Fossil Version Control

A User Guide

Jim Schimpf, Chris X., Torsten Berg and others

16 June 2022

Foreword

This book was started in 2010 using a Dual processor Mac 550 MHz PowerPC running Mac OSX 10.5. Current revisions are now done on MacBook Pro (Intel) running MacOS 10.12.1 at 2.5GHz. It was originally done using Fossil version 1.27 and now revisions to the book are using Fossil 1.36. Because of this some of the pictures here will look slightly different in your copy of Fossil but the maintainers have been careful to update the esthetics but not the functions of the various pages. So you will see different, better looking pages but the functions will be the same.

In November 2020, Fossil is up to version 2.14 and there have been extensive changes to the command set, so documentation is longer but follows the same pattern as in earlier versions. Edits now being done on Debian Linux x64 running on an Intel J1900 cpu at 2GHz with 4 processors and 4 GB RAM. Could be done on a Raspberry Pi or Chromebook.

Contents

Chapter 1

Why source control

1.1 What is source control?

A source control system is software that manages the files in a project. A project (typically a software application, but also this book) usually has a number of files. These files in turn are managed by organizing them in directories and sub-directories. At any particular time this set of files in their present edited state make up a working copy of the project at the latest version. If other people are working on the same project you would give them your current working copy to start with. As they find and fix problems, their working copy will become different from the one that you have (it will be in a different state or version). For you to be able to continue where they left off, you will need some mechanism to update your working copy of the files to the latest version available in the team. As soon as you have updated your files, this new version of the project goes through the same cycle again. Most likely the current version will be identified with a code, this is why software has versions and books have editions.

Software developers on large projects with multiple developers could see this cycle and realized they needed a tool to control the changes. With multiple developers sometimes the same file would be edited by two different people changing it in different ways and records of what got changed would be lost. It was hard to bring out a new release of the software and be sure that all the work of all team members to fix bugs and write enhancements were included.

A tool called **Source Code Control System** (SCCS) was developed at Bell Labs in 1972 to track changes in files. It would remember each of the changes made to a file and store comments about why this was done. It also limited who could edit the file so conflicting edits would not be done.

This was important but developers could see more was needed. They needed

to be able to save the state of all the files in a project and give it a name (i.e., Release 3.14). As software projects mature you will have a released version of the software being used as well as bug reports written against it, while the next release of the software is being developed adding new features. The source control system would have to handle what are now called branches. One branch name for example "Version 1" is released but continues to have fixes added to create Version 1.1, 1.2, etc. At the same time you also have another branch which named "Version 2" with new features added under construction.

In 1986 the open source **Concurrent Version Control System** (CVS) was developed. This system could label groups of files and allow multiple branches (i.e. versions) simultaneously. There have been many other systems developed since then, some open source and some proprietary.

Fossil, which was originally released in 2006, is an easy to install version control system that also includes a ticketing system (see section 5.7), a Wiki (see section 7.4.5) and self hosted web server (see section 7.2.15).

1.2 Why version the files?

Why do you want to use a source control system? You won't be able to create files, delete files, or move files between directories at random. Making changes in your code becomes a check list of steps that must be followed carefully.

With all those hassles, why do it? One of the most horrible feelings as a developer is the "It worked yesterday" syndrome. That is, you had code that worked just fine and now it doesn't. If you work on only one document, it is conceivable that you saved a copy under a different name (perhaps with a date) that you could go back to. But if you did not, you doubtless feel helpless at your inability to get back to working code. With a source control system and careful adherence to procedures you can just go back in time and get yesterday's code, or the code from one hour ago, or from last month. After you have done this, starting from known good code, you can figure out what happened.

Having a source control system also gives you the freedom to experiment, "let's try that radical new technique", and if it doesn't work then it's easy to go back to the previous state.

The rest of this book is a user manual for the Fossil version control system that does code management and much much more. It runs on multiple OS's and is free and open source software (FOSS). It is simple to install as it has only one executable and the repositories it creates are a single file that is easy to back up and are usually only 50 % the size of the original source.

1.2.1 How to get it

If this has interested you then you can get a copy of the Fossil executable here: www.fossil-scm.org/download.html. There are links to Linux, Mac, and Windows executable on this page. The source code is also available if you want or need to compile it yourself. This web site, containing the content you are reading, is self-hosted by Fossil.

