Game of Trees

Stefan Sperling <stsp@openbsd.org>

EuroBSDcon 2019

What is Game of Trees?

Game of Trees is a work-in-progress version control system which attempts to be appealing to OpenBSD developers.

- designed and written from scratch
- OpenBSD-style C code base
- ISC licence
- compatible with Git repositories
- concise and useful manual pages

Contributors

Advice, suggestions, documentation, and code:

- Sebastian Benoit
- Anthony Bentley
- Landry Breuil
- Theo Buehler
- Philip Guenther
- Sebastian Marie
- Klemens Nanni
- Carlos Martín Nieto
- Martin Pieuchot
- Hiltjo Posthuma
- Theo de Raadt
- Gonzalo L. Rodriguez
- Ingo Schwarze
- Stefan Sperling
- Joshua Stein
- Patrick Steinhardt
- Uwe Stuehler

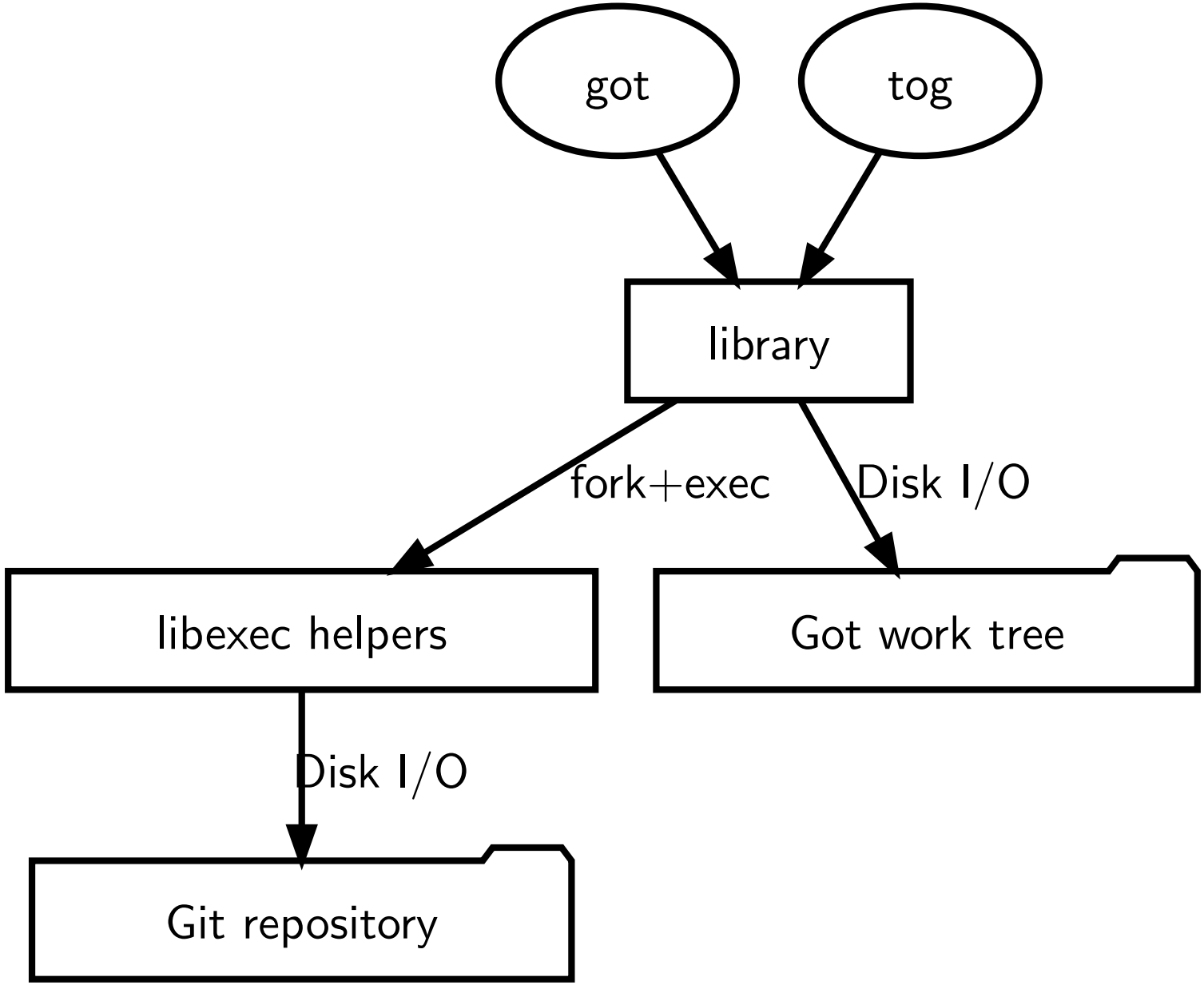
Game of Trees Design

- frontends:
  - got (command line interface)
  - tog (ncurses-based interactive repository browser)
  - frontends call `pledge(2)` and `unveil(2)`
  - frontends use "public" library APIs only

  library:
  - objects, commit graph, work tree, caching, diffing, merging
  - spawns libexec helpers (fork+exec) for repository read access
  - API is unstable; not yet intended for general consumption

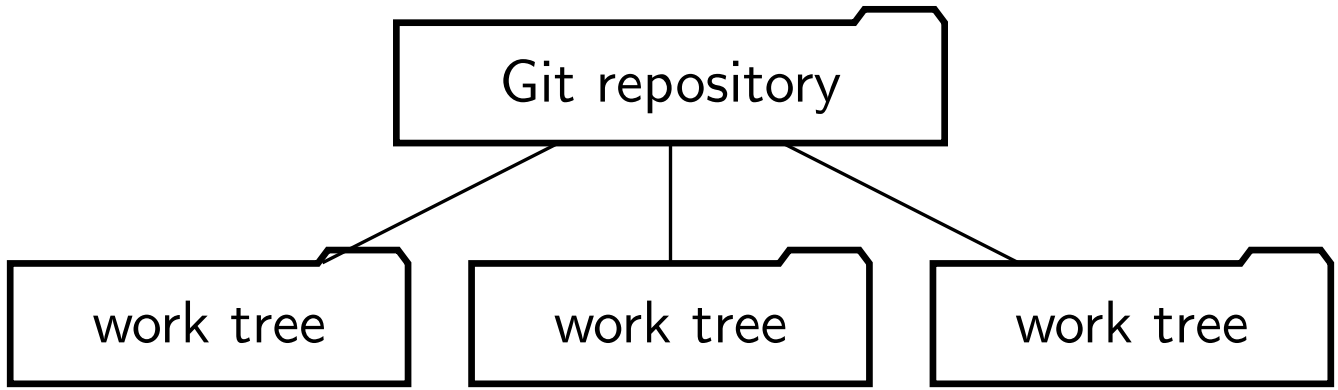
  libexec helpers:
  - standalone event-driven programs
  - decompression and parsing of repository data
  - directly linked to a small subset of library code
  - communication with main library over imsg
  - access to system calls severely limited

Game of Trees Design



Work Trees

A work tree contains a copy of versioned files for editing.
Users may create an arbitrary number of work trees.



