    /* make sure the needed dirs exist */
    if(mkdir(".git/objects") == -1)
        return -1;
    if(mkdir(".git/objects/pack") == -1)
        return -1;
    if(mkdir(".git/refs") == -1)
        return -1;
    if(mkdir(".git/refs/heads") == -1)
        return -1;
    snprint(packtmp, sizeof(packtmp), ".git/objects/pack/recv-%d.pack.tmp", getpid());
    snprint(idxtmp, sizeof(idxtmp), ".git/objects/pack/recv-%d.idx.tmp", getpid());
    if((pfd = create(packtmp, ORDWR, 0644)) == -1)
        return -1;
    packsz = 0;
    while(1){
        n = read(c->rfd, buf, sizeof(buf));
        if(n == 0)
            break;
```

```
            if(n == -1){
                rerrstr(ebuf, sizeof(ebuf));
                if(strstr(ebuf, "hungup") == nil)
                    return -1;
                break;
            }
            if(write(pfd, buf, n) != n)
                return -1;
            packsz += n;
        }
        if(checkhash(pfd, packsz, &h) == -1){
            dprint(1, "hash mismatch\n");
            goto error_1;
        }
        if(indexpack(packtmp, idxtmp, h) == -1){
            dprint(1, "indexing failed: %r\n");
            goto error_1;
        }
        if(rename(packtmp, idxtmp, h) == -1){
            dprint(1, "rename failed: %r\n");
            goto error_2;
        }
        return 0;

    error_2:    remove(idxtmp);
    error_1:    remove(packtmp);
        return -1;
    }
```

⟨*function* `lockrepo` 210a⟩≡ (213c)

```
    int
    lockrepo(void)
    {
        int fd, i;

        for(i = 0; i < 10; i++) {
            if((fd = create(".git/_lock", ORCLOSE|ORDWR|OTRUNC|OEXCL, 0644))!= -1)
                return fd;
            sleep(250);
        }
        return -1;
    }
```

⟨*function* `updaterefs` 210b⟩≡ (213c)

```
    int
    updaterefs(Conn *c, Hash *cur, Hash *upd, char **ref, int nupd)
    {
        char refpath[512], buf[128];
        int i, newidx, hadref, fd, ret, lockfd;
        vlong newtm;
        Object *o;
        Hash h;

        ret = -1;
        hadref = 0;
        newidx = -1;
        /*
         * Date of Magna Carta.
         * Wrong because it  was computed using
         * the proleptic gregorian calendar.
```

```
 */
newtm = -23811206400;
if((lockfd = lockrepo()) == -1){
    snprint(buf, sizeof(buf), "repo locked\n");
    return -1;
}
for(i = 0; i < nupd; i++){
    if(resolveref(&h, ref[i]) == 0){
        hadref = 1;
        if(!hasheq(&h, &cur[i])){
            snprint(buf, sizeof(buf), "old ref changed: %s", ref[i]);
            goto error;
        }
    }
    if(snprint(refpath, sizeof(refpath), ".git/%s", ref[i]) == sizeof(refpath)){
        snprint(buf, sizeof(buf), "ref path too long: %s", ref[i]);
        goto error;
    }
    if(hasheq(&upd[i], &Zhash)){
        remove(refpath);
        continue;
    }
    if((o = readobject(upd[i])) == nil){
        snprint(buf, sizeof(buf), "update to nonexistent hash %H", upd[i]);
        goto error;
    }
    if(o->type != GCommit){
        snprint(buf, sizeof(buf), "not commit: %H", upd[i]);
        goto error;
    }
    if(o->commit->mtime > newtm){
        newtm = o->commit->mtime;
        newidx = i;
    }
    unref(o);
    if(mkpath(refpath) == -1){
        snprint(buf, sizeof(buf), "create path: %r");
        goto error;
    }
    if((fd = create(refpath, OWRITE|OTRUNC, 0644)) == -1){
        snprint(buf, sizeof(buf), "open ref: %r");
        goto error;
    }
    if(fprint(fd, "%H", upd[i]) == -1){
        snprint(buf, sizeof(buf), "upate ref: %r");
        close(fd);
        goto error;
    }
    close(fd);
}
/*
 * Heuristic:
 * If there are no valid refs, and HEAD is invalid, then
 * pick the ref with the newest commits as the default
 * branch.
 *
 * Several people have been caught out by pushing to
 * a repo where HEAD named differently from what got
 * pushed, and this is going to be more of a footgun
 * when 'master', 'main', and 'front' are all in active
```

```
     * use. This should make us pick a useful default in
     * those cases, instead of silently failing.
     */
    if(resolveref(&h, "HEAD") == -1 && hadref == 0 && newidx != -1){
        if((fd = create(".git/HEAD", OWRITE|OTRUNC, 0644)) == -1){
            snprint(buf, sizeof(buf), "open HEAD: %r");
            goto error;
        }
        if(fprint(fd, "ref: %s", ref[newidx]) == -1){
            snprint(buf, sizeof(buf), "write HEAD ref: %r");
            goto error;
        }
        close(fd);
    }
    ret = 0;
error:
    if(ret != 0)
        fmtpkt(c, "ERR %s", buf);
    close(lockfd);
    werrstr(buf);
    return ret;
}
```

⟨*function* recvpack 212a⟩≡ (213c)

```
int
recvpack(Conn *c)
{
    Hash *cur, *upd;
    char **ref;
    int nupd;

    if(!allowwrite)
        fail(c, "read-only repo");
    if(recvnegotiate(c, &cur, &upd, &ref, &nupd) == -1)
        fail(c, "negotiate refs: %r");
    if(nupd != 0 && updatepack(c) == -1)
        sysfatal("update pack: %r");
    if(nupd != 0 && updaterefs(c, cur, upd, ref, nupd) == -1)
        sysfatal("update refs: %r");
    return 0;
}
```

⟨*function* parsecmd 212b⟩≡ (213c)

```
char*
parsecmd(char *buf, char *cmd, int ncmd)
{
    int i;
    char *p;

    for(p = buf, i = 0; *p && i < ncmd - 1; i++, p++){
        if(*p == ' ' || *p == '\t'){
            cmd[i] = 0;
            break;
        }
        cmd[i] = *p;
    }
    while(*p == ' ' || *p == '\t')
        p++;
    return p;
}
```

⟨*function* usage (git9/serve.c) 213a⟩≡ (213c)

```
void
usage(void)
{
    fprint(2, "usage: %s [-dw] [-r rel]\n", argv0);
    exits("usage");
}
```

⟨*function* main (git9/serve.c) 213b⟩≡ (213c)

```
void
main(int argc, char **argv)
{
    char *repo, cmd[32], buf[512];
    Conn c;

    ARGBEGIN{
    case 'd':
        chattygit++;
        break;
    case 'r':
        pathpfx = EARGF(usage());
        if(*pathpfx != '/')
            sysfatal("path prefix must begin with '/'");
        break;
    case 'w':
        allowwrite++;
        break;
    default:
        usage();
        break;
    }ARGEND;
```

⟨git9/serve.c 213c⟩≡

```
#include <u.h>
#include <libc.h>
#include <ctype.h>
#include <auth.h>

#include "git.h"

char    *pathpfx = nil;
int allowwrite;
```

⟨*function* fail (git9/serve.c) 203b⟩

⟨*function* gethead 204a⟩

⟨*function* showrefs 204b⟩

⟨*function* servnegotiate 205⟩

⟨*function* servpack 206a⟩

⟨*function* validref 206b⟩

⟨*function* recvnegotiate 206c⟩

⟨*function* rename (git9/serve.c) 207⟩

⟨*function* checkhash (git9/serve.c) 208a⟩

⟨*function* mkdir 208b⟩

⟨*function* mkpath 209a⟩

⟨*function* updatepack 209b⟩

⟨*function* lockrepo 210a⟩

⟨*function* updaterefs 210b⟩

⟨*function* recvpack 212a⟩

⟨*function* parsecmd 212b⟩

⟨*function* usage (git9/serve.c) 213a⟩

⟨*function* main (git9/serve.c) 213b⟩

```
    interactive = 0;
    if(rfork(RFNAMEG) == -1)
        sysfatal("rfork: %r");
    if(pathpfx != nil){
        if(bind(pathpfx, "/", MREPL) == -1)
            sysfatal("bind: %r");
    }
    if(rfork(RFNOMNT) == -1)
        sysfatal("rfork: %r");

    initconn(&c, 0, 1);
    if(readpkt(&c, buf, sizeof(buf)) == -1)
        sysfatal("readpkt: %r");
    repo = parsecmd(buf, cmd, sizeof(cmd));
    cleanname(repo);
    if(strncmp(repo, "../", 3) == 0)
        fail(&c, "invalid path %s\n", repo);
    if(bind(repo, "/", MREPL) == -1)
        fail(&c, "no such repo: %s", repo);
    if(chdir("/") == -1)
        fail(&c, "no such repo");
    gitinit(nil, 0, nil);
    if(strcmp(cmd, "git-receive-pack") == 0)
        recvpack(&c);
    else if(strcmp(cmd, "git-upload-pack") == 0)
        servpack(&c);
    else
        fail(&c, "unsupported command '%s'", cmd);
    exits(nil);
}
```

git9/util.c

⟨*function* emptydir 214⟩≡ (224)

```
Object*
emptydir(void)
{
    static Object *e;

    if(e != nil)
```

```
        return ref(e);
    e = emalloc(sizeof(Object));
    e->hash = Zhash;
    e->type = GTree;
    e->tree = emalloc(sizeof(Tinfo));
    e->tree->ent = nil;
    e->tree->nent = 0;
    e->flag |= Cloaded|Cparsed;
    e->off = -1;
    ref(e);
    cache(e);
    return e;
}
```

⟨*function* entcmp 215⟩≡ (224)

```
int
entcmp(void *pa, void *pb)
{
    Dirent *ae, *be;
    uchar *a, *b;
    int ca, cb;

    ae = pa;
    be = pb;
    a = (uchar*)ae->name;
    b = (uchar*)be->name;
    /*
     * If the files have the same name, they're equal.
     * Otherwise, If they're trees, they sort as thoug
     * there was a trailing slash.
     *
     * Wat.
     */
    while(1){
        ca = *a++;
        cb = *b++;
        /*
         * because these are dir entries in a tree,
         * the only '/' allowable is the virtual '/'
         * at the end of the file name.
         */
        assert(ca != '/' && cb != '/');
        if(ca != cb){
            if(ca == 0 && (ae->mode & DMDIR))
                ca = '/';
            if(cb == 0 && (be->mode & DMDIR))
                cb = '/';
            return (ca > cb) ? 1 : -1;
        }
        if(ca == 0){
            if(ae->mode & DMDIR)
                ca = '/';
            if(be->mode & DMDIR)
                cb = '/';
            if(ca == cb)
                return 0;
            return (ca > cb) ? 1 : -1;
        }
    }
}
```

⟨*function* hasheq 216a⟩≡ (224)
```
int
hasheq(Hash *a, Hash *b)
{
    return memcmp(a->h, b->h, sizeof(a->h)) == 0;
}
```

⟨*function* charval 216b⟩≡ (224)
```
int
charval(int c)
{
    if(c >= '0' && c <= '9')
        return c - '0';
    if(c >= 'a' && c <= 'f')
        return c - 'a' + 10;
    if(c >= 'A' && c <= 'F')
        return c - 'A' + 10;
    werrstr("invalid hex char");
    return -1;
}
```

⟨*function* emalloc 216c⟩≡ (224)
```
void *
emalloc(ulong n)
{
    void *v;

    v = mallocz(n, 1);
    if(v == nil)
        sysfatal("malloc: %r");
    setmalloctag(v, getcallerpc(&n));
    return v;
}
```

⟨*function* eamalloc 216d⟩≡ (224)
```
void *
eamalloc(ulong n, ulong sz)
{
    uvlong na;
    void *v;

    na = (uvlong)n*(uvlong)sz;
    if(na >= (1ULL<<30))
        sysfatal("alloc: overflow");
    v = mallocz(na, 1);
    if(v == nil)
        sysfatal("malloc: %r");
    setmalloctag(v, getcallerpc(&n));
    return v;
}
```

⟨*function* erealloc 216e⟩≡ (224)
```
void *
erealloc(void *p, ulong n)
{
    void *v;

    v = realloc(p, n);
    if(v == nil)
        sysfatal("realloc: %r");
```

```
    setmalloctag(v, getcallerpc(&p));
    return v;
}
```

⟨*function* `earealloc` 217a⟩≡ (224)

```
void *
earealloc(void *p, ulong n, ulong sz)
{
    uvlong na;
    void *v;

    na = (uvlong)n*(uvlong)sz;
    if(na >= (1ULL<<30))
        sysfatal("alloc: overflow");
    v = realloc(p, na);
    if(v == nil)
        sysfatal("realloc: %r");
    setmalloctag(v, getcallerpc(&p));
    return v;
}
```

⟨*function* `estrdup` 217b⟩≡ (224)

```
char*
estrdup(char *s)
{
    s = strdup(s);
    if(s == nil)
        sysfatal("strdup: %r");
    setmalloctag(s, getcallerpc(&s));
    return s;
}
```

⟨*function* `Hfmt` 217c⟩≡ (224)

```
int
Hfmt(Fmt *fmt)
{
    Hash h;
    int i, n, l;
    char c0, c1;

    l = 0;
    h = va_arg(fmt->args, Hash);
    for(i = 0; i < sizeof h.h; i++){
        n = (h.h[i] >> 4) & 0xf;
        c0 = (n >= 10) ? n-10 + 'a' : n + '0';
        n = h.h[i] & 0xf;
        c1 = (n >= 10) ? n-10 + 'a' : n + '0';
        l += fmtprint(fmt, "%c%c", c0, c1);
    }
    return l;
}
```

⟨*function* `Tfmt` 217d⟩≡ (224)

```
int
Tfmt(Fmt *fmt)
{
    int t;
    int l;

    t = va_arg(fmt->args, int);
```

```
    switch(t){
    case GNone: l = fmtprint(fmt, "none");  break;
    case GCommit:   l = fmtprint(fmt, "commit");    break;
    case GTree: l = fmtprint(fmt, "tree");  break;
    case GBlob: l = fmtprint(fmt, "blob");  break;
    case GTag:  l = fmtprint(fmt, "tag");   break;
    case GOdelta:   l = fmtprint(fmt, "odelta");    break;
    case GRdelta:   l = fmtprint(fmt, "gdelta");    break;
    default:    l = fmtprint(fmt, "?%d?", t);   break;
    }
    return l;
}
```

⟨*function* `Ofmt` 218a⟩≡ (224)