1.3 Source control description

This next section is useful if you have not used source control systems before. I will define some of the vocabulary and explain the basic ideas of source control.

1.3.1 Check out systems

When describing the older source control systems, like SCCS, I said it managed the changes for a single file and also prevented multiple people from working on the same file at the same time. This is representative of a whole class of source control systems. In these you have the idea of "checking-out" a file so you can edit it, while becoming the current "owner". At the same time, while other people using the system can see who is working on the file, they are prevented from touching it. They can get a read-only copy so they can say build software but only the "owner" can edit it. When done editing the "owner" checks it back in, after which anyone else could work on on it. At the same time the system has recorded who had it and the changes made to it.

This system works well in small groups with real time communication. A common problem is that a file is checked out by some one else and you have to make a change in it. In a small group setting, just a shout over the cube wall will solve the problem.

1.3.2 Merge systems

In systems represented by CVS or Subversion the barrier is not getting a file to work on but putting it back under version control. In these systems you pull the source code files to a working directory in your area. Then you edit these files, making necessary changes. When done you commit or check them back into the repository. At this point they are back under version control and the system knows the changes from the last version to this version.

This gets around the problem mentioned above when others are blocked from working on a file. You now have the opposite problem in that more than one person can edit the same file and make changes. This is handled by the check-in process. There only one person at a time may check in a file. That being the case, the system checks the file and if there are changes in the repository file that are *not* in the one to be checked in the check in process stops. The system will ask if

the user wants to merge these changes into his copy, after correcting manually for areas of overlapping change. Once that is done the new version of the file can be checked in.

This type of system is used on large projects like the Linux kernel or other systems where you have a large number of geographically distributed contributors.

1.3.3 Distributed systems

The representatives of two major systems we have described thus far are centralized. That is there is only one repository on a single server. When you get a copy of the files or check in files, it all goes to just one place. These work and can support many, many users. A distributed system is an extension of this where it allows the repositories to be duplicated and has mechanisms to synchronize them.

With a single server, users of the repository must be able to connect to it to get updates and to check in new code. If you are not connected to the server you are stuck and cannot continue working. Distributed systems allow you to have your own copy of the repository to continue working and, when back in communication, to synchronize with the server. This is very useful where people take their work home and cannot access the company network. Each person can have a copy of the repository, continue working, and re-sync upon return to the office.

1.3.4 Common terms

The following is a list of terms I will use when talking about version control or Fossil.

Repository This is the store where the version controlled files are kept. It will be managed by a source control system.

Source control system This is software that manages a group of files by keeping track of changes and allowing multiple users to modify them in a controlled fashion.

Commit in Fossil Store the current set of new and changed files into the repository.

Trunk The main line of code descent in a Fossil repository.

Branch A user defined split in the files served by an SCS. This allow multiple work points on the same repository. Older branches (versions) might have bug fixes applied and newer branches (versions) can have new features added.

Fork In Fossil, an involuntary split in the code path occurs when a file in the repository has changes not in a file to be committed.

Chapter 2

Single user

2.1 Introduction

If you have read this far and are at least persuaded to try, you will want to put one of your projects under software control using Fossil. This chapter is set up to lead you through that task and show you how to adapt your development to using this tool. The assumption is made in this section that you will be the only person using the repository, you are the designer, developer, and maintainer of this project. After you are comfortable using the tool, the next section will show how you use it when you have multiple people working on a project.

2.2 Creating a repository

2.2.1 Introduction

In the spirit of "eating one's own dog food" we will use this book as the project we are going to manage with Fossil. The book is a directory of Markdown text files and the current working area looks like this:

```
fossilbook3
- content
  - book.md
  - image
    - book
      - fossil.png
  - index.md
  - introduction.md
- makefile
- outline.txt
- pandoc
```

```
- metadata-general.yaml
```

It took just an hour or so to start preliminary research and build the framework. Since that's about all I'm going to do today I want to build a repository and put all this stuff under Fossil control.

2.2.2 Create repository

I have a directory called FOSSIL in which I keep all my repositories, Fossil doesn't care but it helps me to keep them all in one place so I can back them up. First I need to create a new repository for the book. This is done using the command line after I move into the Fossil book directory called `fossilbook3`. (Here I have the `fossilbook3` directory side by side with the `FOSSIL` directory).

```
\$ cd fossilbook3 \$ fossil new ../FOSSIL/FossilBook.fossil
```

I create my repositories with the extension `.fossil`, this will be useful later with the `server` command (see section 7.2.15). When the repository is createed, an initial password is assigned with an admin user of "jim" (i.e. me).