Each work tree remembers:

- the path to its Git repository
- its current branch reference
- the commit(s) which file contents were fetched from

Optionally, work tree contents can be limited to a subtree of the full tree stored in the repository.
Git's repository-internal work tree is ignored.

pledge(2) promises

- libexec helpers: `stdio recvfd`
- `got(1)`[1]: `stdio rpath wpath cpath fattr flock proc exec sendfd unveil`
- `tog(1)`: `stdio rpath wpath cpath flock proc tty exec sendfd unveil`

[1] exact list varies by command

unveil(2) exposed paths

- repository: `"r"` or `"rwc"`
- work tree: `"rwc"`
- `/tmp`: `"rwc"`
- libexec helpers: `"x"`
- unversioned files for import: `"r"`

Exceptions:

- `$EDITOR`
  - commit message gets written before unveil(2) is applied
- `$HOME/.gitconfig`
  - opened before unveil(2)
  - parsed by libexec helper

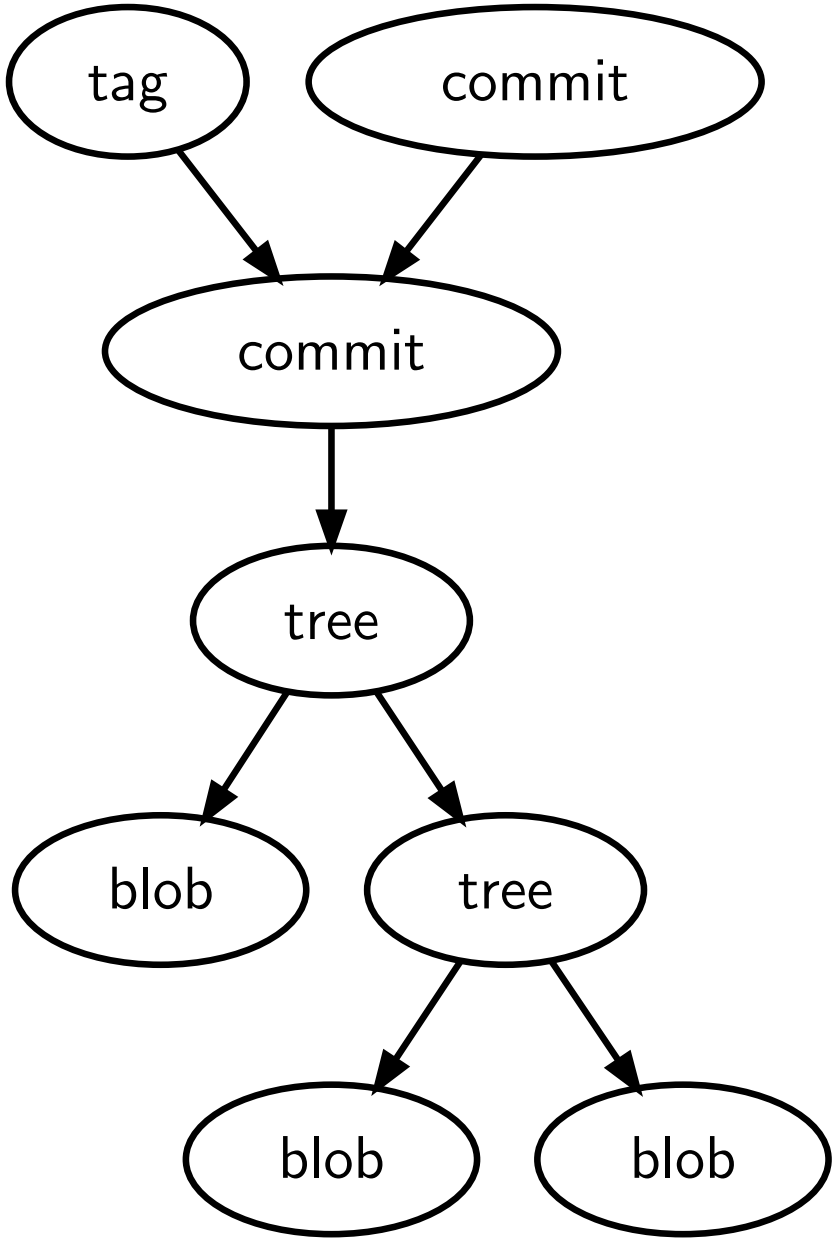
libexec helpers

- `got-read-object` – read header (type, size) from object file
- `got-read-blob` – read blob object file
- `got-read-tree` – read tree object file
- `got-read-commit` – read commit object file
- `got-read-tag` – read tag object file
- `got-read-pack` – extract any type of object from pack file
- `got-read-gitconfig` – read Git configuration file

Git recap coming up

If you are familiar with Git some information will be old news to you. We will keep it short. Lean back and relax :-)

Git repository object types



On-disk object format

Blob, tree, commit, and tag objects:

"object type"	NUL	"object size"	NUL	object data

Type and size are ASCII-encoded strings.
SHA1 hash of all the above constitutes the object's ID.
After hashing, data is compressed with zlib when written to disk.
Each object is stored in a separate file, with a file name based on the ID.

Object data

Blob data:

- file content

Tree data: list of entries:

- entry object ID
- entry `stat(2)` mode
- entry name

Commit data:

- tree object ID
- list of parent commit object IDs
- author + date
- committer + date
- log message

Tag data is similar to commit data.

Pack files (1/2)

Delta-compressed collections of objects.
Pack files contain blobs, trees, commits, and tags, and also:

1. Offset Delta Objects
    - delta base: object at given pack file offset
2. Reference Delta Objects
    - delta base: object with given ID in same pack file

Pack index is stored in a separate file

- list of object IDs and object data offsets
- index entries are sorted by object ID
- ID lookup uses binary search

Pack files (2/2)

Use cases:

- local storage of large object collections
- transmission of a set of objects over the network

At scale, access to objects in pack files is generally faster than access to loose objects in the file system. But pack files come at a cost: Creation is expensive!

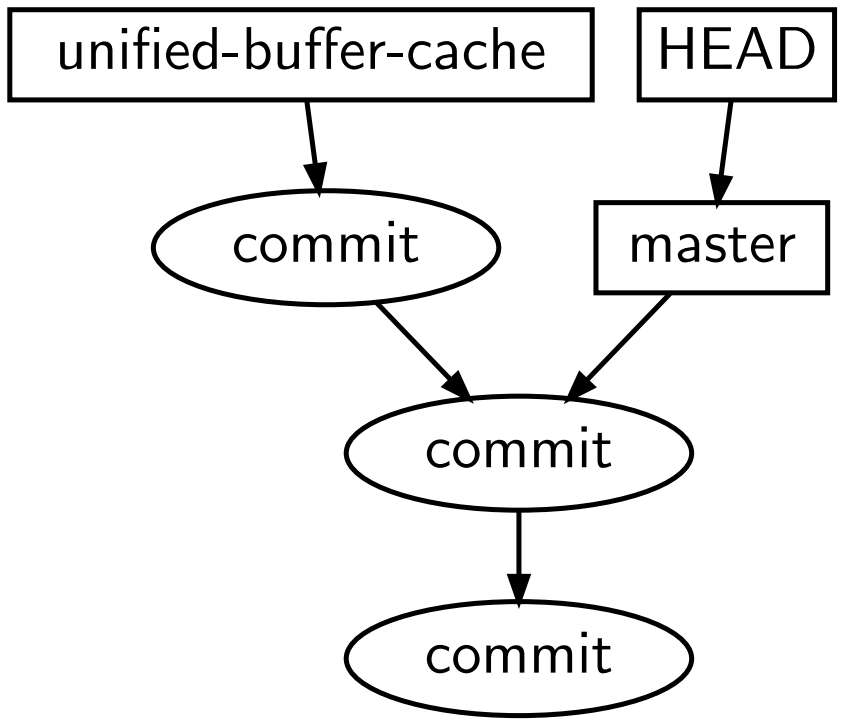
OpenBSD src.git fully packed[2]:

```
49.8M pack-0c3467692f110178cf674ede60894b091c7f8f95.idx
1022M pack-0c3467692f110178cf674ede60894b091c7f8f95.pack
```

[2] Your mileage may vary since packing involves heuristics

References

References provide human-readable keys for object lookup.
References may be symbolic, i.e. point at other references.
Their most prominent use case is looking up branch head commits:



Reference namespaces

References are organized in a namespace hierarchy:

- `refs/heads/...` – local branch heads
- `refs/tags/...` – for tag object lookup
- `refs/remotes/...` – copies of other repositories' histories
- `refs/got/...` – for internal use by Game of Trees

Namespace prefix may be abbreviated or omitted.
Given `unified-buffer-cache`, try to look up:

1. `refs/heads/unified-buffer-cache`
2. `refs/tags/unified-buffer-cache`
3. `refs/remotes/*/unified-buffer-cache`

Git recap done

Any questions about the Git repository format from Henning before we continue?