```
int
Ofmt(Fmt *fmt)
{
    Object *o;
    int l;

    o = va_arg(fmt->args, Object *);
    print("== %H (%T) ==\n", o->hash, o->type);
    switch(o->type){
    case GTree:
        l = fmtprint(fmt, "tree\n");
        break;
    case GBlob:
        l = fmtprint(fmt, "blob %s\n", o->data);
        break;
    case GCommit:
        l = fmtprint(fmt, "commit\n");
        break;
    case GTag:
        l = fmtprint(fmt, "tag\n");
        break;
    default:
        l = fmtprint(fmt, "invalid: %d\n", o->type);
        break;
    }
    return l;
}
```

⟨*function* `Qfmt` 218b⟩≡ (224)

```
int
Qfmt(Fmt *fmt)
{
    Qid q;

    q = va_arg(fmt->args, Qid);
    if(q.path == ~0ULL && q.vers == ~0UL && q.type == 0xff)
        return fmtprint(fmt, "NOQID");
    else
        return fmtprint(fmt, "%llux.%lud.%hhx", q.path, q.vers, q.type);
}
```

⟨*function* `findrepo` 218c⟩≡ (224)

```
/* Finds the directory containing the git repo. */
static void
findrepo(char *buf, int nbuf, int *nrel)
{
```

```
    char *p, *suff;

    suff = "/.git/HEAD";
    if(getwd(buf, nbuf - strlen(suff) - 1) == nil)
        sysfatal("getwd: %r");

    *nrel = 0;
    for(p = buf + strlen(buf); p != nil; p = strrchr(buf, '/')){
        strcpy(p, suff);
        if(access(buf, AEXIST) == 0){
            p[p == buf] = '\0';
            return;
        }
        *nrel += 1;
        *p = '\0';
    }
    sysfatal("not a git repository");
}
```

⟨*function* gitinit 219a⟩≡ (224)

```
void
gitinit(char *root, int nroot, int *nrel)
{
    char repo[512] = ".git";
    Dir *d;

    fmtinstall('H', Hfmt);
    fmtinstall('T', Tfmt);
    fmtinstall('O', Ofmt);
    fmtinstall('Q', Qfmt);
    inflateinit();
    deflateinit();
    authorpat = regcomp("[\t ]*(.*)[\t ]+([0-9]+)[\t ]*([\\-+]?[0-9]+)?");
    osinit(&objcache);
    if(root != nil){
        findrepo(root, nroot, nrel);
        snprint(repo, sizeof(repo), "%s/.git", root);
    }
    if((d = dirstat(repo)) == nil)
        sysfatal("stat %s: %r", repo);
    gitdirmode = d->mode & 0777;
    free(d);
}
```

⟨*function* hparse 219b⟩≡ (224)

```
int
hparse(Hash *h, char *b)
{
    int i, c0, c1;

    for(i = 0; i < nelem(h->h); i++){
        if((c0 = charval(b[2*i+0])) == -1)
            return -1;
        if((c1 = charval(b[2*i+1])) == -1)
            return -1;
        h->h[i] = (c0 << 4) | c1;
    }
    return 0;
}
```

⟨*function* `slurpdir` 220a⟩≡ (224)

```
int
slurpdir(char *p, Dir **d)
{
    int r, f;

    if((f = open(p, OREAD)) == -1)
        return -1;
    r = dirreadall(f, d);
    close(f);
    return r;
}
```

⟨*function* `hassuffix` 220b⟩≡ (224)

```
int
hassuffix(char *base, char *suf)
{
    int nb, ns;

    nb = strlen(base);
    ns = strlen(suf);
    if(ns <= nb && strcmp(base + (nb - ns), suf) == 0)
        return 1;
    return 0;
}
```

⟨*function* `swapsuffix` 220c⟩≡ (224)

```
int
swapsuffix(char *dst, int dstsz, char *base, char *oldsuf, char *suf)
{
    int bl, ol, sl, l;

    bl = strlen(base);
    ol = strlen(oldsuf);
    sl = strlen(suf);
    l = bl + sl - ol;
    if(l + 1 > dstsz || ol > bl)
        return -1;
    memmove(dst, base, bl - ol);
    memmove(dst + bl - ol, suf, sl);
    dst[l] = 0;
    return l;
}
```

⟨*function* `strip` 220d⟩≡ (224)

```
char *
strip(char *s)
{
    char *e;

    while(isspace(*s))
        s++;
    e = s + strlen(s);
    while(e > s && isspace(*--e))
        *e = 0;
    return s;
}
```

⟨*function* _dprint 221a⟩≡ (224)

```
void
_dprint(char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    vfprint(2, fmt, ap);
    va_end(ap);
}
```

⟨*function* showprogress 221b⟩≡ (224)

```
int
showprogress(int x, int pct)
{
    if(!interactive)
        return 0;
    if(x > pct){
        pct = x;
        fprint(2, "\b\b\b\b%3d%%", pct);
    }
    return pct;
}
```

⟨*function* qinit 221c⟩≡ (224)

```
void
qinit(Objq *q)
{
    memset(q, 0, sizeof(Objq));
    q->nheap = 0;
    q->heapsz = 8;
    q->heap = eamalloc(q->heapsz, sizeof(Qelt));
}
```

⟨*function* qclear 221d⟩≡ (224)

```
void
qclear(Objq *q)
{
    free(q->heap);
}
```

⟨*function* qput 221e⟩≡ (224)

```
void
qput(Objq *q, Object *o, int color)
{
    Qelt t;
    int i;

    assert(o->type == GCommit);
    if(q->nheap == q->heapsz){
        q->heapsz *= 2;
        q->heap = earealloc(q->heap, q->heapsz, sizeof(Qelt));
    }
    q->heap[q->nheap].o = o;
    q->heap[q->nheap].color = color;
    q->heap[q->nheap].ctime = o->commit->ctime;
    for(i = q->nheap; i > 0; i = (i-1)/2){
        if(q->heap[i].ctime < q->heap[(i-1)/2].ctime)
            break;
        t = q->heap[i];
```

```
        q->heap[i] = q->heap[(i-1)/2];
        q->heap[(i-1)/2] = t;
    }
    q->nheap++;
}
```

⟨*function* qpop 222a⟩≡ (224)

```
int
qpop(Objq *q, Qelt *e)
{
    int i, l, r, m;
    Qelt t;

    if(q->nheap == 0)
        return 0;
    *e = q->heap[0];
    if(--q->nheap == 0)
        return 1;

    i = 0;
    q->heap[0] = q->heap[q->nheap];
    while(1){
        m = i;
        l = 2*i+1;
        r = 2*i+2;
        if(l < q->nheap && q->heap[m].ctime < q->heap[l].ctime)
            m = l;
        if(r < q->nheap && q->heap[m].ctime < q->heap[r].ctime)
            m = r;
        if(m == i)
            break;
        t = q->heap[m];
        q->heap[m] = q->heap[i];
        q->heap[i] = t;
        i = m;
    }
    return 1;
}
```

⟨*function* murmurhash2 222b⟩≡ (224)

```
u64int
murmurhash2(void *pp, usize n)
{
    u32int m = 0x5bd1e995;
    u32int r = 24;
    u32int h, k;
    uchar *w, *e;

    h = Seed ^ n;
    e = pp;
    e += n & -4;
    for (w = pp; w != e; w += 4) {
        k = (u32int)w[0] | (u32int)w[1] << 8 | (u32int)w[2] << 16 | (u32int)w[3] << 24;
        k *= m;
        k ^= k >> r;
        k *= m;

        h *= m;
        h ^= k;
    }
```

```
    switch (n & 0x3) {
    case 3: h ^= w[2] << 16;
    case 2: h ^= w[1] << 8;
    case 1: h ^= w[0] << 0;
        h *= m;
    }

    h ^= h >> 13;
    h *= m;
    h ^= h >> 15;

    return h;
}
```

⟨*function* parseqid 223a⟩≡ (224)

```
Qid
parseqid(char *s)
{
    char *e;
    Qid q;

    if(strcmp(s, "NOQID") == 0)
        return (Qid){-1, -1, -1};
    e = s;
    q.path = strtoull(e, &e, 16);
    if(*e != '.')
        sysfatal("corrupt qid: %s (%s)", s, e);
    q.vers = strtoul(e+1, &e, 10);
    if(*e != '.')
        sysfatal("corrupt qid: %s (%s)", s, e);
    q.type = strtoul(e+1, &e, 16);
    if(*e != '\0')
        sysfatal("corrupt qid: %s (%x)", s, *e);
    return q;
}
```

⟨*function* okref 223b⟩≡ (224)

```
/*
 * Checks the rules for valid ref names, as defined in
 *   git/Documentation/protocol-common.txt.
 * It does not check that the ref begins with refs/
 * or is called HEAD, since that's only needed in the
 * clone protocol.
 */
int
okref(char *ref)
{
    int n, slashed;
    char *p;
    Rune r;

    slashed = 0;
    if(*ref == '/' || *ref == '.')
        return 0;
    for(p = ref; *p != 0; p += n) {
        n = chartorune(&r, p);
        switch(r){
        case '.':
            if(p[1]== 0 || p[1] == '.')
```

```
                return 0;
            if(strcmp(p, ".lock") == 0)
                return 0;
            break;
        case '/':
            if(p[1] == 0 || p[1] == '.' || p[1] == '/')
                return 0;
            slashed = 1;
            break;
        case '@':
            if(p[1] == '{')
                return 0;
            break;
        case ' ':
        case '~':
        case '^':
        case ':':
        case '?':
        case '*':
        case '[':
        case '\\':
        case Runeerror:
        case 0x7f: /* DEL */
            return 0;
        default:
            if(r < 0x20 || isspacerune(r))
                return 0;
        }
    }
    if(ref == p)
        return 0;
    return slashed;
}
```

⟨git9/util.c 224⟩≡

```
#include <u.h>
#include <libc.h>
#include <ctype.h>

#include "git.h"

Reprog  *authorpat;
Hash    Zhash;
int chattygit;
int interactive = 1;
int gitdirmode = -1;

enum {
    Seed        = 2928213749ULL
};
```

⟨*function* emptydir 214⟩

⟨*function* entcmp 215⟩

⟨*function* hasheq 216a⟩

⟨*function* charval 216b⟩

⟨*function* emalloc 216c⟩

⟨*function* `eamalloc` 216d⟩

⟨*function* `erealloc` 216e⟩

⟨*function* `earealloc` 217a⟩

⟨*function* `estrdup` 217b⟩

⟨*function* `Hfmt` 217c⟩

⟨*function* `Tfmt` 217d⟩

⟨*function* `Ofmt` 218a⟩

⟨*function* `Qfmt` 218b⟩

⟨*function* `findrepo` 218c⟩

⟨*function* `gitinit` 219a⟩

⟨*function* `hparse` 219b⟩

⟨*function* `slurpdir` 220a⟩

⟨*function* `hassuffix` 220b⟩

⟨*function* `swapsuffix` 220c⟩

⟨*function* `strip` 220d⟩

⟨*function* `_dprint` 221a⟩

⟨*function* `showprogress` 221b⟩

⟨*function* `qinit` 221c⟩

⟨*function* `qclear` 221d⟩

⟨*function* `qput` 221e⟩

⟨*function* `qpop` 222a⟩

⟨*function* `murmurhash2` 222b⟩

⟨*function* `parseqid` 223a⟩

⟨*function* `okref` 223b⟩

git9/walk.c

⟨*constant* `NCACHE` 225a⟩≡ (232)

```
#define NCACHE 4096
```

⟨*function* `checkedin` 225b⟩≡ (232)

```
int
checkedin(Idxent *e, int change)
{
    char *p;
    int r;
```

```
    /* clean index, no need to check tree */
    if(cleanidx)
        return e >= &idx[0] && e < &idx[nidx] && e->state != 'R';

    if((p = smprint("%s/%s", bdir, e->path)) == nil)
        sysfatal("smprint: %r");

    r = access(p, AEXIST);
    if(r == 0 && change){
        if(e->state != 'R')
            e->state = 'T';
        staleidx = 1;
    }
    free(p);

    return r == 0;
}
```

⟨*function* pathcmp 226a⟩≡ (232)

```
/*
 * Tricky; we want to know if a file or dir is indexed,
 * but a dir is only indexed if we have a file with dir/
 * listed in the index.
 *
 * as a result, we need to add a virtual '/' at the end
 * of the path if we're doing it, so if we have:
 *  foo.bar/x
 *  foo/y
 * and we want to find out if foo is a directory we should
 * descend into, we need to compare as though foo/ ended
 * with a '/', or we'll bsearch down do foo.bar, not foo.
 *
 * this code resembles entcmp() in util.c, but differs
 * because we're comparing whole paths.
 */
int
pathcmp(char *sa, char *sb, int sadir)
{
    unsigned a, b;

    while(1){
        a = *sa++;
        b = *sb++;
        if(a != b){
            if(a == 0 && sadir)
                a = '/';
            if(a == '/' && b == '/')
                return 0;
            return (a > b) ? 1 : -1;
        }
        if(a == 0)
            return 0;
    }
}
```

⟨*function* indexed 226b⟩≡ (232)

```
int
indexed(char *path, int dir)
{
```

```
    int lo, hi, mid, r;

    r = -1;
    lo = 0;
    hi = nidx-1;
    while(lo <= hi){
        mid = (hi + lo) / 2;
        r = pathcmp(path, idx[mid].path, dir);
        if(r < 0)
            hi = mid-1;
        else if(r > 0)
            lo = mid+1;
        else
            break;
    }
    return r == 0;
}
```

⟨*function* idxcmp (git9/walk.c) 227a⟩≡ (232)

```
int
idxcmp(void *pa, void *pb)
{
    Idxent *a, *b;
    int c;

    a = (Idxent*)pa;
    b = (Idxent*)pb;
    if((c = strcmp(a->path, b->path)) != 0)
        return c;
    /* maintain load order if name is identical */
    return a->order < b->order ? -1 : 1;
}
```

⟨*function* samedata 227b⟩≡ (232)

```
/*
 * compares whether the indexed entry 'a'
 * has the same contents and mode as
 * the entry on disk 'b'; if the indexed
 * entry is nil, does a deep comparison
 * of the checked out file and the file
 * checked in.
 */
int
samedata(Idxent *a, Idxent *b)
{
    char *gitpath, ba[IOUNIT], bb[IOUNIT];
    int fa, fb, na, nb, same;
    Dir *da, *db;

    if(a != nil){
        if(a->qid.path == b->qid.path
        && a->qid.vers == b->qid.vers
        && a->qid.type == b->qid.type
        && a->mode == b->mode
        && a->mode != 0)
            return 1;
    }

    same = 0;
    da = nil;
```

```
    db = nil;
    if((gitpath = smprint("%s/%s", bdir, b->path)) == nil)
        sysfatal("smprint: %r");
    fa = open(gitpath, OREAD);
    fb = open(b->path, OREAD);
    if(fa == -1 || fb == -1)
        goto mismatch;
    da = dirfstat(fa);
    db = dirfstat(fb);
    if(da == nil || db == nil)
        goto mismatch;
    if((da->mode&0100) != (db->mode&0100))
        goto mismatch;
    if(da->length != db->length)
        goto mismatch;
    while(1){
        if((na = readn(fa, ba, sizeof(ba))) == -1)
            goto mismatch;
        if((nb = readn(fb, bb, sizeof(bb))) == -1)
            goto mismatch;
        if(na != nb)
            goto mismatch;
        if(na == 0)
            break;
        if(memcmp(ba, bb, na) != 0)
            goto mismatch;
    }
    if(a != nil){
        a->qid = db->qid;
        a->mode = db->mode;
        staleidx = 1;
    }
    same = 1;

mismatch:
    free(da);
    free(db);
    if(fa != -1)
        close(fa);
    if(fb != -1)
        close(fb);
    return same;
}
```

⟨*function* loadent 228⟩≡ (232)