2.2.3 Connect repository

The repository is created but is empty and has no connection to the `fossilbook3` directory. The next step is to open the repository to the `fossilbook3` directory with the `open` command.

```
\$ fossil open ../FOSSIL/fossilbook3.fossil
```

This will register a connection to the repository and do an initial checkout into the current directory (which is `fossilbook3` in my case). The `open` command seemingly did nothing, as if no files were created. However, checking with the `status` command shows the repository, the directory it's linked to and that we are hooked to the trunk of the store.

```
\$ fossil status

repository: /home/jim/FOSSIL/fossilbook3.fossil local-root:
/home/jim/fossilbook3/ config-db: /home/jim/.config/fossil.db checkout:
21e43f604387da163ea84f0d2f73cbaefff0f681 2020-11-23 18:46:21 UTC parent:
2baae1d5fb4865028ba9f4b104f8fed2f9b44094 2020-11-23 12:45:17 UTC tags:
trunk comment: initial empty check-in (user: jim)
```

The `extra` command shows all the files in the directory that are *not* under control of Fossil. In this case that's all of them since we have not checked in anything.

```
\$ fossil extra

content/introduction.md
```

Note, the status message says `initial empty check-in`. So, Fossil provides us with some basis check-in that is empty. It is the root of your own later check-ins. Note also, you can open the repository more than once. You just need to do this in different working directories. For example, this can be used to work on different branches (see section 4) at the same time.

2.2.4 Add and initial commit

I must now add all the relevant files into the repository with the `add` command. The Fossil `add` is recursive so if I add the top level files it will automatically recurse into the subdirectories and get those files too. Before you do an `add` it pays to tidy up your directory so you don't accidentally add a bunch of transient files (like object files from a compile). It's easy to remove them later but a little tidying before hand can save you some work.

```
\$ fossil add .
```

I simply told fossil to do an `add` of the current directory (`.`) so it got all those files and all the files in the subdirectory. Note the file named `_FOSSIL_` or `.fslckout` (depending on your operating system) that wasn't added to the repository but is located among your working files on disk. This is the tag file that fossil keeps in a directory so it knows what repository it belongs to. Fossil won't add this file since it manages it separately, but everything else is fair game.

One final thing before I can quit for the day, these files have been added or rather they will be added to the repository when I commit them. That must be done and then we can relax and let Fossil manage things.

```
\$ fossil commit -m "Initial Commit"
```

I added a comment text to the commit ("Initial Commit") and it's a good idea to always do this. When later we see the timeline of the commits you will have notes to tell you what was done.

2.2.5 Fossil start up summary

fossil new <name> Creates a new fossil repository.
fossil open <repository> While in a source directory connects this directory to the fossil repository.
fossil add . Will add (recursively) all the files in the current directory and all subdirectories to the repository.
fossil commit -m "Initial Commit" Will put all the currently added files into the repository.

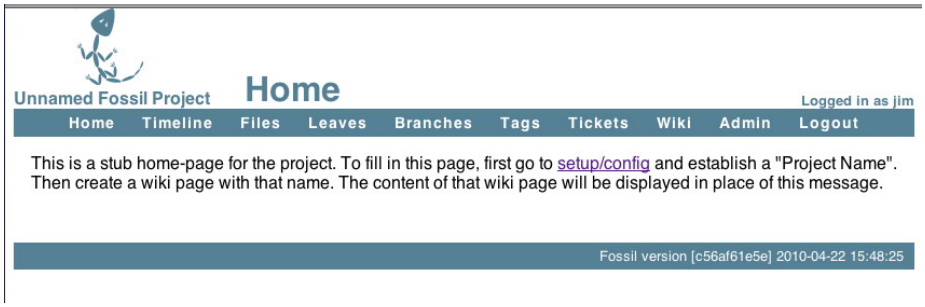
2.3 Set up user interface

One of the surprising features of Fossil is the webserver. This allows it to have a GUI type user interface with no operating system specific code, the GUI is the web browser supplied by your OS. In the previous steps I checked in my project to a Fossil repository, next I have to prepare the web interface for production use.