Game of Trees Command Line Interface

Set of commands was designed from scratch, borrowing user-facing terminology from CVS, SVN, Mercurial, and Git.

- Capture OpenBSD developer workflows
- Prioritize ease of use and simplicity over flexibility
- Offer strictly required features only
  - new features added if requested by OpenBSD developers
- Minimize amount of command line flags and typing
  - parse options with `getopt(3)`, not `getopt_long(3)`
  - allow use of references, tags, and object IDs interchangably
  - accept abbreviated SHA1 object IDs
- No colours!

Current Game of Trees Command Set (1/2)

- `init` – create repositories
- `import` – create commits from unversioned files
- `checkout` – create work trees containing versioned files
- `update` – change work tree's base commit
- `log` – view commit history
- `diff` – view local changes or differences between objects
- `blame` – view line-by-line history of files
- `tree` – list versioned files and folders in repository
- `status` – check work tree for uncommitted local changes
- `ref` – manage references
- `branch` – manage branches
- `tag` – manage tags

Current Game of Trees Command Set (2/2)

- `add` – add unversioned files to version control
- `remove` – remove versioned files
- `revert` – discard uncommitted local changes
- `commit` – create new commit objects
- `cherrypick` – merge change from another branch
- `backout` – undo an already committed change
- `rebase` – merge local branches with incoming changes
- `histedit` – edit commit history of local branches
- `stage` – stage a subset of changes for next commit
- `unstage` – undo staging of changes
- `cat` – show content of arbitrary objects

Current Game of Trees Command Set

- `init`, `import`, `checkout`, `update`
- `log`, `diff`, `blame`, `status`
- `ref`, `branch`, `tag`
- `add`, `remove`, `revert`, `commit`
- `cherrypick`, `backout`, `rebase`, `histedit`
- `stage`, `unstage`
- `cat`, `tree`

Example: Starting from files (1/2)

```
|-- README
`-- src
    |-- Makefile
    `-- prog.c
```

src/prog.c:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        printf("hello world\n");
}
```

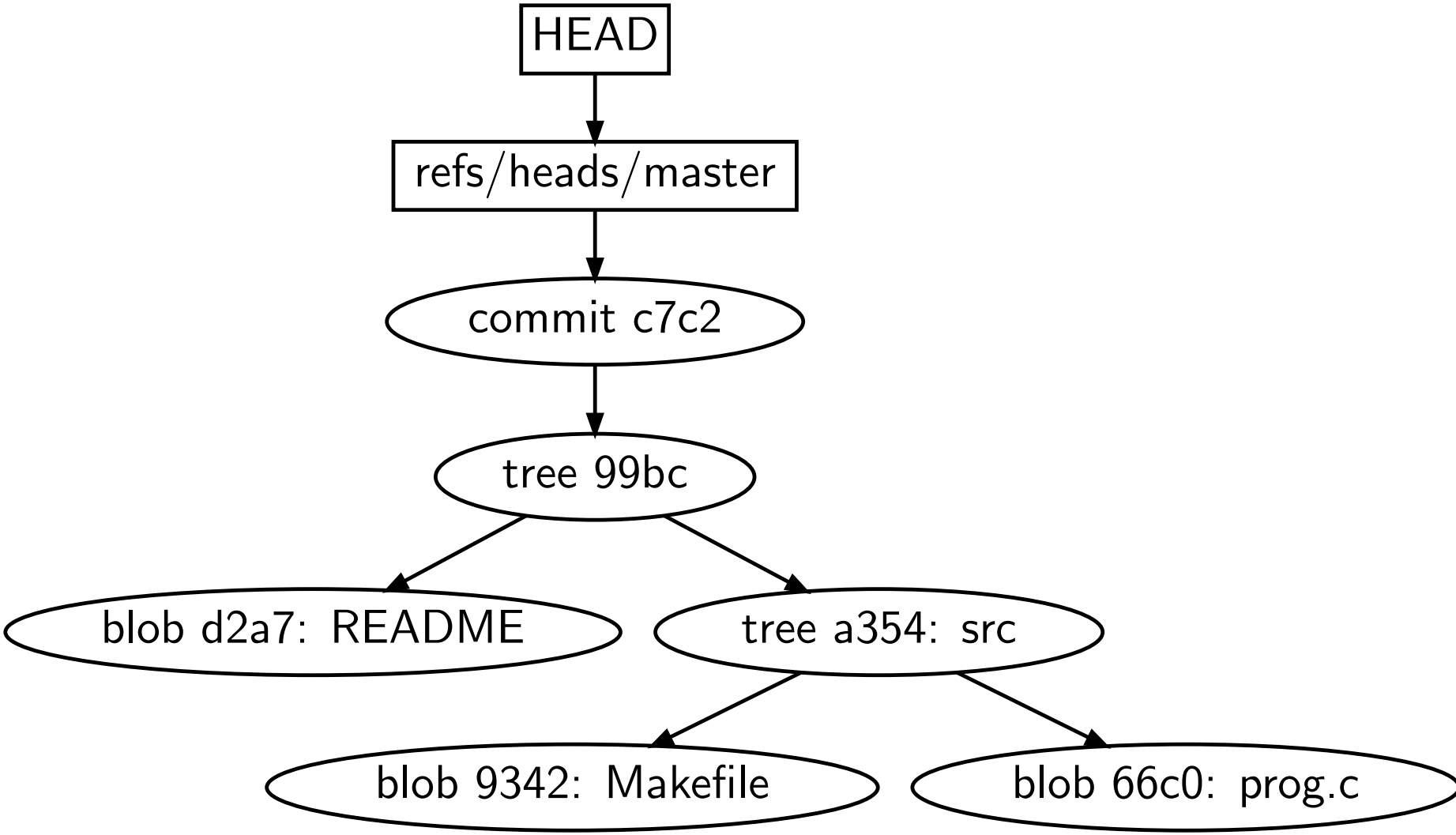
Example: Starting from files (2/2)

Commands: **init**, **import**

```
$ got init /tmp/repo
$ cd /tmp/repo
$ got import -m "import demo project" /tmp/my-files
A  /tmp/my-files/src/Makefile
A  /tmp/my-files/src/prog.c
A  /tmp/my-files/README
Created branch refs/heads/master with \
commit c7c2e404ffb0dd55cc34842431541fdd4f977a28
```

`got import` creates a *root* commit, i.e. the created commit has no parent commits and history begins here. If the `master` branch already exists, a different name must be chosen.

Repository now contains the imported project



Example: Creating a work tree

Command: **checkout**

```
$ got checkout /tmp/repo /tmp/wt
A  /tmp/wt/README
A  /tmp/wt/src/Makefile
A  /tmp/wt/src/prog.c
Now shut up and hack
$
```

Work trees can be placed anywhere in the file system hierarchy.

Example: Checking for uncommitted changes

Command: **status**

```
$ got status
?  src/prog
M  src/prog.c
?  src/prog.d
?  src/prog.o
$
```

For now, unversioned files can be ignored via a `.cvsignore` file, as used in the OpenBSD ports tree.

Example: Viewing uncommitted local changes

Command: **diff**