```
void
loadent(char *dir, Dir *d, int fullpath)
{
    char *path;
    Idxent *e;

    if(fullpath)
        path = estrdup(dir);
    else if((path = smprint("%s/%s", dir, d->name)) == nil)
        sysfatal("smprint: %r");

    cleanname(path);
    if(!intree && (printflg & Uflg) == 0 && !indexed(path, d->qid.type & QTDIR)){
        free(path);
        return;
```

```
    }
    if(d->qid.type & QTDIR){
        loadwdir(path);
        free(path);
    }else{
        if(nwdir == wdirsz){
            wdirsz += wdirsz/2;
            wdir = erealloc(wdir, wdirsz*sizeof(Idxent));
        }
        e = wdir + nwdir;
        e->path = path;
        e->qid = !intree? d->qid : (Qid){-1,-1,-1};
        e->mode = d->mode;
        e->order = nwdir;
        e->state = 'T';
        nwdir++;
    }
}
```

⟨*function* `loadwdir` 229a⟩≡ (232)

```
void
loadwdir(char *path)
{
    int fd, i, n;
    Dir *d, *e;

    d = nil;
    e = nil;
    cleanname(path);
    if(!intree
    && strncmp(path, ".git", 4) == 0
    && (path[4] == '/' || path[4] == 0))
        return;

    if((fd = open(path, OREAD)) < 0)
        goto error;
    if((e = dirfstat(fd)) == nil)
        fprint(2, "fstat: %r");
    if(e->qid.type & QTDIR)
        while((n = dirread(fd, &d)) > 0){
            for(i = 0; i < n; i++)
                loadent(path, &d[i], 0);
            free(d);
        }
    else
        loadent(path, e, 1);
error:
    free(e);
    if(fd != -1)
        close(fd);
}
```

⟨*function* `pfxmatch` 229b⟩≡ (232)

```
int
pfxmatch(char *p, char **pfx, int *pfxlen, int npfx)
{
    int i;

    if(p == nil)
        return 0;
```

```
    if(npfx == 0)
        return 1;
    for(i = 0; i < npfx; i++){
        if(strncmp(p, pfx[i], pfxlen[i]) != 0)
            continue;
        if(p[pfxlen[i]] == '/' || p[pfxlen[i]] == 0)
            return 1;
        if(strcmp(pfx[i], ".") == 0 || *pfx[i] == 0)
            return 1;
    }
    return 0;
}
```

⟨*function* reporel 230a⟩≡ (232)

```
char*
reporel(char *s)
{
    char *p;
    int n;

    if(*s == '/')
        s = estrdup(s);
    else if((s = smprint("%s/%s", wdirpath, s)) == nil)
        sysfatal("smprint: %r");

    p = cleanname(s);
    n = strlen(repopath);
    if(strncmp(s, repopath, n) != 0)
        sysfatal("path outside repo: %s", s);
    p += n;
    if(*p == '/')
        p++;
    memmove(s, p, strlen(p)+1);
    return s;
}
```

⟨*function* show (git9/walk.c) 230b⟩≡ (232)

```
void
show(Biobuf *o, int flg, char *str, char *path)
{
    char *pa, *pb;
    int n;

    dirty |= flg;
    if(!quiet && (printflg & flg)){
        Bprint(o, str);
        n = nslash;
        if(n){
            for(pa = relapath, pb = path; *pa && *pb; pa++, pb++){
                if(*pa != *pb)
                    break;
                if(*pa == '/'){
                    n--;
                    path = pb+1;
                }
            }
            while(n-- > 0)
                Bprint(o, "../");
        }
        Bprint(o, "%s\n", path);
```

```
    }
}
```

⟨*function* `findslashes` 231a⟩≡ (232)

```
void
findslashes(char *path)
{
    char *p;

    p = cleanname(path);
    if(p[0] == '.'){
        if(p[1] == '\0')
            return;
        else if(p[1] == '.' && (p[2] == '/' || p[2] == '\0'))
            sysfatal("relative path escapes git root");
    }

    snprint(relapath, sizeof relapath, "%s/", p);
    p = relapath;
    if(*p == '/')
        p++;

    for(; *p; p++)
        if(*p == '/')
            nslash++;
}
```

⟨*function* `usage (git9/walk.c)` 231b⟩≡ (232)

```
void
usage(void)
{
    fprint(2, "usage: %s [-qbcI] [-f filt] [-b base] [paths...]\n", argv0);
    exits("usage");
}
```

⟨*function* `main (git9/walk.c)` 231c⟩≡ (232)

```
void
main(int argc, char **argv)
{
    char *p, *e, *ln, *base, **argrel, *parts[4], xbuf[8];
    int i, j, c, line, wfd, *argn;
    Biobuf *f, *o, *w;
    Hash h;
    Dir rn;

    gitinit(repopath, sizeof(repopath), &nrel);
    if(getwd(wdirpath, sizeof(wdirpath)) == nil)
        sysfatal("getwd: %r");
    if(chdir(repopath) == -1)
        sysfatal("chdir: %r");
    if(access(".git/fs/ctl", AEXIST) != 0)
        sysfatal("no running git/fs");
    ARGBEGIN{
    case 'q':
        quiet++;
        break;
    case 'c':
        rstr = "";
        tstr = "";
        mstr = "";
```

```
        astr = "";
        ustr = "";
        break;
    case 'f':
        for(p = EARGF(usage()); *p; p++)
            switch(*p){
            case 'T':   printflg |= Tflg;   break;
            case 'A':   printflg |= Aflg;   break;
            case 'M':   printflg |= Mflg;   break;
            case 'R':   printflg |= Rflg;   break;
            case 'U':   printflg |= Uflg;   break;
            default:    usage();        break;
        }
        break;
    case 'b':
        isindexed = 0;
        base = EARGF(usage());
        if(resolveref(&h, base) == -1)
            sysfatal("no such ref '%s'", base);
        bdir = smprint(".git/fs/object/%H/tree", h);
        break;
    case 'I':
        /* invalidate index */
        staleidx = 1;
        break;
    case 'r':
        findslashes(EARGF(usage()));
        break;
    default:
        usage();
    }ARGEND;
```

⟨git9/walk.c 232⟩≡

```
#include <u.h>
#include <libc.h>
#include "git.h"

typedef struct Idxed    Idxed;
typedef struct Idxent   Idxent;

⟨constant NCACHE 225a⟩

enum {
    Rflg    = 1 << 0,
    Mflg    = 1 << 1,
    Aflg    = 1 << 2,
    Uflg    = 1 << 3,
    /* everything after this is not an error */
    Tflg    = 1 << 4,
};

struct Idxed {
    char**  cache;
    int n;
    int max;
};

Idxed   idxtab[NCACHE];
char    repopath[1024];
char    wdirpath[1024];
```

```
char    relapath[1024];
char    *rstr   = "R ";
char    *mstr   = "M ";
char    *astr   = "A ";
char    *ustr   = "U ";
char    *tstr   = "T ";
char    *bdir = ".git/fs/HEAD/tree";
int nslash;
int nrel;
int quiet;
int dirty;
int isindexed = 1;
int intree;
int printflg;

Idxent  *idx;
int idxsz;
int nidx;

int cleanidx = 0;   /* skip tree check for checkedin() */
int staleidx = 0;

Idxent  *wdir;
int wdirsz;
int nwdir;

void    loadwdir(char*);
```

⟨*function* `checkedin` 225b⟩

⟨*function* `pathcmp` 226a⟩

⟨*function* `indexed` 226b⟩

⟨*function* `idxcmp (git9/walk.c)` 227a⟩

⟨*function* `samedata` 227b⟩

⟨*function* `loadent` 228⟩

⟨*function* `loadwdir` 229a⟩

⟨*function* `pfxmatch` 229b⟩

⟨*function* `reporel` 230a⟩

⟨*function* `show (git9/walk.c)` 230b⟩

⟨*function* `findslashes` 231a⟩

⟨*function* `usage (git9/walk.c)` 231b⟩

⟨*function* `main (git9/walk.c)` 231c⟩

```
    if(printflg == 0)
        printflg = Tflg | Aflg | Mflg | Rflg;

    argrel = emalloc(argc*sizeof(char*));
    argn = emalloc(argc*sizeof(int));
```

```
    for(i = 0; i < argc; i++){
        argrel[i] = reporel(argv[i]);
        argn[i] = strlen(argrel[i]);
    }

    if(isindexed && !staleidx){
        if((f = Bopen(".git/INDEX9", OREAD)) == nil){
            if(access(".git/index9", AEXIST) == 0){
                fprint(2, "index format conversion needed:\n");
                fprint(2, "\tcd %s && git/fs\n", repopath);
                fprint(2, "\t@{cd .git/index9/removed >[2]/dev/null && walk -f | sed 's/^/R NOQID 0 /'} >> .git
                fprint(2, "\t@{cd .git/fs/HEAD/tree && walk -f | sed 's/^/T NOQID 0 /'} >> .git/INDEX9\n");
                exits("noindex");
            }
            staleidx = 1;
            goto Stale;
        }

        nidx = 0;
        idxsz = 32;
        idx = emalloc(idxsz*sizeof(Idxent));

        line = 0;
        while((ln = Brdstr(f, '\n', 1)) != nil){
            line++;
            /* allow blank lines */
            if(ln[0] == 0 || ln[0] == '\n')
                continue;
            if(getfields(ln, parts, nelem(parts), 0, " \t") != nelem(parts))
                sysfatal(".git/INDEX9:%d: corrupt index", line);
            cleanname(parts[3]);
            if(nidx == idxsz){
                idxsz += idxsz/2;
                idx = erealloc(idx, idxsz*sizeof(Idxent));
            }
            idx[nidx].state = *parts[0];
            idx[nidx].qid = parseqid(parts[1]);
            idx[nidx].mode = strtol(parts[2], nil, 8);
            idx[nidx].path = estrdup(parts[3]);
            idx[nidx].order = nidx;
            nidx++;
            free(ln);
        }
        Bterm(f);
    } else {
Stale:
        nwdir = 0;
        wdirsz = 32;
        wdir = emalloc(wdirsz*sizeof(Idxent));

        if(chdir(bdir) == -1)
            sysfatal("chdir: %r");

        /* load whole tree into index when stale */
        intree = 1;
        if(staleidx || argc == 0)
            loadwdir(".");
        else for(i = 0; i < argc; i++)
            loadwdir(argrel[i]);
```

```
    if(chdir(repopath) == -1)
        sysfatal("chdir: %r");

    /* use as index */
    idx = wdir;
    nidx = nwdir;
    idxsz = wdirsz;

    cleanidx = 1;
}
qsort(idx, nidx, sizeof(Idxent), idxcmp);

nwdir = 0;
wdirsz = 32;
wdir = emalloc(wdirsz*sizeof(Idxent));

intree = 0;
if(argc == 0)
    loadwdir(".");
else for(i = 0; i < argc; i++)
    loadwdir(argrel[i]);
qsort(wdir, nwdir, sizeof(Idxent), idxcmp);

if((o = Bfdopen(1, OWRITE)) == nil)
    sysfatal("open out: %r");

i = 0;
j = 0;
while(i < nidx || j < nwdir){
    /* find the last entry we tracked for a path */
    while(i+1 < nidx && strcmp(idx[i].path, idx[i+1].path) == 0){
        staleidx = 1;
        i++;
    }
    while(j+1 < nwdir && strcmp(wdir[j].path, wdir[j+1].path) == 0)
        j++;
    if(i < nidx && !pfxmatch(idx[i].path, argrel, argn, argc)){
        i++;
        continue;
    }
    if(i >= nidx)
        c = 1;
    else if(j >= nwdir)
        c = -1;
    else
        c = strcmp(idx[i].path, wdir[j].path);
    /* exists in both index and on disk */
    if(c == 0){
        if(idx[i].state == 'R'){
            if(checkedin(&idx[i], 0))
                show(o, Rflg, rstr, idx[i].path);
            else{
                idx[i].state = 'U';
                staleidx = 1;
            }
        }else if(idx[i].state == 'A' && !checkedin(&idx[i], 1))
            show(o, Aflg, astr, idx[i].path);
        else if(!samedata(&idx[i], &wdir[j]))
            show(o, Mflg, mstr, idx[i].path);
        else
```

```
                show(o, Tflg, tstr, idx[i].path);
            i++;
            j++;
        /* only exists in index */
        }else if(c < 0){
            if(checkedin(&idx[i], 0))
                show(o, Rflg, rstr, idx[i].path);
            else{
                idx[i].state = 'U';
                staleidx = 1;
            }
            i++;
        /* only exists on disk */
        }else{
            if(checkedin(&wdir[j], 0)){
                if(samedata(nil, &wdir[j]))
                    show(o, Tflg, tstr, wdir[j].path);
                else
                    show(o, Mflg, mstr, wdir[j].path);
            }else if(printflg & Uflg && pfxmatch(wdir[j].path, argrel, argn, argc))
                show(o, Uflg, ustr, wdir[j].path);
            j++;
        }
    }
    Bterm(o);

    if(isindexed && staleidx)
    if((wfd = create(".git/INDEX9.new", OWRITE, 0644)) != -1){
        if((w = Bfdopen(wfd, OWRITE)) == nil){
            close(wfd);
            goto Nope;
        }
        for(i = 0; i < nidx; i++){
            while(i+1 < nidx && strcmp(idx[i].path, idx[i+1].path) == 0)
                i++;
            if(idx[i].state == 'U')
                continue;
            Bprint(w, "%c %Q %o %s\n",
                idx[i].state,
                idx[i].qid,
                idx[i].mode,
                idx[i].path);
        }
        Bterm(w);
        nulldir(&rn);
        rn.name = "INDEX9";
        if(remove(".git/INDEX9") == -1)
            if(access(".git/INDEX9", AEXIST) == 0)
                goto Nope;
        if(dirwstat(".git/INDEX9.new", &rn) == -1)
            sysfatal("rename: %r");
    }

Nope:
    if(!dirty)
        exits(nil);

    p = xbuf;
    e = p + sizeof(xbuf);
    for(i = 0; (1 << i) != Tflg; i++)
```

```
        if(dirty & (1 << i))
            p = seprint(p, e, "%c", "RMAUT"[i]);
    *p = '\0';
    exits(xbuf);
}
```

E.2 diff/

diff/diff.c

⟨*function* done*(diff)* 237a⟩≡ (238a)