The Fossil web server uses the first available port starting at 8080 instead of the standard port 80 for all HTTP access. When run it will automatically start your Web browser and open the repository home page.
Fossil automatically finds an open port and will give a message on the command line telling you what port number is used. You can still use the `-port` option if you want to control the port number.

```
\$ fossil ui
```

This command shows how it's started. When I do this my browser starts if it is not running already and I am presented with the following home page:



Following the advice on the page I go to `setup/config`. I am going to do the minimum setup that you should do on all projects. As you get familiar with Fossil you will probably have many more things that you will customize for your taste but what follows are the only things you *have* to do.

I have entered in a project name and put in a description, the project name will be the name of the initial Wiki page (see section 7.4.5) and the description is useful for others to see what you are doing here. Then I go to the bottom of the page and pick the `Apply Changes` button.

Next I pick the `Admin` tab (you can see it in the header bar) and the pick `Users` from that page. I am presented with the users and will use this to set the password of the project. See section 6 for details on what else you can customize from the `Admin` page.

As you can see Fossil automatically configures a number of users beyond just the creator. The `anonymous` user you have already seen if you went to the Fossil web site to download the code. This user can view and get code but

cannot commit code. On the right side of the page are the many options you can give to a user, it's worth reading it all when you set up your repository. The important one is me (jim) which has "s" or "Super User Capabilities". This means I can do anything with the repository.

I will now edit the user "Jim" to make sure it has the settings I want. In this case you *must* set the password. Remember way back where Fossil set it during the creation of the repositroy (Figure[fig:Create-Repository]), it's a very good idea to change this to something you can remember rather than the original random one.

I have put in my contact information (e-mail address) and while you cannot see it I have typed in a password that I will remember. Then I applied the changes.

Now the repository is ready for further work, it's rather bare bones at this point but the most important things are set up.

2.3.1 User interface summary

fossil ui Run in the source directory will start a browser based user interface to fossil.

fossil ui -port <IP port #> Can be used if port 8080 if already in use on your system.

On the first run it is important to configure your project with a name and set the password for yourself.

2.4 Update repository

After writing the above section of the book I now have created a some new files and changed some of the existing files in the repository. Before quitting for the day I should add these new files into the repository and commit the changes saving this milestone in the project.

```
\$ fossil extra
```

I run `fossil extra` to see these new files. Choose the ones to add and note if there are perhaps any temporary files that should *not* be stored in the repository. I also ran `fossil status`. This shows changes to files that are already in the repository. The only files changed are `content/introduction.md` and `content/single-user.md`.

All I have to do now is add in the directory `content/image` which will add in the image files I want in the repository. Then I commit the changes to the repository and we can move on to other tasks of the day.

```
\$ fossil add content/image
```

After doing this commit I can bring up the Fossil `ui`(see section 7.2.15) and view the project `Timeline` by picking that tab on the Fossil header. We get this:

```
Timeline placeholder
```

You can see all my check-ins thus far and you can see after the check-in from Figure [fig:Update-for-new] I did another check-in because I missed some changes in the outline. The check-ins are labeled with the first 10 digits of their hash value and these are active links which you can click to view in detail what was changed in that version.

I clicked on the very last check-in (the LEAF) and the display is shown above. There are many things you can do at this point. From the list of changed files you can pick the `diff` link and it will show in text form the changes made in that particular file. The `Other Links` section has a very useful ZIP Archive. Clicking this will download a ZIP of this version to your browser. You will find this useful if you want to get a particular version, in fact this is normally how you get a new version of Fossil from https://fossil-scm.org/. The edit link will be used later to modify a leaf.

2.4.1 Update summary

fossil status will tell you the updated files before you commit.
fossil extra will list files not in the repository.
fossil commit - m "Commit comment" Commits a change (or changes). It is very important to have a descriptive comment on your commit.

2.5 Tickets

Besides managing your code Fossil has a trouble ticket system. This means you can create a ticket for a problem or feature you are going to add to your system then track your progress. Also you can tie the tickets to specific check-ins of your files. For software this is very useful for bug fixes and feature additions. For example you can look for a bug in the ticket list then have it take you to the change that fixed the problem. Then you know exactly what you did and not have to be confused by other changes you might have made.