```
$ got diff
diff c7c2e404ffb0dd55cc34842431541fdd4f977a28 /tmp/wt
blob - 66c06bb066b1b5f7c72359c21ee6dafd54e256e1
file + src/prog.c
--- src/prog.c
+++ src/prog.c
@@ -1,5 +1,5 @@
 #include <stdio.h>
 int main(int argc, char *argv[])
 {
-       printf("hello world\n");
+       printf("I like the flowers\n");
 }
$
```

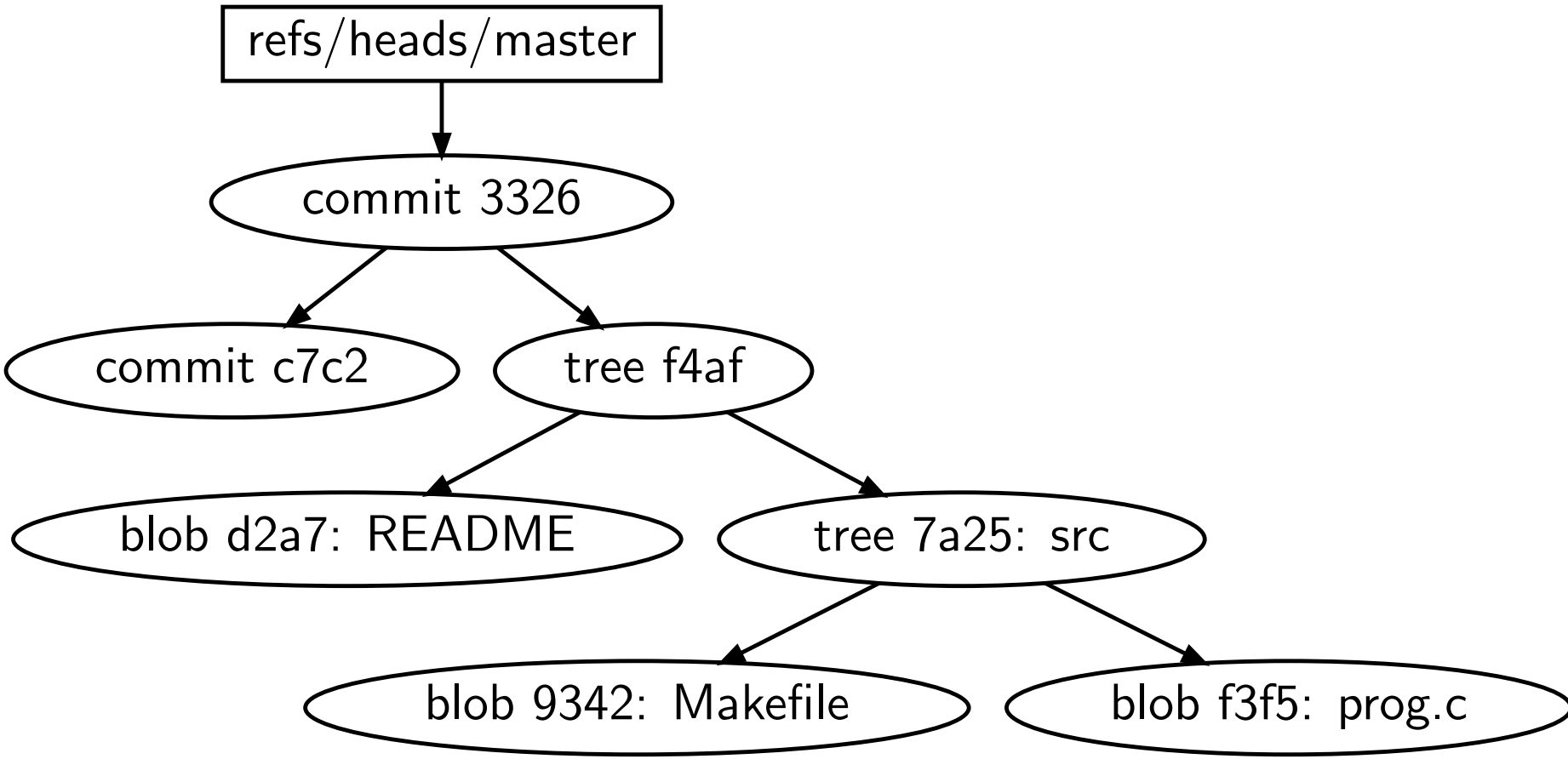
Example: Committing local changes

Command: **commit**

```
$ got commit -m "spread the message"
M  src/prog.c
Created commit 3326e467f63840c7aa10937634412d100d04e6e2
$
```

Most commands have short aliases.
`got ci` is the short alias for `got commit`.

Newly created commit on the `master` branch



Example: Discarding local changes

Command: **revert**

```
$ got status
M  src/prog.c
$ got revert src/prog.c
R  src/prog.c
$ got status
$
```

`got revert` is destructive and cannot be undone automatically!

Example: Discarding local changes selectively (1/4)

Two changes, one of them bad:

```
@@ -1,5 +1,7 @@
 #include <stdio.h>
 int main(int argc, char *argv[])
 {
-       printf("I like the flowers\n");
+       printf("I like the daffodils\n");
 }
+
+/* syntax error
```

Example: Discarding local changes selectively (2/4)

Command: **revert -p** (mnemonic: “patch”)

```
$ got revert -p src/prog.c
-----------------------------------------------
@@ -1,5 +1,7 @@
 #include <stdio.h>
 int main(int argc, char *argv[])
 {
-       printf("I like the flowers\n");
+       printf("I like the daffodils\n");
 }

 /* syntax error
-----------------------------------------------
M  src/prog.c (change 1 of 2)
revert this change? [y/n/q] n
```

Example: Discarding local changes selectively (3/4)

```
--------------------------------------------
@@ -3,3 +3,5 @@ int main(int argc, char *argv[])
 {
        printf("I like the daffodils\n");
 }
+
+/* syntax error
--------------------------------------------
M  src/prog.c (change 2 of 2)
revert this change? [y/n/q] y
$
```

Example: Discarding local changes selectively (4/4)

The bad change has been wiped out:

```
@@ -1,5 +1,5 @@
 #include <stdio.h>
 int main(int argc, char *argv[])
 {
-       printf("I like the flowers\n");
+       printf("I like the daffodils\n");
 }
$ got commit -m 'prefer the plant genus narcissus'
```

Real world use case: `got revert -p` can wipe out debug printfs

Backing out commited changes (1/3)

Creating reversed changes requires a work tree which:

- is fully updated to the latest commit on the relevant branch
- may already contain other arbitrary local changes
- will carry the reversed change until `got commit`

Example: Backing out commited changes (2/3)

Command: **backout**

```
$ got backout 349d8
G  src/prog.c
Backed out commit 349d80308aa17d90842afc5d460fade6b6de095c
$
```

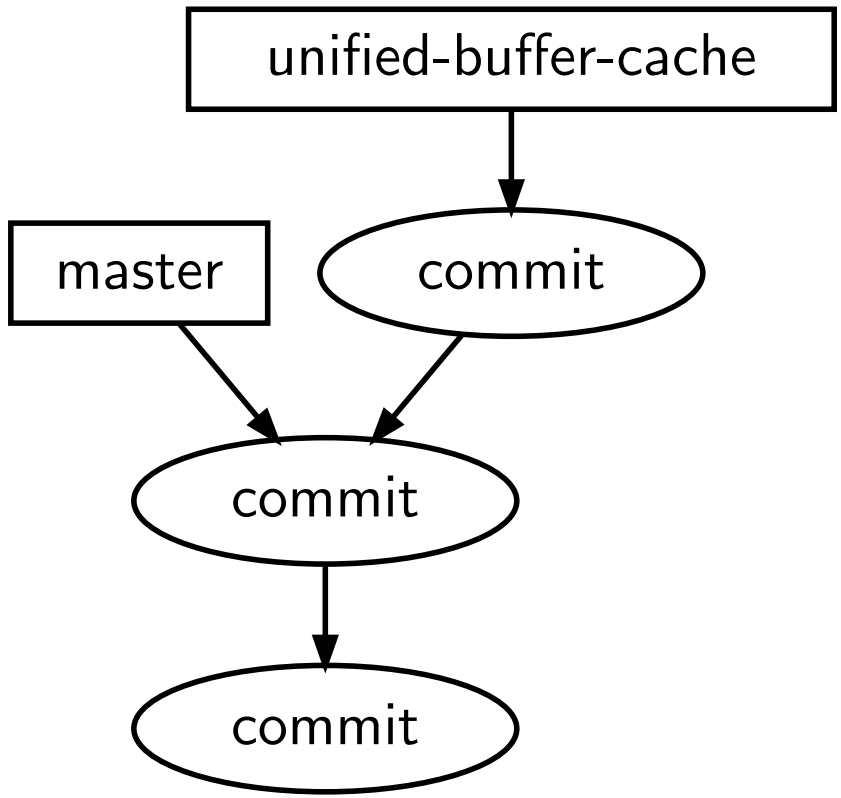
Example: Backing out commited changes (3/3)

Backed-out change has been merged locally for future commit:

```
@@ -1,5 +1,5 @@
 #include <stdio.h>
 int main(int argc, char *argv[])
 {
-       printf("I like the daffodils\n");
+       printf("I like the flowers\n");
 }
```

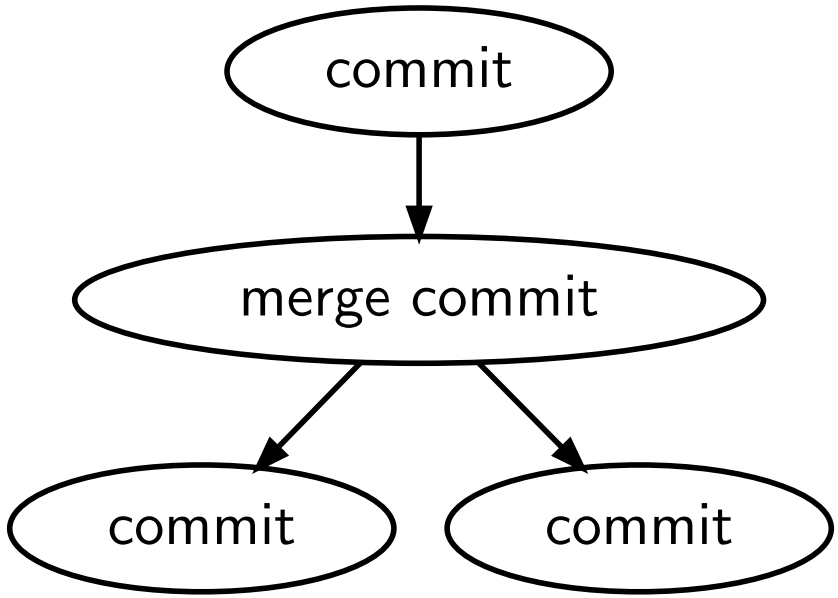
Branches (1/3)

Branches are modeled as references which point at a chain of commit objects:



Branches (2/3)

Game of Trees intentionally steers users towards creating a linear history. There is currently no command[3] which would allow users to create commits with multiple parent commits like this:



[3] `got merge` will have to be added for special cases such as vendor branches

Branches (3/3)

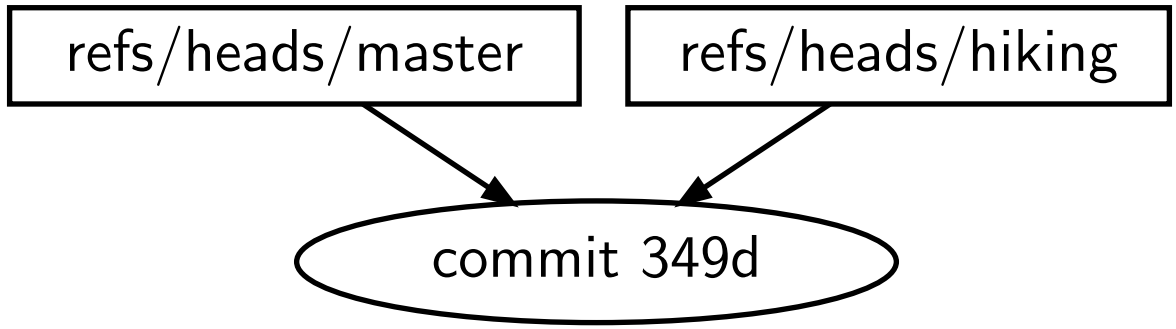
Changes made in other repositories must be copied to the local repository before they become visible.
Branches can temporarily store incoming changes separately from local changes.
Once incoming and local changes are stored, they can be merged.

Example: Creating a branch

Command: **branch**

```
$ got branch -l
* master: 349d80308aa17d90842afc5d460fade6b6de095c
$ got branch hiking
$ got branch -l
  hiking: 349d80308aa17d90842afc5d460fade6b6de095c
* master: 349d80308aa17d90842afc5d460fade6b6de095c
$
```

Repository now contains a new reference: `hiking`



In this example, we will store our local changes in `hiking` and incoming changes in `master`[4]

[4] Storing incoming changes under `refs/remotes/` is supported, too.

Example: Switching between branches

Command: **update -b** (mnemonic: “branch”)

```
$ got update -b hiking
Switching work tree from refs/heads/master to refs/heads/hiking
Already up-to-date
$ got branch -l
* hiking: 349d80308aa17d90842afc5d460fade6b6de095c
  master: 349d80308aa17d90842afc5d460fade6b6de095c
$
```

Example: Creating commits on a branch (1/2)

Commit hiking-related change 1:

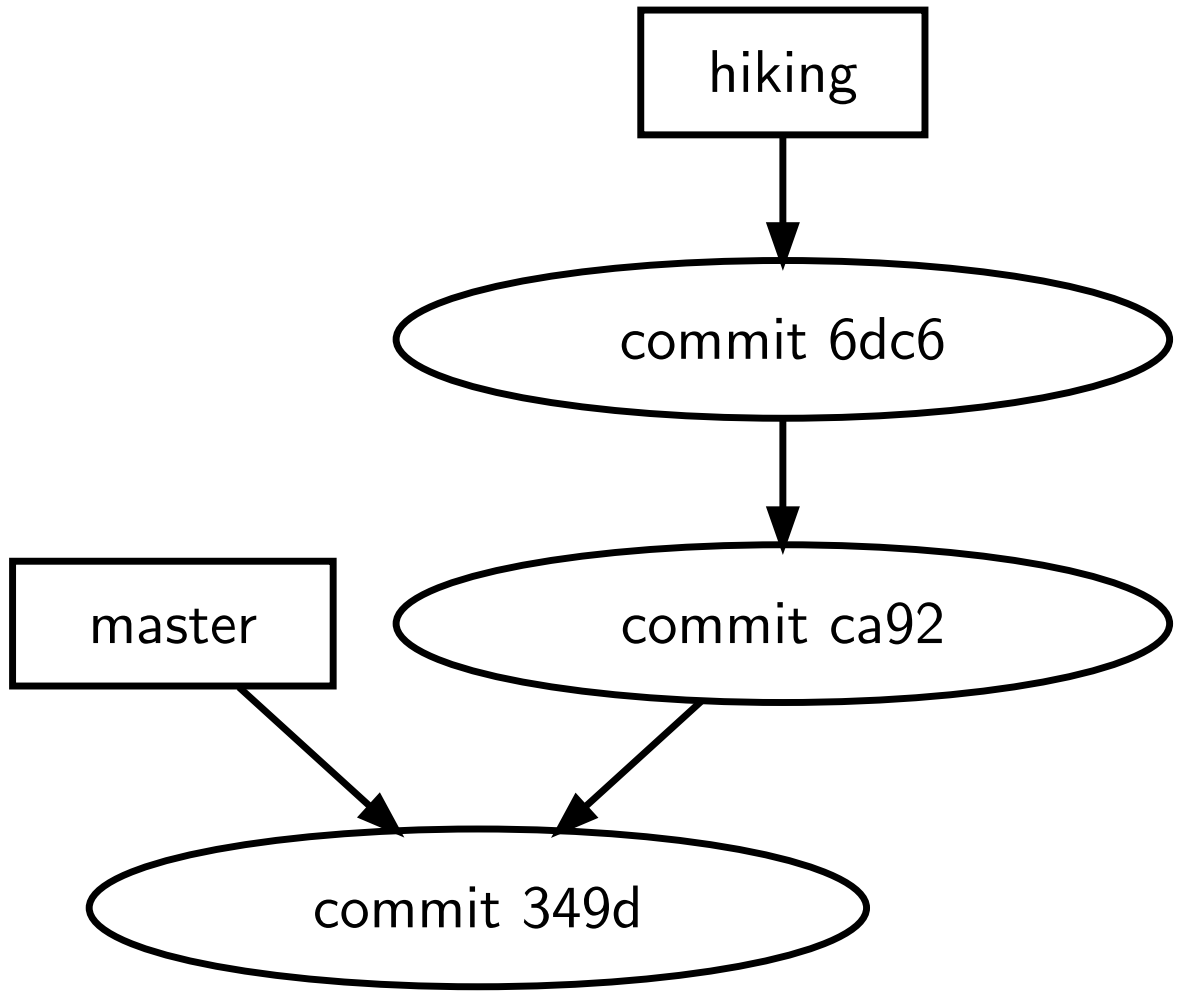
```
@@ -1,5 +1,5 @@
 #include <stdio.h>
 int main(int argc, char *argv[])
 {
-       printf("I like the daffodils\n");
+       printf("I like the mountains\n");
 }
$ got commit -m "this is a hiking club"
```

Example: Creating commits on a branch (2/2)

Commit hiking-related change 2:

```
@@ -2,4 +2,5 @@
 int main(int argc, char *argv[])
 {
         printf("I like the mountains\n");
+        printf("I like the rolling hills\n");
 }
$ got commit -m "with a computer problem"
```

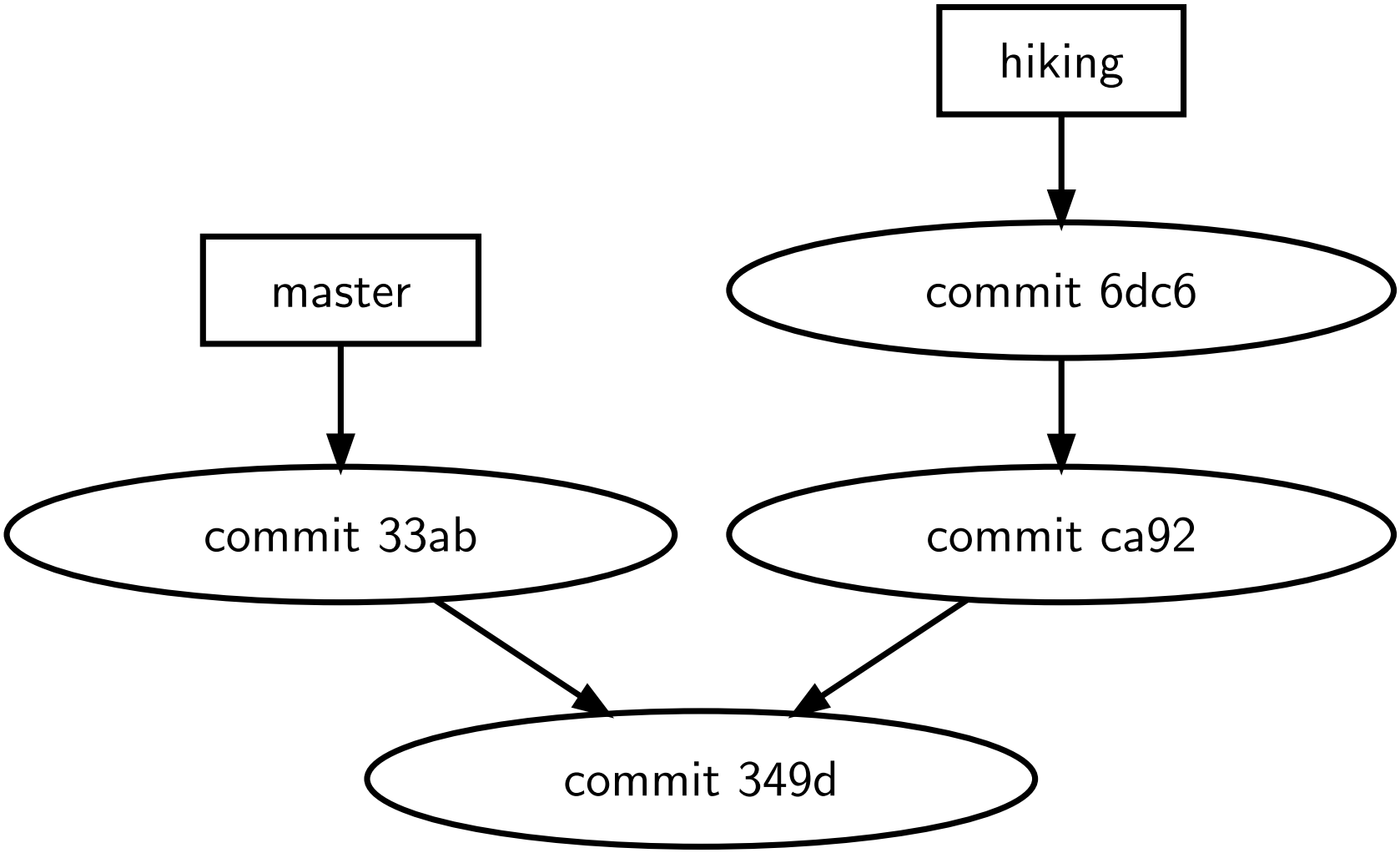
The `hiking` branch now contains new commits



A separate change arrives on the `master` branch

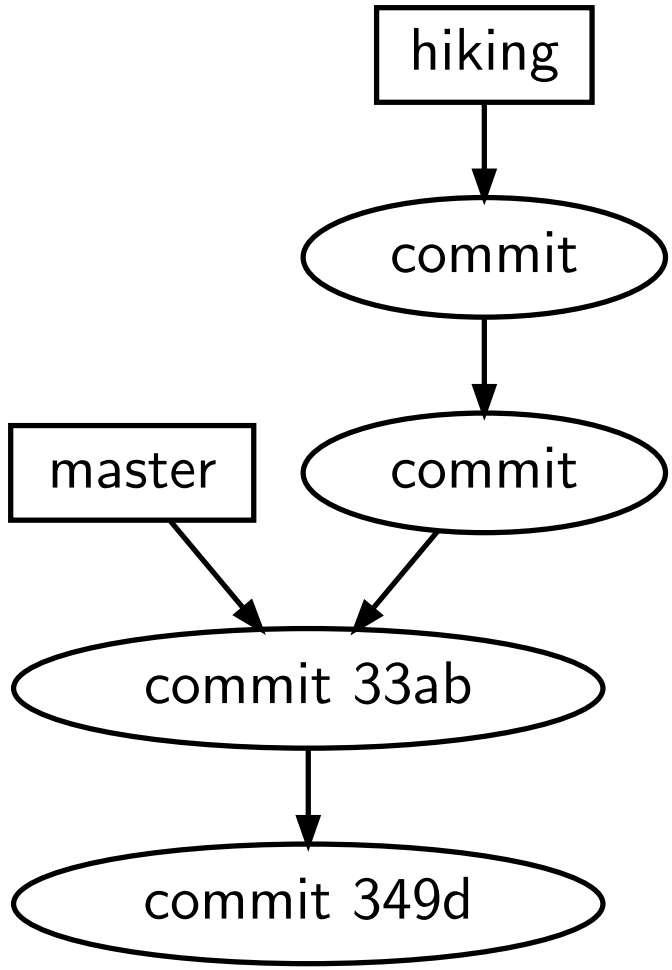
```
@@ -1,5 +1,5 @@
 #include <stdio.h>
 int main(int argc, char *argv[])
 {
-       printf("I like the daffodils\n");
+       printf("I like the fireplace\n");
 }
$ got commit -m "light it up"
```

Repository with two distinct branches



Commits on the `hiking` branch are based on commit 349d.

We want commit history to be linear again, like this:



Commit object IDs on `master` must not change!

Those commits are a permanent part of project history.

Commits on `hiking` may be rebased on top of commit 33ab.

Rebasing branches to linearize commit history

Rebasing `hiking` onto `master` requires a work tree which:

- is fully updated to the latest commit on the `master` branch
- does not contain any local changes
- will be switched to a temporary branch[5] to accumulate rebased commits
- will then be switched to the rebased `hiking` branch

[5] Automatically created in the `refs/got` namespace

Example: rebasing branches

Command: **rebase**