```
void
done(int status)
{
    switch(status)
    {
    case 0:
        exits("");
    case 1:
        exits("some");
    default:
        exits("error");
    }
}
```

⟨*function* usage*(diff)* 237b⟩≡ (238a)

```
void
usage(void)
{
    fprint(2, "usage: %s [-abcefmnrw] file1 ... file2\n", argv0);
    exits("usage");
}
```

⟨*function* main*(diff)* 237c⟩≡ (238a)

```
void
main(int argc, char *argv[])
{
    int i;
    Dir *fsb, *tsb;

    Binit(&stdout, 1, OWRITE);
    ARGBEGIN{
    case 'e':
    case 'f':
    case 'n':
    case 'c':
    case 'a':
    case 'u':
        mode = ARGC();
        break;
    case 'w':
        bflag = 2;
        break;

    case 'b':
        bflag = 1;
        break;
```

```
        case 'r':
            rflag = 1;
            break;

        case 'm':
            mflag = 1;
            break;

        case 'h':
        default:
            usage();
        }ARGEND;
        if (argc < 2)
            usage();
        if ((tsb = dirstat(argv[argc-1])) == nil)
            sysfatal("can't stat %s", argv[argc-1]);
        if (argc > 2) {
            if (!DIRECTORY(tsb))
                sysfatal("not directory: %s", argv[argc-1]);
            mflag = 1;
        } else {
            if ((fsb = dirstat(argv[0])) == nil)
                sysfatal("can't stat %s", argv[0]);
            if (DIRECTORY(fsb) && DIRECTORY(tsb))
                mflag = 1;
            free(fsb);
        }
        free(tsb);
        for (i = 0; i < argc-1; i++)
            diff(argv[i], argv[argc-1], 0);

        done(anychange);
        /*NOTREACHED*/
    }
```

⟨diff/diff.c 238a⟩≡

```
    #include <u.h>
    #include <libc.h>
    #include <bio.h>

    #include "diff.h"
```

⟨*function* done*(diff)* 237a⟩

⟨*function* usage*(diff)* 237b⟩

⟨*function* main*(diff)* 237c⟩

diff/diff.h

⟨*constant* MAXPATHLEN*(diff)* 238b⟩≡ (239b)

```
    #define MAXPATHLEN  1024
```

⟨*constant* MAXLINELEN*(diff)* 238c⟩≡ (239b)

```
    #define MAXLINELEN  4096
```

⟨*macro* DIRECTORY*(diff)* 238d⟩≡ (239b)

```
    #define DIRECTORY(s)        ((s)->qid.type&QTDIR)
```

⟨*macro* REGULAR_FILE*(diff)* 239a⟩≡ (239b)

```
#define REGULAR_FILE(s)     ((s)->type == 'M' && !DIRECTORY(s))
```

⟨diff/diff.h 239b⟩≡

```
typedef struct Line Line;
typedef struct Cand Cand;
typedef struct Diff Diff;
typedef struct Change   Change;

struct Line {
    int serial;
    int value;
};

struct Cand {
    int x;
    int y;
    int pred;
};

struct Change
{
    int oldx;
    int oldy;
    int newx;
    int newy;
};

struct Diff {
    Cand cand;
    Line *file[2], line;
    int len[2];
    int binary;
    int bindiff;
    Line *sfile[2]; /*shortened by pruning common prefix and suffix*/
    int slen[2];
    int pref, suff; /*length of prefix and suffix*/
    int *class; /*will be overlaid on file[0]*/
    int *member;    /*will be overlaid on file[1]*/
    int *klist; /*will be overlaid on file[0] after class*/
    Cand *clist;    /* merely a free storage pot for candidates */
    int clen;
    int *J;     /*will be overlaid on class*/
    long *ixold;    /*will be overlaid on klist*/
    long *ixnew;    /*will be overlaid on file[1]*/
    char *file1;
    char *file2;
    Biobuf *input[2];
    Biobuf *b0;
    Biobuf *b1;
    int firstchange;
    Change *changes;
    int nchanges;
};

extern char mode;
extern char bflag;
extern char rflag;
extern char mflag;
extern int anychange;
```

```
extern Biobuf   stdout;

⟨constant MAXPATHLEN(diff) 238b⟩
⟨constant MAXLINELEN(diff) 238c⟩

⟨macro DIRECTORY(diff) 238d⟩
⟨macro REGULAR_FILE(diff) 239a⟩

int mkpathname(char *, char *, char *);
char *mktmpfile(int, Dir **);
char *statfile(char *, Dir **);
void *emalloc(unsigned);
void *erealloc(void *, unsigned);
void diff(char *, char *, int);
void diffreg(char*, char*, char*, char*);
void diffdir(char *, char *, int);
void calcdiff(Diff *, char *, char *, char *, char *);
Biobuf *prepare(Diff*, int, char *, char *);
void check(Diff *, Biobuf *, Biobuf *);
void change(Diff *, int, int, int, int);
void freediff(Diff *);
void flushchanges(Diff *);
void fetch(Diff *d, long *f, int a, int b, Biobuf *bp, char *s);
int readline(Biobuf*, char*, int);
```

diff/diffdir.c

⟨function itemcmp(diff) 240a⟩≡ (243a)

```
static int
itemcmp(void *v1, void *v2)
{
    char **d1 = v1, **d2 = v2;

    return strcmp(*d1, *d2);
}
```

⟨function scandir(diff) 240b⟩≡ (243a)

```
static char **
scandir(char *name)
{
    char **cp;
    Dir *db;
    int nitems;
    int fd, n;

    if ((fd = open(name, OREAD)) < 0) {
        fprint(2, "%s: can't open %s: %r\n", argv0, name);
        /* fake an empty directory */
        cp = emalloc(sizeof(char*));
        cp[0] = 0;
        return cp;
    }
    cp = 0;
    nitems = 0;
    if((n = dirreadall(fd, &db)) > 0){
        while (n--) {
            cp = erealloc(cp, (nitems+1)*sizeof(char*));
            cp[nitems] = emalloc(strlen((db+n)->name)+1);
            strcpy(cp[nitems], (db+n)->name);
```

```
            nitems++;
        }
        free(db);
    }
    cp = erealloc(cp, (nitems+1)*sizeof(char*));
    cp[nitems] = 0;
    close(fd);
    qsort((char *)cp, nitems, sizeof(char*), itemcmp);
    return cp;
}
```

⟨*function* `isdotordotdot` *(diff)* 241a⟩≡ (243a)

```
static int
isdotordotdot(char *p)
{
    if (*p == '.') {
        if (!p[1])
            return 1;
        if (p[1] == '.' && !p[2])
            return 1;
    }
    return 0;
}
```

⟨*function* `diffdir` *(diff)* 241b⟩≡ (243a)

```
void
diffdir(char *f, char *t, int level)
{
    char  **df, **dt, **dirf, **dirt;
    char *from, *to;
    int res;
    char fb[MAXPATHLEN+1], tb[MAXPATHLEN+1];

    df = scandir(f);
    dt = scandir(t);
    dirf = df;
    dirt = dt;
    while (*df || *dt) {
        from = *df;
        to = *dt;
        if (from && isdotordotdot(from)) {
            df++;
            continue;
        }
        if (to && isdotordotdot(to)) {
            dt++;
            continue;
        }
        if (!from)
            res = 1;
        else if (!to)
            res = -1;
        else
            res = strcmp(from, to);
        if (res < 0) {
            if (mode == 0 || mode == 'n')
                Bprint(&stdout, "Only in %s: %s\n", f, from);
            df++;
            continue;
        }
```

```
        if (res > 0) {
            if (mode == 0 || mode == 'n')
                Bprint(&stdout, "Only in %s: %s\n", t, to);
            dt++;
            continue;
        }
        if (mkpathname(fb, f, from))
            continue;
        if (mkpathname(tb, t, to))
            continue;
        diff(fb, tb, level+1);
        df++; dt++;
    }
    for (df = dirf; *df; df++)
        free(*df);
    for (dt = dirt; *dt; dt++)
        free(*dt);
    free(dirf);
    free(dirt);
}
```

⟨*function* `diff` *(diff)* 242⟩≡ (243a)

```
void
diff(char *f, char *t, int level)
{
    char *fp, *tp, *p, fb[MAXPATHLEN+1], tb[MAXPATHLEN+1];
    Dir *fsb, *tsb;

    fsb = nil;
    tsb = nil;
    if ((fp = statfile(f, &fsb)) == 0)
        goto Return;
    if ((tp = statfile(t, &tsb)) == 0)
        goto Return;
    if (DIRECTORY(fsb) && DIRECTORY(tsb)) {
        if (rflag || level == 0)
            diffdir(fp, tp, level);
        else
            Bprint(&stdout, "Common subdirectories: %s and %s\n", fp, tp);
    }
    else if (REGULAR_FILE(fsb) && REGULAR_FILE(tsb))
        diffreg(fp, f, tp, t);
    else {
        if (REGULAR_FILE(fsb)) {
            if ((p = utfrrune(f, '/')) == 0)
                p = f;
            else
                p++;
            if (mkpathname(tb, tp, p) == 0)
                diffreg(fp, f, tb, t);
        } else {
            if ((p = utfrrune(t, '/')) == 0)
                p = t;
            else
                p++;
            if (mkpathname(fb, fp, p) == 0)
                diffreg(fb, f, tp, t);
        }
    }
Return:
```

```
    free(fsb);
    free(tsb);
}
```

⟨diff/diffdir.c 243a⟩≡
```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include "diff.h"

⟨function itemcmp(diff) 240a⟩

⟨function scandir(diff) 240b⟩

⟨function isdotordotdot(diff) 241a⟩

⟨function diffdir(diff) 241b⟩

⟨function diff(diff) 242⟩
```

diff/diffio.c

⟨macro MIN(diff) 243b⟩≡ (249)
```
#define MIN(x, y)   ((x) < (y) ? (x): (y))
```

⟨function readline(diff) 243c⟩≡ (249)
```
int
readline(Biobuf *bp, char *buf, int nbuf)
{
    int c;
    char *p, *e;

    p = buf;
    e = p + nbuf-1;
    do {
        c = Bgetc(bp);
        if (c < 0) {
            if (p == buf)
                return -1;
            break;
        }
        *p++ = c;
        if (c == '\n')
            break;
    } while (p < e);
    *p = 0;
    if (c != '\n' && c >= 0) {
        do c = Bgetc(bp);
        while (c >= 0 && c != '\n');
    }
    return p - buf;
}
```

⟨constant HALFLONG(diff) 243d⟩≡ (249)
```
#define HALFLONG 16
```

⟨macro low(diff) 243e⟩≡ (249)
```
#define low(x)  (x&((1L<<HALFLONG)-1))
```

⟨*macro* high*(diff)* 244a⟩≡ (249)

```
#define high(x) (x>>HALFLONG)
```

⟨*function* readhash*(diff)* 244b⟩≡ (249)

```
/*
 * hashing has the effect of
 * arranging line in 7-bit bytes and then
 * summing 1-s complement in 16-bit hunks
 */
static int
readhash(Biobuf *bp, char *buf, int nbuf)
{
    long sum;
    unsigned shift;
    char *p;
    int len, space;

    sum = 1;
    shift = 0;
    if ((len = readline(bp, buf, nbuf)) == -1)
        return 0;
    p = buf;
    switch(bflag)   /* various types of white space handling */
    {
    case 0:
        while (len--) {
            sum += (long)*p++ << (shift &= (HALFLONG-1));
            shift += 7;
        }
        break;
    case 1:
        /*
         * coalesce multiple white-space
         */
        for (space = 0; len--; p++) {
            if (isspace(*p)) {
                space++;
                continue;
            }
            if (space) {
                shift += 7;
                space = 0;
            }
            sum += (long)*p << (shift &= (HALFLONG-1));
            shift += 7;
        }
        break;
    default:
        /*
         * strip all white-space
         */
        while (len--) {
            if (isspace(*p)) {
                p++;
                continue;
            }
            sum += (long)*p++ << (shift &= (HALFLONG-1));
            shift += 7;
        }
        break;
```

```
    }
    sum = low(sum) + high(sum);
    return ((short)low(sum) + (short)high(sum));
}
```

⟨*function* prepare*(diff)* 245a⟩≡ (249)

```
Biobuf *
prepare(Diff *d, int i, char *arg, char *orig)
{
    Line *p;
    int j, h;
    Biobuf *bp;
    char *cp, buf[MAXLINELEN];
    int nbytes;
    Rune r;

    if (i == 0) {
        d->file1 = orig;
        d->firstchange = 0;
    } else
        d->file2 = orig;
    bp = Bopen(arg, OREAD);
    if (!bp)
        sysfatal("cannot open %s: %r", arg);
    if (d->binary)
        return bp;
    nbytes = Bread(bp, buf, MIN(1024, MAXLINELEN));
    if (nbytes > 0) {
        cp = buf;
        while (cp < buf+nbytes-UTFmax) {
            /*
             * heuristic for a binary file in the
             * brave new UNICODE world
             */
            cp += chartorune(&r, cp);
            if (r == 0 || (r > 0x7f && r <= 0xa0)) {
                d->binary++;
                return bp;
            }
        }
        Bseek(bp, 0, 0);
    }
    p = emalloc(3*sizeof(Line));
    for (j = 0; h = readhash(bp, buf, sizeof(buf)); p[j].value = h)
        p = erealloc(p, (++j+3)*sizeof(Line));
    d->len[i] = j;
    d->file[i] = p;
    d->input[i] = bp;
    return bp;
}
```

⟨*function* squishspace*(diff)* 245b⟩≡ (249)

```
static int
squishspace(char *buf)
{
    char *p, *q;
    int space;

    for (space = 0, q = p = buf; *q; q++) {
        if (isspace(*q)) {
```

```
            space++;
            continue;
        }
        if (space && bflag == 1) {
            *p++ = ' ';
            space = 0;
        }
        *p++ = *q;
    }
    *p = 0;
    return p - buf;
}
```

⟨*function* `check` *(diff)* 246a⟩≡ (249)

```
/*
 * need to fix up for unexpected EOF's
 */
void
check(Diff *d, Biobuf *bf, Biobuf *bt)
{
    int f, t, flen, tlen;
    char fbuf[MAXLINELEN], tbuf[MAXLINELEN];

    d->ixold[0] = 0;
    d->ixnew[0] = 0;
    for (f = t = 1; f < d->len[0]; f++) {
        flen = readline(bf, fbuf, sizeof(fbuf));
        d->ixold[f] = d->ixold[f-1] + flen;     /* ftell(bf) */
        if (d->J[f] == 0)
            continue;
        do {
            tlen = readline(bt, tbuf, sizeof(tbuf));
            d->ixnew[t] = d->ixnew[t-1] + tlen; /* ftell(bt) */
        } while (t++ < d->J[f]);
        if (bflag) {
            flen = squishspace(fbuf);
            tlen = squishspace(tbuf);
        }
        if (flen != tlen || strcmp(fbuf, tbuf))
            d->J[f] = 0;
    }
    while (t < d->len[1]) {
        tlen = readline(bt, tbuf, sizeof(tbuf));
        d->ixnew[t] = d->ixnew[t-1] + tlen; /* fseek(bt) */
        t++;
    }
}
```

⟨*function* `range` *(diff)* 246b⟩≡ (249)