When you click on `Tickets` in the meu it will bring up this window. You can create a new ticket, look at the list, or generate a new report. Keeping things simple I will just use the `All Tickets` list for now.

`Initial Ticket Window placeholder`

Picking `New Ticket` I get a form that I fill out like so:

`Ticket Form placeholder`

Pretty simple actually. You can put as much or as little detail in here as you wish, but remember this stuff might be really vital six weeks or six months from now so think of what you would want to know then.

Note, there is no `Submit` button now. I need to click on `Preview` in order to check how the ticket will be formatted before I can submit (and there are four different ways of formatting tickets). When everything is OK, I press `Submit`. I get this showing what I entered.

`Viewing a Ticket placeholder`

Finally picking `Tickets` then `All Tickets` I can see my new ticket in the list marked as "Open" and in a distinctive color.

`Ticket List with open ticket placeholder`

I try, in handling tickets, to have links from ticket to the commit that addressed the problem and a link from the commit back to the offending ticket. This way looking at the ticket I can get to the changes made and from the timeline I can get to the ticket and its resolution. To do this I will make sure and put the 10 digit hash label from the ticket into the check-in comment and put a link in the resolved ticket to the check-in.

Since I have now written the chapter and put in all these images of what to do I can now add in all the new images to the repository and check this in as another completed operation. And I do that like this:

```
\$ fossil add Images/single-user
```

First I added in all the new image files. I am lazy and just told it to add in all the files in the `single-user` directory. I have previously added some of those like config-initial.png but Fossil is smart and knows this and won't add that one twice. Even though it shows it "ADDED", it really didn't.

The commit line is very important, as you can see I put the 10 digit hash code for the ticket in brackets in the comment. As we will see in the Wiki section (section 7.4.5) this is a link to the ticket, so when viewing the comment in the

timeline or elsewhere you can click the bracketed item and you would go to the ticket.

Now that I have the items checked in I have to close the ticket. I do that by clicking on its link in the ticket list, that will go the the `View Ticket` window as shown in Figure [fig:viewticket]. From there I pick `edit` and fill it in as shown:

```
figure placeholder
```

I mark it as "Closed". If you have code you can mark this as fixed, tested, or a number of other choices. Another very important step, I brought up the timeline and copied the link for the commit I had just done in Figure [fig:checkin]. By doing this my ticket is now cross linked with the commit and the commit has a link back to the ticket.

2.5.1 Ticket summary

- Tickets are a useful way of reminding you what needs done or bugs fixed
- When you commit a change that affects a ticket, put the 10 digit hash label of the ticket into the commit comment surrounded by brackets, e.g. `[<10 digit hash>]` so you can link to the ticket
- When you close the ticket put in the hash label of the commit that fixed it.

The ticket system is widely customizable. See section 6.1.

2.6 Wiki

As we saw in Figure [fig:Starting-Webserver] Fossil has a browser based user interface. In addition to the pages that are built in you can add pages to that web site via a wiki. This allows you to add code descriptions, user manuals, or other documentation. Fossil keeps all that stuff in one place under version control. A wiki is a web site where you can add pages and links from within your browser. You are given an initial page then if you type `[newpage]` this text will turn into a link and if clicked will take you to a new blank page. Remember in Figure [fig:Initial-Webserver-page] that is the initial page for your project and from there you can add new pages. These pages are automatically managed by the Fossil's version control system. You don't have to add or commit.

Since I did the setup on the repository (see Figure [fig:Initial-Configuration]) the home page has changed to this:

```
Home page placeholder
```

Not very helpful so the in rest of this chapter I will use the Wiki functions of Fossil to make this more useful. If I pick the Wiki item from the menu bar I get:

`Wiki page placeholder`

These are the controls that let you manage and modify the wiki. In essence, this is the 'Help' page of the wiki. Most important for now is the `Formatting rules` link. This link takes you to a page that describes what you can do to format a wiki page. If you just type text on a page it will appear but be formatted by your browser. You can type HTML commands to control this formating. It's worth your time to carefully read this page and note what you can and cannot do. The page just lists the valid HTML commands, and if you don't know what they mean, I would suggest you find a page like this https://web.stanford.edu/group/csp/cs21/htmlcheatsheet.pdf and keep it handy.

Besides the HTML markup the most powerful command for the Wiki is `[page]` where it links to a new page. This is how you add pages and how you build your web site of documentation for the repository.