```
$ got up -b master
Switching work tree from refs/heads/hiking to refs/heads/master
U  src/prog.c
Updated to commit 33ab33ecfacdc0a966a6272277a15e8692e80cae
$ got rebase hiking
C  src/prog.c
got: conflicts must be resolved before rebasing can continue
$
```

3-way merge conflicts

```
#include <stdio.h>
int main(int argc, char *argv[])
{
<<<<<<< commit ca929e9ee5e836a6af8d70a94e7abdb326c4e77c
        printf("I like the mountains\n");
=======
        printf("I like the fireplace\n");
>>>>>>> src/prog.c
}
```

Game of Trees checks file content for conflict markers.

Files containing conflict markers may not be committed[6].

[6] This is a feature, not a bug.

Excursion: diff3 algorithm (1/3)

Common misconception:

"Merging works like `diff` + `patch`" **Wrong**!!!

Given file O with this content:

1 2 3 4 5 6

And two files A and B, each derived from O:

A:	7	2	3	4	5	6
B:	1	2	3	4	8	6

(Imagine that each number represents a line of text.)

Excursion: diff3 algorithm (2/3)

Compare O to A and mark regions with differences.
Also compare O to B and mark regions with differences.
Do **not** compare A to B, which regular `diff` would do!

A:	⑦	2	3	4	5	6
O:	①	2	3	4	⑤	6
B:	1	2	3	4	⑧	6

Regions changed by A and B do not overlap so there is no conflict.
Merge result:

7 2 3 4 **8** 6

Excursion: diff3 algorithm (3/3)

Given two different files A and B, each derived from O:

A:	7	2	3	4	8	6
B:	9	0	3	4	8	6

Compare O to A and O to B, and mark changed regions:

A:	⑦	2	3	4	⑧	6
O:	①	②	3	4	⑤	6
B:	⑨	⓪	3	4	⑧	6

A and B do not agree on one of the two overlapping regions.
Merge result is ambiguous. Output contains a merge conflict:

7	**2**				
		3	4	**8**	6
9	**0**				

`http://www.cis.upenn.edu/~bcpierce/papers/diff3-short.pdf`

Example: resolve conflicts and continue rebasing

Command: **rebase -c** (mnemonic: “continue”)

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        printf("I like the mountains\n");
        printf("I like the fireplace\n");
}
```

```
$ got rebase -c
ca929e9ee5e8 -> 3d8aa2ddf185: this is a hiking club
C  src/prog.c
got: conflicts must be resolved before rebasing can continue
$ f#ck this shit!
ksh: f#ck: not found
```

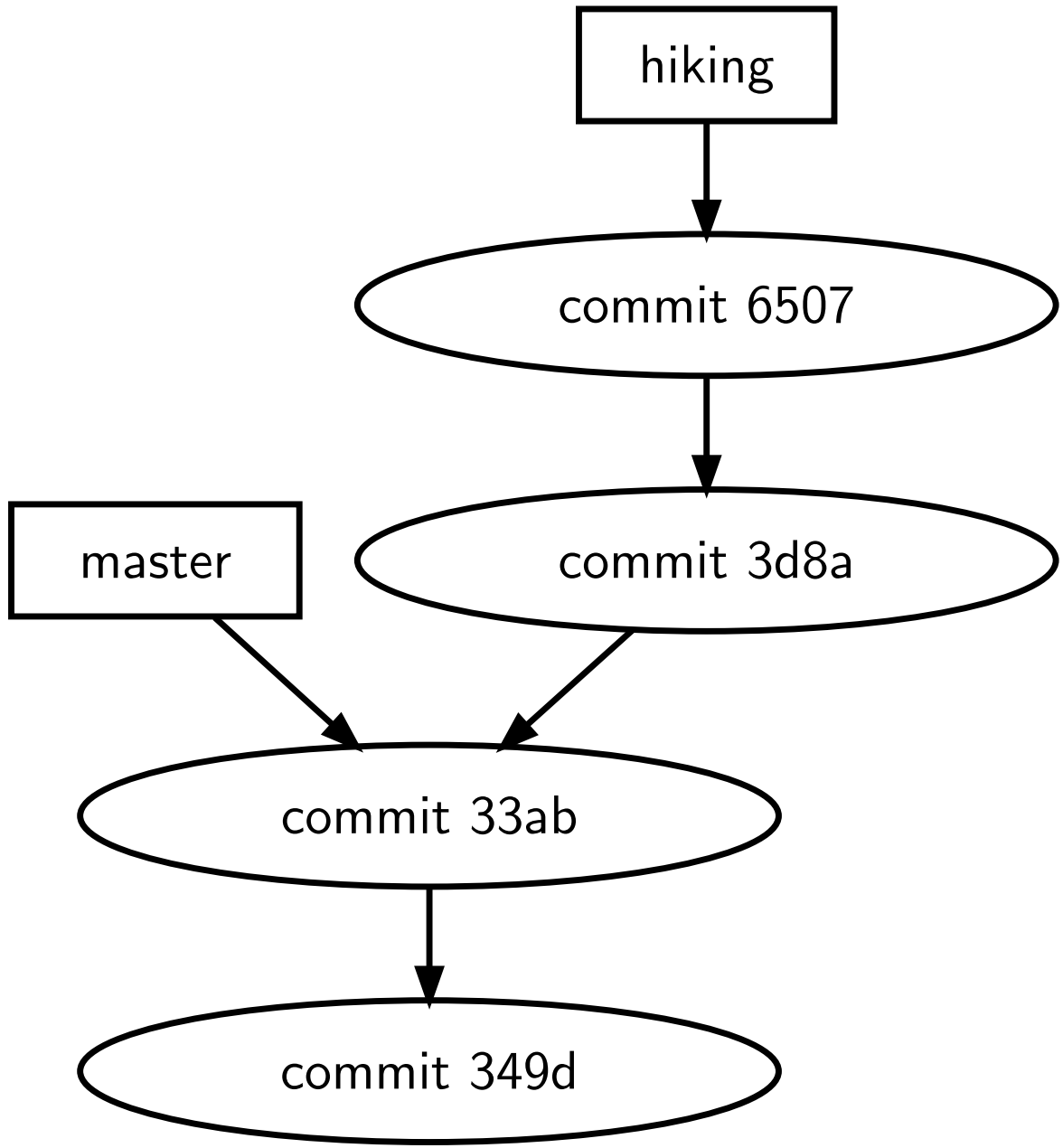
More 3-way merge conflicts

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        printf("I like the mountains\n");
<<<<<<< commit 6dc62c2c8c5b24483b02e8afb21a951d521a71f0
        printf("I like the rolling hills\n");
=======
        printf("I like the fireplace\n");
>>>>>>> src/prog.c
}
```

Example: resolve conflicts and continue rebasing

```
@@ -2,5 +2,7 @@
 int main(int argc, char *argv[])
 {
         printf("I like the mountains\n");
+        printf("I like the rolling hills\n");
         printf("I like the fireplace\n");
+        printf("when the light is low\n");
 }
$ got rebase -c
6dc62c2c8c5b -> 65070d4f62eb: with a computer problem
Switching work tree to refs/heads/hiking
$
```

Rebase: Mission accomplished!



Example: View rebased commit history

Command: **log -p** (mnemonic: “patch”)