```
static void
range(int a, int b, char *separator)
{
    Bprint(&stdout, "%d", a > b ? b: a);
    if (a < b)
        Bprint(&stdout, "%s%d", separator, b);
}
```

⟨*function* `fetch` *(diff)* 246c⟩≡ (249)

```
void
fetch(Diff *d, long *f, int a, int b, Biobuf *bp, char *s)
```

```
{
    char buf[MAXLINELEN];
    int maxb, len;

    if(a <= 1)
        a = 1;
    if(bp == d->input[0])
        maxb = d->len[0];
    else
        maxb = d->len[1];
    if(b > maxb)
        b = maxb;
    if(a > maxb)
        return;
    Bseek(bp, f[a-1], 0);
    while (a++ <= b) {
        len = readline(bp, buf, sizeof(buf));
        if(len == 0 || buf[len-1] != '\n'){
            Bprint(&stdout, "%s%s\n", s, buf);
            Bprint(&stdout, "\\ No newline at end of file\n");
        }else
            Bprint(&stdout, "%s%s", s, buf);
    }
}
```

⟨*function* change*(diff)* 247⟩≡ (249)

```
void
change(Diff *df, int a, int b, int c, int d)
{
    char verb;
    char buf[4];
    Change *ch;

    if (a > b && c > d)
        return;
    anychange = 1;
    if (mflag && df->firstchange == 0) {
        if(mode) {
            buf[0] = '-';
            buf[1] = mode;
            buf[2] = ' ';
            buf[3] = '\0';
        } else {
            buf[0] = '\0';
        }
        Bprint(&stdout, "diff %s%s %s\n", buf, df->file1, df->file2);
        df->firstchange = 1;
    }
    verb = a > b ? 'a': c > d ? 'd': 'c';
    switch(mode) {
    case 'e':
        range(a, b, ",");
        Bputc(&stdout, verb);
        break;
    case 0:
        range(a, b, ",");
        Bputc(&stdout, verb);
        range(c, d, ",");
        break;
    case 'n':
```

```
        Bprint(&stdout, "%s:", df->file1);
        range(a, b, ",");
        Bprint(&stdout, " %c ", verb);
        Bprint(&stdout, "%s:", df->file2);
        range(c, d, ",");
        break;
    case 'f':
        Bputc(&stdout, verb);
        range(a, b, " ");
        break;
    case 'c':
    case 'a':
    case 'u':
        if(df->nchanges%1024 == 0)
            df->changes = erealloc(df->changes, (df->nchanges+1024)*sizeof(df->changes[0]));
        ch = &df->changes[df->nchanges++];
        ch->oldx = a;
        ch->oldy = b;
        ch->newx = c;
        ch->newy = d;
        return;
    }
    Bputc(&stdout, '\n');
    if (mode == 0 || mode == 'n') {
        fetch(df, df->ixold, a, b, df->input[0], "< ");
        if (a <= b && c <= d)
            Bprint(&stdout, "---\n");
    }
    fetch(df, df->ixnew, c, d, df->input[1], mode == 0 || mode == 'n' ? "> ": "");
    if (mode != 0 && mode != 'n' && c <= d)
        Bprint(&stdout, ".\n");
}
```

⟨*function* `changeset` *(diff)* 248a⟩≡ (249)

```
int
changeset(Diff *d, int i)
{
    while(i < d->nchanges && d->changes[i].oldy + 1 + 2*Lines > d->changes[i+1].oldx)
        i++;
    if(i < d->nchanges)
        return i+1;
    return d->nchanges;
}
```

⟨*function* `flushchanges` *(diff)* 248b⟩≡ (249)

```
void
flushchanges(Diff *df)
{
    vlong a, b, c, d, at, hdr;
    vlong i, j;

    if(df->nchanges == 0)
        return;

    hdr = 0;
    for(i=0; i < df->nchanges; ){
        j = changeset(df, i);
        a = df->changes[i].oldx - Lines;
        b = df->changes[j-1].oldy + Lines;
        c = df->changes[i].newx - Lines;
```

```
        d = df->changes[j-1].newy + Lines;
        if(a < 1)
            a = 1;
        if(c < 1)
            c = 1;
        if(b > df->len[0])
            b = df->len[0];
        if(d > df->len[1])
            d = df->len[1];
        if(mode == 'a'){
            a = 1;
            b = df->len[0];
            c = 1;
            d = df->len[1];
            j = df->nchanges;
        }
        if(mode == 'u'){
            if(!hdr){
                Bprint(&stdout, "--- %s\n", df->file1);
                Bprint(&stdout, "+++ %s\n", df->file2);
                hdr = 1;
            }
            Bprint(&stdout, "@@ -%lld,%lld +%lld,%lld @@\n", a, b-a+1, c, d-c+1);
        }else{
            Bprint(&stdout, "%s:", df->file1);
            range(a, b, ",");
            Bprint(&stdout, " - ");
            Bprint(&stdout, "%s:", df->file2);
            range(c, d, ",");
            Bputc(&stdout, '\n');
        }
        at = a;
        for(; i<j; i++){
            fetch(df, df->ixold, at, df->changes[i].oldx-1, df->input[0], mode == 'u' ? " " : "  ");
            fetch(df, df->ixold, df->changes[i].oldx, df->changes[i].oldy, df->input[0], mode == 'u' ? "-" : "-
            fetch(df, df->ixnew, df->changes[i].newx, df->changes[i].newy, df->input[1], mode == 'u' ? "+" : "+
            at = df->changes[i].oldy+1;
        }
        fetch(df, df->ixold, at, b, df->input[0], mode == 'u' ? " " : "  ");
    }
    df->nchanges = 0;
}
```

⟨diff/diffio.c 249⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>
#include "diff.h"

⟨macro MIN(diff) 243b⟩

⟨function readline(diff) 243c⟩

⟨constant HALFLONG(diff) 243d⟩
⟨macro low(diff) 243e⟩
⟨macro high(diff) 244a⟩

⟨function readhash(diff) 244b⟩
```

⟨*function* prepare*(diff)* 245a⟩

⟨*function* squishspace*(diff)* 245b⟩

⟨*function* check*(diff)* 246a⟩

⟨*function* range*(diff)* 246b⟩

⟨*function* fetch*(diff)* 246c⟩

⟨*function* change*(diff)* 247⟩

```
enum
{
    Lines = 3,  /* number of lines of context shown */
};
```

⟨*function* changeset*(diff)* 248a⟩

⟨*function* flushchanges*(diff)* 248b⟩

diff/diffreg.c

⟨*function* sort*(diff)* 250a⟩≡ (256c)

```
static void
sort(Line *a, int n)    /*shellsort CACM #201*/
{
    int m;
    Line *ai, *aim, *j, *k;
    Line w;
    int i;

    m = 0;
    for (i = 1; i <= n; i *= 2)
        m = 2*i - 1;
    for (m /= 2; m != 0; m /= 2) {
        k = a+(n-m);
        for (j = a+1; j <= k; j++) {
            ai = j;
            aim = ai+m;
            do {
                if (aim->value > ai->value ||
                   aim->value == ai->value &&
                   aim->serial > ai->serial)
                    break;
                w = *ai;
                *ai = *aim;
                *aim = w;

                aim = ai;
                ai -= m;
            } while (ai > a && aim >= ai);
        }
    }
}
```

⟨*function* unsort*(diff)* 250b⟩≡ (256c)

```
static void
unsort(Line *f, int l, int *b)
```

```
{
    int *a;
    int i;

    a = malloc((l+1)*sizeof(int));
    for(i=1;i<=l;i++)
        a[f[i].serial] = f[i].value;
    for(i=1;i<=l;i++)
        b[i] = a[i];
    free(a);
}
```

⟨*function* prune*(diff)* 251a⟩≡ (256c)

```
static void
prune(Diff *d)
{
    int i,j;

    for(d->pref = 0; d->pref < d->len[0] && d->pref < d->len[1] &&
        d->file[0][d->pref+1].value == d->file[1][d->pref+1].value;
        d->pref++) ;
    for(d->suff=0; d->suff < d->len[0] - d->pref && d->suff < d->len[1] - d->pref &&
        d->file[0][d->len[0] - d->suff].value == d->file[1][d->len[1] - d->suff].value;
        d->suff++) ;
    for(j=0;j<2;j++) {
        d->sfile[j] = d->file[j]+d->pref;
        d->slen[j] = d->len[j]-d->pref-d->suff;
        for(i=0;i<=d->slen[j];i++)
            d->sfile[j][i].serial = i;
    }
}
```

⟨*function* equiv*(diff)* 251b⟩≡ (256c)

```
static void
equiv(Line *a, int n, Line *b, int m, int *c)
{
    int i, j;

    i = j = 1;
    while(i<=n && j<=m) {
        if(a[i].value < b[j].value)
            a[i++].value = 0;
        else if(a[i].value == b[j].value)
            a[i++].value = j;
        else
            j++;
    }
    while(i <= n)
        a[i++].value = 0;
    b[m+1].value = 0;
    j = 0;
    while(++j <= m) {
        c[j] = -b[j].serial;
        while(b[j+1].value == b[j].value) {
            j++;
            c[j] = b[j].serial;
        }
    }
    c[j] = -1;
}
```

⟨*function* `newcand`*(diff)* 252a⟩≡ (256c)

```
static int
newcand(Diff *d, int x, int  y, int pred)
{
    Cand *q;

    d->clist = erealloc(d->clist, (d->clen+1)*sizeof(Cand));
    q = d->clist + d->clen;
    q->x = x;
    q->y = y;
    q->pred = pred;
    return d->clen++;
}
```

⟨*function* `search`*(diff)* 252b⟩≡ (256c)

```
static int
search(Diff *d, int *c, int k, int y)
{
    int i, j, l;
    int t;

    if(d->clist[c[k]].y < y)    /*quick look for typical case*/
        return k+1;
    i = 0;
    j = k+1;
    while((l=(i+j)/2) > i) {
        t = d->clist[c[l]].y;
        if(t > y)
            j = l;
        else if(t < y)
            i = l;
        else
            return l;
    }
    return l+1;
}
```

⟨*function* `stone`*(diff)* 252c⟩≡ (256c)

```
static int
stone(Diff *d, int *a, int n, int *b, int *c)
{
    int i, k,y;
    int j, l;
    int oldc, tc;
    int oldl;

    k = 0;
    c[0] = newcand(d, 0, 0, 0);
    for(i=1; i<=n; i++) {
        j = a[i];
        if(j==0)
            continue;
        y = -b[j];
        oldl = 0;
        oldc = c[0];
        do {
            if(y <= d->clist[oldc].y)
                continue;
            l = search(d, c, k, y);
            if(l!=oldl+1)
```

```
                oldc = c[l-1];
            if(l<=k) {
                if(d->clist[c[l]].y <= y)
                    continue;
                tc = c[l];
                c[l] = newcand(d, i, y, oldc);
                oldc = tc;
                oldl = l;
            } else {
                c[l] = newcand(d, i,y,oldc);
                k++;
                break;
            }
        } while((y=b[++j]) > 0);
    }
    return k;
}
```

⟨*function* unravel*(diff)* 253a⟩≡ (256c)

```
static void
unravel(Diff *d, int p)
{
    int i;
    Cand *q;

    for(i=0; i<=d->len[0]; i++) {
        if (i <= d->pref)
            d->J[i] = i;
        else if (i > d->len[0]-d->suff)
            d->J[i] = i+d->len[1] - d->len[0];
        else
            d->J[i] = 0;
    }
    for(q=d->clist+p; q->y != 0; q= d->clist + q->pred)
        d->J[q->x+d->pref] = q->y+d->pref;
}
```

⟨*constant* BUF*(diff)* 253b⟩≡ (256c)

```
#define BUF 4096
```

⟨*function* cmp*(diff)* 253c⟩≡ (256c)

```
static int
cmp(Biobuf* b1, Biobuf* b2)
{
    int n;
    uchar buf1[BUF], buf2[BUF];
    int f1, f2;
    vlong nc = 1;
    uchar *b1s, *b1e, *b2s, *b2e;

    f1 = Bfildes(b1);
    f2 = Bfildes(b2);
    seek(f1, 0, 0);
    seek(f2, 0, 0);
    b1s = b1e = buf1;
    b2s = b2e = buf2;
    for(;;){
        if(b1s >= b1e){
            if(b1s >= &buf1[BUF])
                b1s = buf1;
```

```
            n = read(f1, b1s,  &buf1[BUF] - b1s);
            b1e = b1s + n;
        }
        if(b2s >= b2e){
            if(b2s >= &buf2[BUF])
                b2s = buf2;
            n = read(f2, b2s,  &buf2[BUF] - b2s);
            b2e = b2s + n;
        }
        n = b2e - b2s;
        if(n > b1e - b1s)
            n = b1e - b1s;
        if(n <= 0)
            break;
        if(memcmp((void *)b1s, (void *)b2s, n) != 0){
            return 1;
        }
        nc += n;
        b1s += n;
        b2s += n;
    }
    if(b1e - b1s == b2e - b2s)
        return 0;
    return 1;
}
```

⟨*function* `calcdiff` *(diff)* 254⟩≡ (256c)

```
void
calcdiff(Diff *d, char *f, char *fo, char *t, char *to)
{
    Biobuf *b0, *b1;
    int k;

    b0 = prepare(d, 0, f, fo);
    if (!b0)
        return;
    b1 = prepare(d, 1, t, to);
    if (!b1) {
        Bterm(b0);
        return;
    }
    if (d->binary){
        // could use b0 and b1 but this is simpler.
        if(cmp(b0, b1))
            d->bindiff = 1;
        Bterm(b0);
        Bterm(b1);
        return;
    }
    d->clen = 0;
    prune(d);
    sort(d->sfile[0], d->slen[0]);
    sort(d->sfile[1], d->slen[1]);

    d->member = (int *)d->file[1];
    equiv(d->sfile[0], d->slen[0], d->sfile[1], d->slen[1], d->member);
    d->member = erealloc(d->member, (d->slen[1]+2)*sizeof(int));

    d->class = (int *)d->file[0];
    unsort(d->sfile[0], d->slen[0], d->class);
```

```
    d->class = erealloc(d->class, (d->slen[0]+2)*sizeof(int));

    d->klist = emalloc((d->slen[0]+2)*sizeof(int));
    d->clist = emalloc(sizeof(Cand));
    k = stone(d, d->class, d->slen[0], d->member, d->klist);
    free(d->member);
    free(d->class);

    d->J = emalloc((d->len[0]+2)*sizeof(int));
    unravel(d, d->klist[k]);
    free(d->clist);
    free(d->klist);

    d->ixold = emalloc((d->len[0]+2)*sizeof(long));
    d->ixnew = emalloc((d->len[1]+2)*sizeof(long));
    Bseek(b0, 0, 0); Bseek(b1, 0, 0);
    check(d, b0, b1);
}
```