```
$ got log -p
-----------------------------------------------
commit 65070d4f62eb1c91aff54d06fd955ed1067f1173 (hiking)
from: Flan Hacker <flan_hacker@openbsd.org>
date: Wed Sep 18 11:37:42 2038 UTC

 with a computer problem

diff 0e508849a2308e87394772661b2fdffacf3fb7a3 167a1902994851229d99833d4ca853206
blob - 4cd87ed36bc46c0eb9740af0e41e5d0c3bf889e1
blob + 954c20c02c8814d34309fbe1c475a24261ce916c
--- src/prog.c
+++ src/prog.c
@@ -2,5 +2,7 @@
 int main(int argc, char *argv[])
 {
         printf("I like the mountains\n");
+       printf("I like the rolling hills\n");
         printf("I like the fireplace\n");
+       printf("when the light is low\n");
 }
```

Example: View rebased commit history

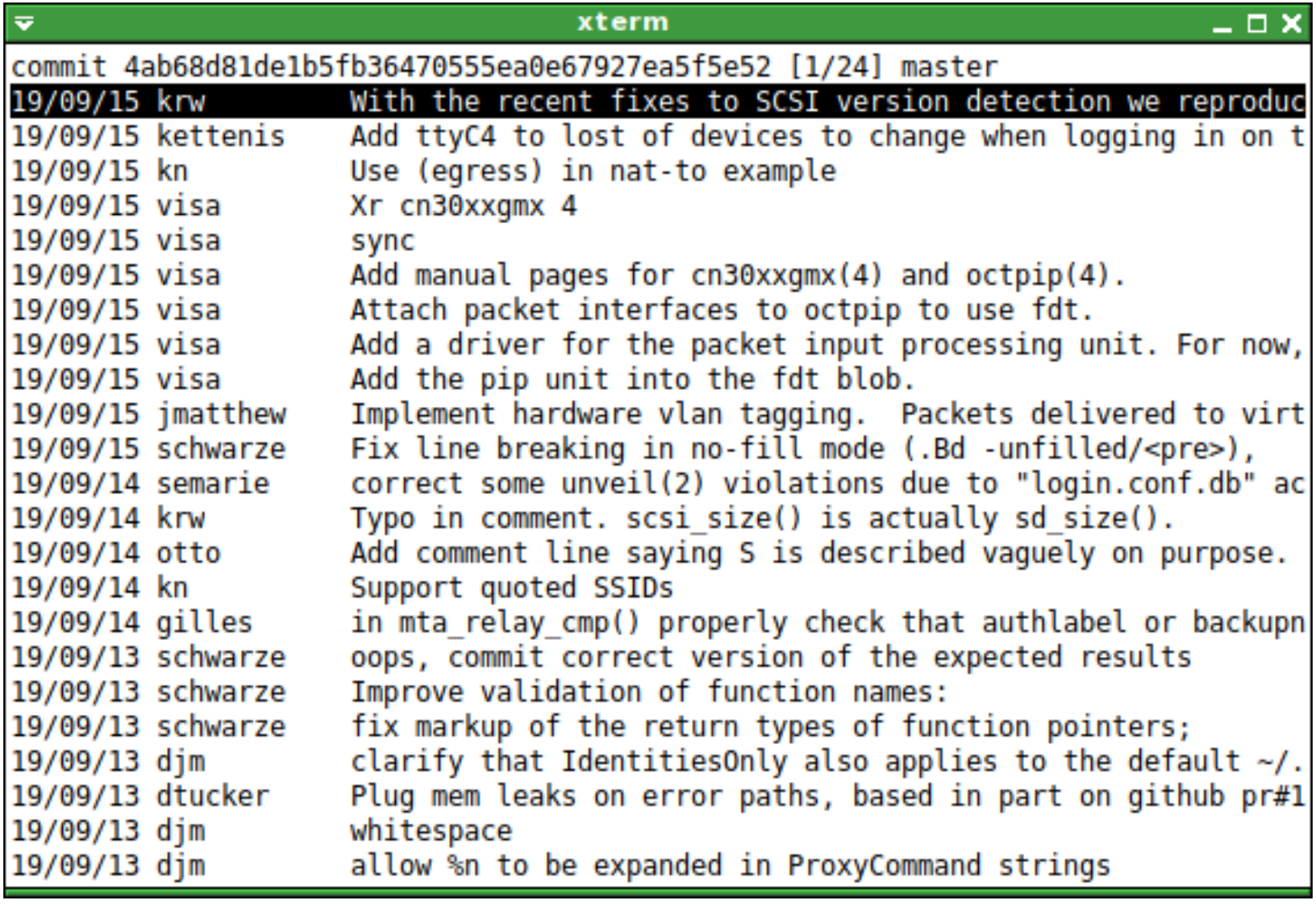
```
---------------------------------------------
commit 3d8aa2ddf1853fd829a1a816eb14fbfff488468a
from: Flan Hacker <flan_hacker@openbsd.org>
date: Wed Sep 18 11:31:05 2038 UTC

 this is a hiking club

diff 54f052b6a309c5aa9d766d465aee7063f5caceff 0e508849a2308e87394772661b2fdffac
blob - 04005c8685e43cc4e3fae7a925df2c084bbd0c46
blob + 4cd87ed36bc46c0eb9740af0e41e5d0c3bf889e1
--- src/prog.c
+++ src/prog.c
@@ -1,5 +1,6 @@
 #include <stdio.h>
 int main(int argc, char *argv[])
 {
+       printf("I like the mountains\n");
        printf("I like the fireplace\n");
 }

---------------------------------------------
commit 33ab33ecfacdc0a966a6272277a15e8692e80cae (master)
from: Flan Hacker <flan_hacker@openbsd.org>
date: Wed Sep 18 08:33:07 2038 UTC
```

Features added for Git users

In Game of Trees, these features are supported but optional:

- `got stage` – stage a subset of changes for next commit
  - `got commit` will commit staged changes only if present
  - `got stage -p`: stage changes interactively
- `got histedit` – edit commit history of local branches
  - edit, reorder, drop commits
  - merge multiple commits into one

tog(1) – read-only repository browser



Supports `log`, `diff`, `blame`, and `tree` views

Project Timeline (2017, 2018)

- September 2017: Git hallway track at EuroBSDcon in Paris
- November 2017: p2k17 backroom Git meeting, read references (**p2k17**), read loose objects (**s2k17**)
- December 2017: diff objects
- February 2018: read packfiles
- March 2018: `got(1)` CLI, check out files, `pledge(2)`
- April 2018: `tog(1)` TUI, `fork(2)`, imsg (**p2k18**)
- June 2018: commit graph, pthreads in `tog(1)` (**g2k18**)
- Sep 2018: fork+exec (**n2k18**)
- December 2018: update files to different commits

Project Timeline (2019)

- January 2019: apply `unveil(2)`
- February 2019: 3-way merge, `got(1)` used for OpenBSD dev
- March 2019: add/remove files, per-path updates (**t2k19**)
- May 2019: create commits (**g2k19**)
- June 2019: merge changes between branches
- July 2019: rebase branches, edit branch history
- August 2019: commit staging, got-0.1 release
- September 2019: got-0.15 release

Planned features

`gotadmin(8)` – repository administration

- create pack files (**important next step**)
- consistency check
- garbage collection of unreferenced objects
- import Git “fast-import” data streams
- also export such data streams
  - we may want a plaintext backup format – is this suitable?

Planned features

`gotd(8)` – privsep server deamon

- manage queue of incoming commits
- sanity-check incoming commits
  - don't bother with hook scripts; all checks are built-in
- server-side rebasing of commits to enforce linear history
- network protocol libexec helpers
  - SSH protocol speaker (read/write)
  - HTTPS protocol speaker (read-only)
  - speaker compatible with Git (common denominator)
- mirror repositories from/to remote servers

Planned features

Pull changes from remote repositories

- fetch remote commits to `refs/remote` namespace
- attempt to rebase the work tree's current branch
- manual rebase required in case of merge conflicts

Push changes to remote repositories served by `gotd(8)`
`https://gameoftrees.org/notes-pull-push`

Planned features

Provide “push by default” commit behaviour

- commit to central server and local repo at the same time
- can be supported for uncommitted/staged changes only
- simplified workflow
    - just checkout, commit, pull/update, commit, ...

Planned features

`gotweb(8)` – repository viewer for browsers

- implement as frontend alongside `got(1)` and `tog(1)`
- provide read-only browsing of repositories
- integrate with existing tooling such as:
  - `httpd(8)`
  - `slowcgi(8)`
  - Kristap's `kcgi(8)`

Thank you for listening! Got questions?

https://gameoftrees.org