⟨*function* output*(diff)* 255⟩≡ (256c)

```
static void
output(Diff *d)
{
    int m, i0, i1, j0, j1;

    if(d->bindiff)
        print("binary files %s %s differ\n", d->file1, d->file2);
    if(d->binary)
        return;
    m = d->len[0];
    d->J[0] = 0;
    d->J[m+1] = d->len[1]+1;
    if (mode != 'e') {
        for (i0 = 1; i0 <= m; i0 = i1+1) {
            while (i0 <= m && d->J[i0] == d->J[i0-1]+1)
                i0++;
            j0 = d->J[i0-1]+1;
            i1 = i0-1;
            while (i1 < m && d->J[i1+1] == 0)
                i1++;
            j1 = d->J[i1+1]-1;
            d->J[i1] = j1;
            change(d, i0, i1, j0, j1);
        }
    } else {
        for (i0 = m; i0 >= 1; i0 = i1-1) {
            while (i0 >= 1 && d->J[i0] == d->J[i0+1]-1 && d->J[i0])
                i0--;
            j0 = d->J[i0+1]-1;
            i1 = i0+1;
            while (i1 > 1 && d->J[i1-1] == 0)
                i1--;
            j1 = d->J[i1-1]+1;
            d->J[i1] = j1;
            change(d, i1 , i0, j1, j0);
        }
    }
    if (m == 0)
        change(d, 1, 0, 1, d->len[1]);
    flushchanges(d);
```

```
}
```

⟨*function* `diffreg` *(diff)* 256a⟩≡ (256c)

```
void
diffreg(char *f, char *fo, char *t, char *to)
{
    Diff d;

    memset(&d, 0, sizeof(d));
    calcdiff(&d, f, fo, t, to);
    output(&d);
    freediff(&d);
}
```

⟨*function* `freediff` *(diff)* 256b⟩≡ (256c)

```
void
freediff(Diff *d)
{
    if(d->input[0] != nil)
        Bterm(d->input[0]);
    if(d->input[1] != nil)
        Bterm(d->input[1]);
    free(d->J);
    free(d->ixold);
    free(d->ixnew);
}
```

⟨`diff/diffreg.c` 256c⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>

#include "diff.h"

/*  diff - differential file comparison
*
*   Uses an algorithm due to Harold Stone, which finds
*   a pair of longest identical subsequences in the two
*   files.
*
*   The major goal is to generate the match vector J.
*   J[i] is the index of the line in file1 corresponding
*   to line i file0. J[i] = 0 if there is no
*   such line in file1.
*
*   Lines are hashed so as to work in core. All potential
*   matches are located by sorting the lines of each file
*   on the hash (called value). In particular, this
*   collects the equivalence classes in file1 together.
*   Subroutine equiv replaces the value of each line in
*   file0 by the index of the first element of its
*   matching equivalence in (the reordered) file1.
*   To save space equiv squeezes file1 into a single
*   array member in which the equivalence classes
*   are simply concatenated, except that their first
*   members are flagged by changing sign.
*
*   Next the indices that point into member are unsorted into
*   array class according to the original order of file0.
*
```

```
*   The cleverness lies in routine stone. This marches
*   through the lines of file0, developing a vector klist
*   of "k-candidates". At step i a k-candidate is a matched
*   pair of lines x,y (x in file0 y in file1) such that
*   there is a common subsequence of lenght k
*   between the first i lines of file0 and the first y
*   lines of file1, but there is no such subsequence for
*   any smaller y. x is the earliest possible mate to y
*   that occurs in such a subsequence.
*
*   Whenever any of the members of the equivalence class of
*   lines in file1 matable to a line in file0 has serial number
*   less than the y of some k-candidate, that k-candidate
*   with the smallest such y is replaced. The new
*   k-candidate is chained (via pred) to the current
*   k-1 candidate so that the actual subsequence can
*   be recovered. When a member has serial number greater
*   that the y of all k-candidates, the klist is extended.
*   At the end, the longest subsequence is pulled out
*   and placed in the array J by unravel.
*
*   With J in hand, the matches there recorded are
*   check'ed against reality to assure that no spurious
*   matches have crept in due to hashing. If they have,
*   they are broken, and "jackpot " is recorded--a harmless
*   matter except that a true match for a spuriously
*   mated line may now be unnecessarily reported as a change.
*
*   Much of the complexity of the program comes simply
*   from trying to minimize core utilization and
*   maximize the range of doable problems by dynamically
*   allocating what is needed and reusing what is not.
*   The core requirements for problems larger than somewhat
*   are (in words) 2*length(file0) + length(file1) +
*   3*(number of k-candidates installed),  typically about
*   6n words for files of length n.
*/
```

⟨*function* `sort` *(diff)* 250a⟩

⟨*function* `unsort` *(diff)* 250b⟩

⟨*function* `prune` *(diff)* 251a⟩

⟨*function* `equiv` *(diff)* 251b⟩

⟨*function* `newcand` *(diff)* 252a⟩

⟨*function* `search` *(diff)* 252b⟩

⟨*function* `stone` *(diff)* 252c⟩

⟨*function* `unravel` *(diff)* 253a⟩

⟨*constant* `BUF` *(diff)* 253b⟩
⟨*function* `cmp` *(diff)* 253c⟩

⟨*function* `calcdiff` *(diff)* 254⟩

⟨*function* `output` *(diff)* 255⟩

⟨*function* `diffreg`*(diff)* 256a⟩

⟨*function* `freediff`*(diff)* 256b⟩

diff/merge3.c

⟨*macro* `min`*(diff)* 258a⟩≡ (261c)
```
#define min(a, b) ((a)<(b)?(a):(b))
```

⟨*macro* `max`*(diff)* 258b⟩≡ (261c)
```
#define max(a, b) ((a)>(b)?(a):(b))
```

⟨*function* `changecmp`*(diff)* 258c⟩≡ (261c)
```
static int
changecmp(void *a, void *b)
{
    return ((Change*)a)->oldx - ((Change*)b)->oldx;
}
```

⟨*function* `addchange`*(diff)* 258d⟩≡ (261c)
```
static void
addchange(Diff *df, int a, int b, int c, int d)
{
    Change *ch;

    if (a > b && c > d)
        return;
    if(df->nchanges%1024 == 0)
        df->changes = erealloc(df->changes, (df->nchanges+1024)*sizeof(df->changes[0]));
    ch = &df->changes[df->nchanges++];
    ch->oldx = a;
    ch->oldy = b;
    ch->newx = c;
    ch->newy = d;
}
```

⟨*function* `collect`*(diff)* 258e⟩≡ (261c)
```
static void
collect(Diff *d)
{
    int m, i0, i1, j0, j1;

    m = d->len[0];
    d->J[0] = 0;
    d->J[m+1] = d->len[1]+1;
    for (i0 = m; i0 >= 1; i0 = i1-1) {
        while (i0 >= 1 && d->J[i0] == d->J[i0+1]-1 && d->J[i0])
            i0--;
        j0 = d->J[i0+1]-1;
        i1 = i0+1;
        while (i1 > 1 && d->J[i1-1] == 0)
            i1--;
        j1 = d->J[i1-1]+1;
        d->J[i1] = j1;
        addchange(d, i1 , i0, j1, j0);
    }
    if (m == 0)
        addchange(d, 1, 0, 1, d->len[1]);
    qsort(d->changes, d->nchanges, sizeof(Change), changecmp);
}
```

⟨*function* overlaps*(diff)* 259a⟩≡ (261c)

```
static int
overlaps(int lx, int ly, int rx, int ry)
{
    if(lx <= rx)
        return ly >= rx;
    else
        return ry >= lx;
}
```

⟨*function* same*(diff)* 259b⟩≡ (261c)

```
static int
same(Diff *l, Change *lc, Diff *r, Change *rc)
{
    char lbuf[MAXLINELEN], rbuf[MAXLINELEN];
    int i, ll, rl, lx, ly, rx, ry;

    lx = lc->newx;
    ly = lc->newy;
    rx = rc->newx;
    ry = rc->newy;
    if(ly - lx != ry - rx)
        return 0;
    assert(ly <= l->len[1] && ry <= r->len[1]);
    Bseek(l->input[1], l->ixnew[lx-1], 0);
    Bseek(r->input[1], r->ixnew[rx-1], 0);
    for(i = 0; i <= (ly - lx); i++){
        ll = readline(l->input[1], lbuf, sizeof(lbuf));
        rl = readline(r->input[1], rbuf, sizeof(rbuf));
        if(ll != rl)
            return 0;
        if(memcmp(lbuf, rbuf, ll) != 0)
            return 0;
    }
    return 1;
}
```

⟨*function* merge*(diff)* 259c⟩≡ (261c)

```
char*
merge(Diff *l, Diff *r)
{
    int lx, ly, rx, ry;
    int il, ir, delta;
    Change *lc, *rc;
    char *status;
    vlong ln;

    il = 0;
    ir = 0;
    ln = 0;
    status = nil;
    collect(l);
    collect(r);
    while(il < l->nchanges || ir < r->nchanges){
        lc = nil;
        rc = nil;
        lx = -1;
        ly = -1;
        rx = -1;
        ry = -1;
```

```
if(il < l->nchanges){
    lc = &l->changes[il];
    lx = (lc->oldx < lc->oldy) ? lc->oldx : lc->oldy;
    ly = (lc->oldx < lc->oldy) ? lc->oldy : lc->oldx;
}
if(ir < r->nchanges){
    rc = &r->changes[ir];
    rx = (rc->oldx < rc->oldy) ? rc->oldx : rc->oldy;
    ry = (rc->oldx < rc->oldy) ? rc->oldy : rc->oldx;
}
if(lc != nil && rc != nil && overlaps(lx, ly, rx, ry)){
    /*
     * align the edges of the chunks, expanding them
     * so that when we compare for sameness, we are
     * comparing same-sized chunks.
     */
    if(lc->oldx < rc->oldx){
        delta = rc->oldx - lc->oldx;
        rc->oldx = max(rc->oldx-delta, 1);
        rc->newx = max(rc->newx-delta, 1);
    }else{
        delta = lc->oldx - rc->oldx;
        lc->oldx = max(lc->oldx-delta, 1);
        lc->newx = max(lc->newx-delta, 1);
    }
    if(lc->oldy > rc->oldy){
        delta = lc->oldy - rc->oldy;
        rc->oldy = min(rc->oldy+delta, r->len[0]);
        rc->newy = min(rc->newy+delta, r->len[1]);
    }else{
        delta = rc->oldy - lc->oldy;
        lc->oldy = min(lc->oldy+delta, l->len[0]);
        lc->newy = min(lc->newy+delta, l->len[1]);
    }
    if(same(l, lc, r, rc)){
        fetch(l, l->ixold, ln, lc->oldx-1, l->input[0], "");
        fetch(l, l->ixnew, lc->newx, lc->newy, l->input[1], "");
    }else{
        fetch(l, l->ixold, ln, lc->oldx-1, l->input[0], "");
        Bprint(&stdout, "<<<<<<<<<< %s\n", l->file2);
        fetch(l, l->ixnew, lc->newx, lc->newy, l->input[1], "");
        Bprint(&stdout, "========== original\n");
        fetch(l, l->ixold, lc->oldx, lc->oldy, l->input[0], "");
        Bprint(&stdout, "========== %s\n", r->file2);
        fetch(r, r->ixnew, rc->newx, rc->newy, r->input[1], "");
        Bprint(&stdout, ">>>>>>>>>>\n");
        status = "conflict";
    }
    ln = lc->oldy+1;
    il++;
    ir++;
}else if(lc != nil && (rc == nil || lx < rx)){
    fetch(l, l->ixold, ln, lc->oldx-1, l->input[0], "");
    fetch(l, l->ixnew, lc->newx, lc->newy, l->input[1], "");
    ln = lc->oldy+1;
    il++;
}else if(rc != nil && (lc == nil || rx < lx)){
    fetch(l, r->ixold, ln, rc->oldx-1, r->input[0], "");
    fetch(r, r->ixnew, rc->newx, rc->newy, r->input[1], "");
    ln = rc->oldy+1;
```

```
            ir++;
        }else
            abort();
    }
    if(ln <= l->len[0])
        fetch(l, l->ixold, ln, l->len[0], l->input[0], "");
    return status;
}
```

⟨*function* `usage (diff/merge3.c)` *(diff)* 261a⟩≡ (261c)

```
void
usage(void)
{
    fprint(2, "usage: %s theirs base ours\n", argv0);
    exits("usage");
}
```

⟨*function* `main (diff/merge3.c)` *(diff)* 261b⟩≡ (261c)

```
void
main(int argc, char **argv)
{
    Diff l, r;
    char *x;

    ARGBEGIN{
    default:
        usage();
    }ARGEND;

    if(argc != 3)
        usage();
    Binit(&stdout, 1, OWRITE);
    memset(&l, 0, sizeof(l));
    memset(&r, 0, sizeof(r));
    calcdiff(&l, argv[1], argv[1], argv[0], argv[0]);
    calcdiff(&r, argv[1], argv[1], argv[2], argv[2]);
    if(l.binary || r.binary)
        sysfatal("cannot merge binaries");
    x = merge(&l, &r);
    freediff(&l);
    freediff(&r);
    exits(x);
}
```

⟨`diff/merge3.c` 261c⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>

#include "diff.h"

⟨macro min(diff) 258a⟩
⟨macro max(diff) 258b⟩

⟨function changecmp(diff) 258c⟩

⟨function addchange(diff) 258d⟩

⟨function collect(diff) 258e⟩
```

⟨*function* `overlaps` *(diff)* 259a⟩

⟨*function* `same` *(diff)* 259b⟩

⟨*function* `merge` *(diff)* 259c⟩

⟨*function* `usage (diff/merge3.c)` *(diff)* 261a⟩

⟨*function* `main (diff/merge3.c)` *(diff)* 261b⟩

diff/util.c

⟨*function* `emalloc` *(diff)* 262a⟩≡ (263b)

```
void *
emalloc(unsigned n)
{
    register void *p;

    if ((p = malloc(n)) == 0)
        sysfatal("malloc: %r");
    return p;
}
```

⟨*function* `erealloc` *(diff)* 262b⟩≡ (263b)

```
void *
erealloc(void *p, unsigned n)
{
    void *rp;

    if ((rp = realloc(p, n)) == 0)
        sysfatal("realloc: %r");
    return rp;
}
```

⟨*function* `mkpathname` *(diff)* 262c⟩≡ (263b)

```
int
mkpathname(char *pathname, char *path, char *name)
{
    if (strlen(path) + strlen(name) > MAXPATHLEN)
        sysfatal("pathname %s/%s too long", path, name);
    sprint(pathname, "%s/%s", path, name);
    return 0;
}
```

⟨*function* `mktmpfile` *(diff)* 262d⟩≡ (263b)

```
char *
mktmpfile(int input, Dir **sb)
{
    int fd, i;
    char *p;
    char buf[8192];

    p = mktemp(tmp[whichtmp++]);
    /*
     * Because we want this file to stick around
     * for the entire run of the program, we leak
     * the fd intentionally here; when we exit,
     * the system will remove the file for us.
     */
```

```
    fd = create(p, OWRITE|ORCLOSE, 0600);
    if (fd < 0)
        sysfatal("cannot create %s: %r", p);
    while ((i = read(input, buf, sizeof(buf))) > 0) {
        if ((i = write(fd, buf, i)) < 0)
            break;
    }
    *sb = dirfstat(fd);
    if (i < 0)
        sysfatal("cannot read/write %s: %r", p);
    return p;
}
```

⟨*function* `statfile` *(diff)* 263a⟩≡ (263b)

```
char *
statfile(char *file, Dir **sb)
{
    Dir *dir;
    int input;

    dir = dirstat(file);
    if(dir == nil) {
        if (strcmp(file, "-") || (dir = dirfstat(0)) == nil)
            sysfatal("cannot stat %s: %r", file);
        free(dir);
        return mktmpfile(0, sb);
    } else if (!REGULAR_FILE(dir) && !DIRECTORY(dir)) {
        free(dir);
        if ((input = open(file, OREAD)) == -1)
            sysfatal("cannot open %s: %r", file);
        file = mktmpfile(input, sb);
        close(input);
    } else
        *sb = dir;
    return file;
}
```

⟨`diff/util.c` 263b⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include "diff.h"

Biobuf  stdout;
char    mode;           /* '\0', 'e', 'f', 'h' */
char    bflag;          /* ignore multiple and trailing blanks */
char    rflag;          /* recurse down directory trees */
char    mflag;          /* pseudo flag: doing multiple files, one dir */
int anychange;

static char *tmp[] = {"/tmp/diff1XXXXXXXXXXX", "/tmp/diff2XXXXXXXXXXX"};
static int whichtmp;
```

⟨*function* `emalloc` *(diff)* 262a⟩

⟨*function* `erealloc` *(diff)* 262b⟩

⟨*function* `mkpathname` *(diff)* 262c⟩

⟨*function* `mktmpfile` *(diff)* 262d⟩

⟨*function* `statfile` *(diff)* 263a⟩

E.3 misc/

misc/patch.c

⟨*function* `readline` *(patch.c)* 264a⟩≡ (277)

```
char*
readline(Biobuf *f, int *lnum)
{
    char *ln;

    if((ln = Brdstr(f, '\n', 0)) == nil)
        return nil;
    *lnum += 1;
    return ln;
}
```

⟨*function* `emalloc` *(patch.c)* 264b⟩≡ (277)

```
void *
emalloc(ulong n)
{
    void *v;

    v = mallocz(n, 1);
    if(v == nil)
        fail("malloc: %r");
    setmalloctag(v, getcallerpc(&n));
    return v;
}
```

⟨*function* `erealloc` *(patch.c)* 264c⟩≡ (277)

```
void *
erealloc(void *v, ulong n)
{
    if(n == 0)
        n++;
    v = realloc(v, n);
    if(v == nil)
        fail("malloc: %r");
    setmalloctag(v, getcallerpc(&n));
    return v;
}
```

⟨*function* `fail` *(patch.c)* 264d⟩≡ (277)

```
void
fail(char *fmt, ...)
{
    char msg[ERRMAX];
    va_list ap;

    finish(0);
    va_start(ap, fmt);
    vsnprint(msg, sizeof(msg), fmt, ap);
    va_end(ap);
    fprint(2, "%s\n", msg);
    exits(msg);
}
```

⟨*function* `fileheader` *(patch.c)* 265a⟩≡ (277)

```
int
fileheader(char *s, char *pfx, char **name)
{
    int len, n, nnull;
    char *e;

    if((strncmp(s, pfx, strlen(pfx))) != 0)
        return -1;
    for(s += strlen(pfx); *s; s++)
        if(!isspace(*s))
            break;
    for(e = s; *e; e++)
        if(isspace(*e))
            break;
    if(s == e)
        return -1;
    nnull = strlen("/dev/null");
    if((e - s) != nnull || strncmp(s, "/dev/null", nnull) != 0){
        n = strip;
        while(s != e && n > 0){
            while(s != e && *s == '/')
                s++;
            while(s != e && *s != '/')
                s++;
            n--;
        }
        while(*s == '/')
            s++;
        if(*s == '\0')
            fail("too many components stripped");
    }
    len = (e - s) + 1;
    *name = emalloc(len);
    strecpy(*name, *name + len, s);
    return 0;
}
```

⟨*function* `hunkheader` *(patch.c)* 265b⟩≡ (277)

```
int
hunkheader(Hunk *h, char *s, char *oldpath, char *newpath, int lnum)
{
    char *e;

    memset(h, 0, sizeof(*h));
    h->lnum = lnum;
    h->oldpath = strdup(oldpath);
    h->newpath = strdup(newpath);
    h->origlen = 0;
    h->origsz = 32;
    h->orig = emalloc(h->origsz);
    h->oldlen = 0;
    h->oldsz = 32;
    h->old = emalloc(h->oldsz);
    h->newlen = 0;
    h->newsz = 32;
    h->new = emalloc(h->newsz);
    cleanname(h->oldpath);
    cleanname(h->newpath);
    if(strncmp(s, "@@ -", 4) != 0)
```

```
        return -1;
    e = s + 4;
    h->oldln = strtol(e, &e, 10);
    h->oldcnt = 1;
    if(*e == ','){
        e++;
        h->oldcnt = strtol(e, &e, 10);
    }
    while(*e == ' ' || *e == '\t')
        e++;
    if(*e != '+')
        return -1;
    e++;
    h->newln = strtol(e, &e, 10);
    h->newcnt = 1;
    if(e == s)
        return -1;
    if(*e == ','){
        e++;
        h->newcnt = strtol(e, &e, 10);
    }
    if(e == s || *e != ' ')
        return -1;
    if(strncmp(e, " @@", 3) != 0)
        return -1;
    /*
     * empty files have line number 0: keep that,
     * otherwise adjust down.
     */
    if(h->oldln > 0)
        h->oldln--;
    if(h->newln > 0)
        h->newln--;
    if(h->oldln < 0 || h->newln < 0 || h->oldcnt < 0 || h->newcnt < 0)
        fail("malformed hunk %s", s);
    return 0;
}
```

⟨*function* addorig*(patch.c)* 266a⟩≡ (277)

```
void
addorig(Hunk *h, char *ln)
{
    int n;

    n = strlen(ln);
    while(h->origlen + n >= h->origsz){
        h->origsz *= 2;
        h->orig = erealloc(h->orig, h->origsz);
    }
    memcpy(h->orig + h->origlen, ln, n);
    h->origlen += n;
}
```

⟨*function* addnew*(patch.c)* 266b⟩≡ (277)

```
void
addnew(Hunk *h, char *ln)
{
    int n;

    ln++;
```

```
    n = strlen(ln);
    while(h->newlen + n >= h->newsz){
        h->newsz *= 2;
        h->new = erealloc(h->new, h->newsz);
    }
    memcpy(h->new + h->newlen, ln, n);
    h->newlen += n;
}
```

⟨*function* addold*(patch.c)* 267a⟩≡ (277)

```
void
addold(Hunk *h, char *ln)
{
    int n;

    ln++;
    n = strlen(ln);
    while(h->oldlen + n >= h->oldsz){
        h->oldsz *= 2;
        h->old = erealloc(h->old, h->oldsz);
    }
    memcpy(h->old + h->oldlen, ln, n);
    h->oldlen += n;
}
```

⟨*function* addmiss*(patch.c)* 267b⟩≡ (277)

```
int
addmiss(Hunk *h, char *ln, int *nold, int *nnew)
{
    if(ln == nil)
        return 1;
    else if(ln[0] != '-' && ln[0] != '+')
        return 0;
    if(ln[0] == '-'){
        addold(h, ln);
        *nold += 1;
    }else{
        addnew(h, ln);
        *nnew += 1;
    }
    return 1;
}
```

⟨*function* addhunk*(patch.c)* 267c⟩≡ (277)

```
void
addhunk(Patch *p, Hunk *h)
{
    p->hunk = erealloc(p->hunk, ++p->nhunk*sizeof(Hunk));
    p->hunk[p->nhunk-1] = *h;
}
```

⟨*function* hunkcmp*(patch.c)* 267d⟩≡ (277)

```
int
hunkcmp(void *a, void *b)
{
    int c;

    c = strcmp(((Hunk*)a)->oldpath, ((Hunk*)b)->oldpath);
    if(c != 0)
        return c;
    return ((Hunk*)a)->oldln - ((Hunk*)b)->oldln;
}
```

⟨*function* `swapint` *(patch.c)* 268a⟩≡ (277)

```
void
swapint(int *a, int *b)
{
    int t;

    t = *a;
    *a = *b;
    *b = t;
}
```

⟨*function* `swapstr` *(patch.c)* 268b⟩≡ (277)

```
void
swapstr(char **a, char **b)
{
    char *t;

    t = *a;
    *a = *b;
    *b = t;
}
```

⟨*function* `trimhunk` *(patch.c)* 268c⟩≡ (277)

```
void
trimhunk(char c, Hunk *h)
{
    if((c == ' ' || c == '-') && h->oldlen > 0 && h->old[h->oldlen-1] == '\n'){
        h->oldcnt--;
        h->oldlen--;
    }
    if((c == ' ' || c == '+') && h->newlen > 0 && h->new[h->newlen-1] == '\n'){
        h->newcnt--;
        h->newlen--;
    }
}
```

⟨*function* `parse` *(patch.c)* 268d⟩≡ (277)

```
Patch*
parse(Biobuf *f, char *name)
{
    char *ln, *old, *new, c;
    int i, oldcnt, newcnt, lnum;
    Patch *p;
    Hunk h, *ph;

    ln = nil;
    lnum = 0;
    p = emalloc(sizeof(Patch));
comment:
    free(ln);
    while((ln = readline(f, &lnum)) != nil){
        if(strncmp(ln, "--- ", 4) == 0)
            goto patch;
        free(ln);
    }
    if(p->nhunk == 0)
        fail("%s: could not find start of patch", name);
    goto out;

patch:
```

```
    if(fileheader(ln, "--- ", &old) == -1)
        goto comment;
    free(ln);

    if((ln = readline(f, &lnum)) == nil)
        goto out;
    if(fileheader(ln, "+++ ", &new) == -1)
        goto comment;
    free(ln);

    if((ln = readline(f, &lnum)) == nil)
        goto out;
hunk:
    oldcnt = 0;
    newcnt = 0;
    if(hunkheader(&h, ln, old, new, lnum) == -1)
        goto comment;
    free(ln);

    while(1){
        if((ln = readline(f, &lnum)) == nil){
            if(oldcnt != h.oldcnt)
                fail("%s:%d: malformed hunk: mismatched -hunk size %d != %d", name, lnum, oldcnt, h.oldcnt);
            if(newcnt != h.newcnt)
                fail("%s:%d: malformed hunk: mismatched +hunk size %d != %d", name, lnum, newcnt, h.newcnt);
            addhunk(p, &h);
            break;
        }
        c = ln[0];
        addorig(&h, ln);
        switch(ln[0]){
        default:
            fail("%s:%d: malformed hunk: leading junk", name, lnum);
        case '\\':
            if(strncmp(ln, "\\ No newline", nelem("\\ No newline")-1) == 0)
                trimhunk(c, &h);
            /* ignore unknown directives */
            break;
        case '-':
            addold(&h, ln);
            oldcnt++;
            break;
        case '+':
            addnew(&h, ln);
            newcnt++;
            break;
        case '\n':
            addold(&h, " \n");
            addnew(&h, " \n");
            oldcnt++;
            newcnt++;
            break;
        case ' ':
            addold(&h, ln);
            addnew(&h, ln);
            oldcnt++;
            newcnt++;
            break;
        }
        free(ln);
```

```
        if(oldcnt > h.oldcnt || newcnt > h.newcnt)
            fail("%s:%d: malformed hunk: oversized hunk", name, lnum);
        if(oldcnt < h.oldcnt || newcnt < h.newcnt)
            continue;

        addhunk(p, &h);
        if((ln = readline(f, &lnum)) == nil)
            goto out;
        if(strncmp(ln, "\\ No newline", nelem("\\ No newline")-1) == 0)
            trimhunk(c, &p->hunk[p->nhunk-1]);
        if(strncmp(ln, "--- ", 4) == 0)
            goto patch;
        if(strncmp(ln, "@@ ", 3) == 0)
            goto hunk;
        goto comment;
    }

out:
    if(reverse){
        for(i = 0; i < p->nhunk; i++){
            ph = &p->hunk[i];
            swapint(&ph->oldln, &ph->newln);
            swapint(&ph->oldcnt, &ph->newcnt);
            swapint(&ph->oldlen, &ph->newlen);
            swapint(&ph->oldsz, &ph->newsz);
            swapstr(&ph->oldpath, &ph->newpath);
            swapstr(&ph->old, &ph->new);
        }
    }
    qsort(p->hunk, p->nhunk, sizeof(Hunk), hunkcmp);
    free(old);
    free(new);
    free(ln);
    return p;
}
```

⟨*function* rename*(patch.c)* 270a⟩≡ (277)

```
int
rename(int fd, char *name)
{
    Dir st;
    char *p;

    nulldir(&st);
    if((p = strrchr(name, '/')) == nil)
        st.name = name;
    else
        st.name = p + 1;
    return dirfwstat(fd, &st);
}
```

⟨*function* mkpath*(patch.c)* 270b⟩≡ (277)

```
int
mkpath(char *path)
{
    char *p, buf[ERRMAX];
    int f;

    if(*path == '\0')
        return 0;
```

```
    for(p = strchr(path+1, '/'); p != nil; p = strchr(p+1, '/')){
        *p = '\0';
        if(access(path, AEXIST) != 0){
            if((f = create(path, OREAD, DMDIR | 0777)) == -1){
                rerrstr(buf, sizeof(buf));
                if(strstr(buf, "exist") == nil)
                    return -1;
            }
            close(f);
        }
        *p = '/';
    }
    return 0;
}
```

⟨*function* `blat` *(patch.c)* 271a⟩≡ (277)

```
void
blat(char *old, char *new, char *o, usize len, int mode)
{
    char *tmp;
    int fd;

    tmp = nil;
    if(strcmp(old, "/dev/null") == 0 && !dryrun)
        if(access(new, AEXIST) != -1)
            fail("would clobber: %s", new);
    if(strcmp(new, "/dev/null") != 0 && !dryrun){
        if(mkpath(new) == -1)
            fail("mkpath %s: %r", new);
        if((tmp = smprint("%s.tmp%d", new, getpid())) == nil)
            fail("smprint: %r");
        if((fd = create(tmp, OWRITE, mode|0200)) == -1)
            fail("open %s: %r", tmp);
        if(write(fd, o, len) != len)
            fail("write %s: %r", tmp);
        close(fd);
    }
    if((changed = realloc(changed, (nchanged+1)*sizeof(Fchg))) == nil)
        fail("realloc: %r");
    if((changed[nchanged].new = strdup(new)) == nil)
        fail("strdup: %r");
    if((changed[nchanged].old = strdup(old)) == nil)
        fail("strdup: %r");
    changed[nchanged].tmp = tmp;
    nchanged++;
}
```

⟨*function* `finish` *(patch.c)* 271b⟩≡ (277)

```
int
finish(int ok)
{
    Fchg *c;
    int i, fd;

    for(i = 0; i < nchanged; i++){
        c = &changed[i];
        if(!ok){
            if(c->tmp != nil && remove(c->tmp) == -1)
                fprint(2, "remove %s: %r\n", c->tmp);
            goto Free;
```

```
        }
        if(!dryrun){
            if(strcmp(c->new, "/dev/null") == 0){
                if(remove(c->old) == -1){
                    ok = 0;
                    fprint(2, "remove %s: %r\n", c->old);
                    goto Free;
                }
                goto Print;
            }
            if((fd = open(c->tmp, ORDWR)) == -1){
                ok = 0;
                fprint(2, "open %s: %r\n", c->tmp);
                goto Free;
            }
            if(strcmp(c->old, c->new) == 0 && remove(c->old) == -1
            || rename(fd, c->new) == -1
            || close(fd) == -1){
                ok = 0;
                fprint(2, "cleanup: %r");
            }
        }
Print:
        if(strcmp(c->new, "/dev/null") == 0)
            print("%s\n", c->old);
        else
            print("%s\n", c->new);
Free:
        free(c->tmp);
        free(c->old);
        free(c->new);
    }
    free(changed);
    return ok;
}
```

⟨*function* slurp*(patch.c)* 272⟩≡ (277)

```
int
slurp(Fbuf *f, char *path)
{
    int n, i, fd, sz, len, nlines, linesz;
    char *buf;
    int *lines;
    Dir *d;

    if((fd = open(path, OREAD)) == -1){
        fprint(2, "open %s: %r", path);
        return -1;
    }
    if((d = dirfstat(fd)) == nil)
        fail("stat %s: %r", path);
    sz = 8192;
    len = 0;
    buf = emalloc(sz);
    while(1){
        if(len == sz){
            sz *= 2;
            buf = erealloc(buf, sz);
        }
        n = read(fd, buf + len, sz - len);
```

```
        if(n == 0)
            break;
        if(n == -1)
            fail("read %s: %r", path);
        len += n;
    }

    nlines = 0;
    linesz = 32;
    lines = emalloc(linesz*sizeof(int));
    lines[nlines++] = 0;
    for(i = 0; i < len; i++){
        if(buf[i] != '\n')
            continue;
        if(nlines+2 == linesz){
            linesz *= 2;
            lines = erealloc(lines, linesz*sizeof(int));
        }
        lines[nlines++] = i+1;
    }
    lines[nlines] = len;
    f->len = len;
    f->buf = buf;
    f->lines = lines;
    f->nlines = nlines;
    f->lastln = 0;
    f->lastfuzz = 0;
    f->mode = d->mode;
    free(d);
    close(fd);
    return 0;
}
```

⟨*function* searchln*(patch.c)* 273a⟩≡ (277)

```
char*
searchln(Fbuf *f, Hunk *h, int ln)
{
    int off;

    off = f->lines[ln];
    if(off + h->oldlen > f->len)
        return nil;
    if(memcmp(f->buf + off, h->old, h->oldlen) != 0)
        return nil;
    f->lastln = ln + h->oldcnt;
    f->lastfuzz = ln - h->oldln;
    return f->buf + off;
}
```

⟨*function* search*(patch.c)* 273b⟩≡ (277)

```
char*
search(Fbuf *f, Hunk *h)
{
    int ln, oldln, fuzz, scanning;
    char *p;

    oldln = h->oldln + f->lastfuzz;
    if(oldln + h->oldcnt > f->nlines)
        oldln = f->nlines - h->oldcnt;
    scanning = oldln >= f->lastln;
```

```
    for(fuzz = 0; scanning && fuzz < 250; fuzz++){
        scanning = 0;
        ln = oldln - fuzz;
        if(ln >= f->lastln){
            scanning = 1;
            if((p = searchln(f, h, ln)) != nil)
                return p;
        }
        ln = oldln + fuzz + 1;
        if(ln + h->oldcnt <= f->nlines){
            scanning = 1;
            if((p = searchln(f, h, ln)) != nil)
                return p;
        }
    }
    return nil;
}
```

⟨*function* `rejected` *(patch.c)* 274a⟩≡ (277)

```
void
rejected(Hunk *h, char *fname)
{
    fprint(2, "%s:%d: skipping failed hunk %s:%d\n", fname, h->lnum, h->oldpath, h->oldln);
    fprint(rejfd, "--- %s:%d \n", h->oldpath, h->oldln);
    fprint(rejfd, "+++ %s:%d \n", h->newpath, h->newln);
    fprint(rejfd, "@@ -%d,%d +%d,%d @@\n", h->oldln, h->oldcnt, h->newln, h->newcnt);
    write(rejfd, h->orig, h->origlen);
}
```

⟨*function* `append` *(patch.c)* 274b⟩≡ (277)

```
char*
append(char *o, int *sz, char *s, char *e)
{
    int n;

    n = (e - s);
    o = erealloc(o, *sz + n);
    memcpy(o + *sz, s, n);
    *sz += n;
    return o;
}
```

⟨*function* `apply` *(patch.c)* 274c⟩≡ (277)

```
int
apply(Patch *p, char *fname)
{
    char *o, *s, *n, *curfile, *nextfile;
    int i, osz;
    Hunk *h, *prevh;
    Fbuf f;

    n = nil;
    o = nil;
    osz = 0;
    curfile = nil;
    h = nil;
    prevh = nil;
    nchanged = 0;
    changed = nil;
    for(i = 0; i < p->nhunk; i++){
```

```
        h = &p->hunk[i];
        if(strcmp(h->newpath, "/dev/null") == 0)
            nextfile = h->oldpath;
        else
            nextfile = h->newpath;
        if(curfile == nil || strcmp(curfile, nextfile) != 0){
            if(curfile != nil){
                if(!dryrun)
                    o = append(o, &osz, n, f.buf + f.len);
                blat(prevh->oldpath, prevh->newpath, o, osz, f.mode);
                osz = 0;
            }
            if(!dryrun){
                if(slurp(&f, h->oldpath) == -1){
                    rejected(h, fname);
                    goto Next;
                }
                n = f.buf;
            }
            curfile = nextfile;
        }
        if(!dryrun){
            char *e;
            s = n;
            e = search(&f, h);
            if(e == nil){
                if(rejfd == -1)
                    fail("%s:%d: unable to find hunk offset near %s:%d", fname, h->lnum, h->oldpath, h->oldln);
                else{
                    rejected(h, fname);
                    goto Next;
                }
            }
            o = append(o, &osz, s, e);
            o = append(o, &osz, h->new, h->new + h->newlen);
            n = e + h->oldlen;
        }
Next:
        prevh = h;
    }
    if(curfile != nil){
        if(!dryrun)
            o = append(o, &osz, n, f.buf + f.len);
        blat(h->oldpath, h->newpath, o, osz, f.mode);
    }
    free(o);
    return 0;
}
```

⟨*function* `freepatch`*(patch.c)* 275⟩≡ (277)

```
void
freepatch(Patch *p)
{
    Hunk *h;
    int i;

    for(i = 0; i < p->nhunk; i++){
        h = &p->hunk[i];
        free(h->oldpath);
        free(h->newpath);
```

```
        free(h->old);
        free(h->new);
    }
    free(p->hunk);
    free(p->name);
    free(p);
}
```

⟨*function* usage*(patch.c)* 276a⟩≡ (277)

```
void
usage(void)
{
    fprint(2, "usage: %s [-nR] [-p nstrip] [-r rejfile] [patch...]\n", argv0);
    exits("usage");
}
```

⟨*function* main*(patch.c)* 276b⟩≡ (277)

```
void
main(int argc, char **argv)
{
    Biobuf *f;
    Patch *p;
    int i, ok;

    ARGBEGIN{
    case 'd':
        workdir = EARGF(usage());
        break;
    case 'r':
        rejfile = EARGF(usage());
        break;
    case 'p':
        strip = atoi(EARGF(usage()));
        break;
    case 'n':
        dryrun++;
        break;
    case 'R':
        reverse++;
        break;
    default:
        usage();
        break;
    }ARGEND;

    ok = 1;
    if(rejfile != nil){
        rejfd = create(rejfile, OWRITE, 0644);
        if(rejfd == -1)
            fail("open %s: %r", rejfile);
    }
    if(argc == 0){
        if((f = Bfdopen(0, OREAD)) == nil)
            fail("open stdin: %r");
        if((p = parse(f, "stdin")) == nil)
            fail("parse patch: %r");
        if(workdir != nil && chdir(workdir) == -1)
            fail("chdir %s: %r", workdir);
        if(apply(p, "stdin") == -1){
            fprint(2, "apply stdin: %r\n");
```

```
            ok = 0;
        }
        freepatch(p);
        Bterm(f);
        ok = finish(ok);
    }else{
        for(i = 0; ok && i < argc; i++){
            if((f = Bopen(argv[i], OREAD)) == nil)
                fail("open %s: %r", argv[i]);
            if((p = parse(f, argv[i])) == nil)
                fail("parse patch: %r");
            if(workdir != nil && chdir(workdir) == -1)
                fail("chdir %s: %r", workdir);
            if(apply(p, argv[i]) == -1){
                fprint(2, "apply %s: %r\n", argv[i]);
                ok = 0;
            }
            freepatch(p);
            Bterm(f);
            ok = finish(ok);
        }
    }
    exits(ok ? nil : "failed");
}
```

⟨misc/patch.c 277⟩≡

```
#include <u.h>
#include <libc.h>
#include <ctype.h>
#include <bio.h>

typedef struct Patch Patch;
typedef struct Hunk Hunk;
typedef struct Fbuf Fbuf;
typedef struct Fchg Fchg;

struct Patch {
    char    *name;
    Hunk    *hunk;
    usize   nhunk;
};

struct Hunk {
    int lnum;

    char    *orig;
    int origlen;
    int origsz;

    char    *oldpath;
    int oldln;
    int oldcnt;
    int oldlen;
    int oldsz;
    char    *old;

    char    *newpath;
    int newln;
    int newcnt;
    int newlen;
```

```
    int newsz;
    char    *new;
};

struct Fbuf {
    int *lines;
    int nlines;
    int lastln;
    int lastfuzz;
    char    *buf;
    int len;
    int mode;
};

struct Fchg {
    char    *tmp;
    char    *old;
    char    *new;
};

int finish(int);
void    fail(char*, ...);

int strip;
int reverse;
int rejfd = -1;
char    *rejfile;
char    *workdir;
Fchg    *changed;
int nchanged;
int dryrun;
```

⟨*function* `readline` *(patch.c)* 264a⟩

⟨*function* `emalloc` *(patch.c)* 264b⟩

⟨*function* `erealloc` *(patch.c)* 264c⟩

⟨*function* `fail` *(patch.c)* 264d⟩

⟨*function* `fileheader` *(patch.c)* 265a⟩

⟨*function* `hunkheader` *(patch.c)* 265b⟩

⟨*function* `addorig` *(patch.c)* 266a⟩

⟨*function* `addnew` *(patch.c)* 266b⟩

⟨*function* `addold` *(patch.c)* 267a⟩

⟨*function* `addmiss` *(patch.c)* 267b⟩

⟨*function* `addhunk` *(patch.c)* 267c⟩

⟨*function* `hunkcmp` *(patch.c)* 267d⟩

⟨*function* `swapint` *(patch.c)* 268a⟩

⟨*function* `swapstr` *(patch.c)* 268b⟩

⟨*function* `trimhunk`*(patch.c)* 268c⟩

⟨*function* `parse`*(patch.c)* 268d⟩

⟨*function* `rename`*(patch.c)* 270a⟩

⟨*function* `mkpath`*(patch.c)* 270b⟩

⟨*function* `blat`*(patch.c)* 271a⟩

⟨*function* `finish`*(patch.c)* 271b⟩

⟨*function* `slurp`*(patch.c)* 272⟩

⟨*function* `searchln`*(patch.c)* 273a⟩

⟨*function* `search`*(patch.c)* 273b⟩

⟨*function* `rejected`*(patch.c)* 274a⟩

⟨*function* `append`*(patch.c)* 274b⟩

⟨*function* `apply`*(patch.c)* 274c⟩

⟨*function* `freepatch`*(patch.c)* 275⟩

⟨*function* `usage`*(patch.c)* 276a⟩

⟨*function* `main`*(patch.c)* 276b⟩

Glossary

VCS = Version Control System
DVCS = Distributed Version Control System
SHA1 = Secure Hash Algorithm 1
URL = Uniform Resource Locator

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Bibliography

[CS14] Scott Chacon and Ben Straub. *Pro Git*. 2014. Available at https://git-scm.com/book/en/v2. cited page(s) 11, 12

[EJ01] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1), September 2001. Available at http://www.ietf.org/rfc/rfc3174.txt. cited page(s) 16

[Gru86] Dick Grune. Concurrent versions system, a method for independent cooperation. Technical report, Vrije Universiteit, Amsterdam, 1986. Unpublished but available at https://dickgrune.com/Books/Publications/Concurrent_Versions_System,_a_method_for_independent_cooperation.ps. cited page(s) 10

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 12

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 12

[O'S09] Brian O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly, 2009. Available at http://hgbook.red-bean.com/. cited page(s) 11

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 12

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 16, 21

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 10, 12

[Pad25] Yoann Padioleau. *Principia Softwarica: The Plan 9 Utilities*. 2025. cited page(s) 12

[Roc75] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, pages 364–370, December 1975. Available at http://www.basepath.com/aup/talks/SCCS-Slideshow.pdf. cited page(s) 10

[sha93] Secure hash standard, April 1993. Latest version available at https://csrc.nist.gov/publications/detail/fips/180/4/final. cited page(s) 16

[Tic85] Walter F. Tichy. Rcs - a system for version control. *Software Practice and Experience*, pages 637–654, July 1985. Available at http://www.gnu.org/s/rcs/tichy-paper.pdf. cited page(s) 10