The wiki is customizable. E.g. you may want to show the list of wiki pages instead of the help page when picking the wiki from Fossil's menu. See section 6.2 for ths and for other details.

2.6.1 Wiki formatting

I now begin work. What I want to do is change the home page to be non-empty and also put a link on the home page to the PDF of this book. In Figure [fig:Wiki-controls] I click on the first item, the FossilBook home page. This takes me to the home page again but now I have an "Edit" option. We also have a "History" option so I could look at older versions of the page.

`This shows my initial edit and a preview:`

The page shows an edit window where I type things I want displayed and at the top is a row of tabs inclusing a "Preview" tab showing how the page will lookl ike. As you can see I typed some simple HTML to make a large and centered title. The body of the text I just typed and as you see the browser fits the text to the screen. You can have multiple paragraphs by just putting blank lines between your text. Next I wanted a bulleted list and this is done by typing two spaces, a '*' then two more spaces. On each of these lines I have a link to a new (not yet created page). If you look I put these in the form `[<new page> | <title> ]`. This way I can have a long string that describes the link but have a nice short (no embedded spaces) page name.

OK, I will save my changes and then go to the new pages. I am doing some complicated things here. The first link is to the book PDF. This will be a file I create in the repository. The Pandoc program I'm using creates the PDF. I will do that, save it, and add it to the repository. But I don't want to link to a static

file, that is I want the most current version of the PDF, the one I save each time I quit for the day. To do this we have to put in a link that looks like this:

```
[http:doc/tip/FossilBook.pdf | Book (pdf) ]
```

This is a special link the Fossil wiki understands, `doc` says this is documentation. `tip` says use the most current version; you could put a version link here. And finally since I am going to put the book PDF at the top level I only need the file name. If it was in a subdirectory I would have to say `doc/tip/subdir/filename`.

The second link is just to a regular page and I can just edit that one just like I did this the main page.

2.6.2 Wiki summary

- Format your text using HTML commands such as `<h1>Title</h1>` for page headings
- Create and link pages using `[<page> | <Link text>]`
- The page designation `http:doc/tip/\<document path relative to repository>` will display any document in the repository that your browser can handle (i.e. pdf, http etc)
- Never click on a link till *after* you have saved the page

Chapter 3

Multiple users

3.1 Introduction

In the previous chapter I went through using Fossil with just one user (me). In this chapter we will get into using it with multiple users. Thanks to Fossil's distributed design once the set up is done using it is not much different than the single user case with Fossil managing automatically the multiple user details.

3.2 Setup

In the previous chapter the Fossil repository was a file on our system and we did commits to it and pulled copies of the source from it. Fossil is a distributed source control system. What this means is that there is a remote server repository in a place that all users can access. Each user has their own "cloned" copy of the repository and Fossil will automatically synchronize the users repository with the remote repository. From a each user's perspective you have your local repository and work with it using the same commands shown in chapter 2. It's just that now Fossil will keep your local repository in sync with the remote repository. This remote repository acts as a server.

3.2.1 Server

I have the FossilBook.fossil repository and now have to put it in place so multiple users can access it. There are two ways, the first is using fossil's built in webserver to host the file and the second is using the operating systems supported web server (if present) and a cgi type access.

3.2.1.1 Self hosted

A self-hosted server is quite simply the easiest way to do it. The downside is that you are responsible for keeping the machine available and the webserver up. That is, don't turn the machine off when you quit for the day or some other user is going to be upset. All I have to do is this:

```
\$ fossil server FossilBook.fossil \&
```

This is on a UNIX system, the "&" at then end of the command line runs the fossil webserver in the background. If I know this machine has an IP address of 192.168.1.200 then from any other machine in the network I can type into my browser:

http://192.168.1.200:8081

and I can access the Fossil web server.

As you can see this is simple and works on any system that runs Fossil. As long as you carefully make sure it's always running and available for others this can be a very easy way to make the repository remotely available.

The problems with this method are:

1. If you have multiple repositories you have to use the server not the ui command, have all your repositories in the same directory, and have them all use the extension .fossil.
2. If the machine goes offline (i.e. for OS update) or other reason it might not automatically restart the Fossil servers.
3. Backup of the repositories might not be done.

This method does work, and if you only have one repository and a diligent owner of the remote machine, it will work and work well.

3.2.1.2 Server hosted

If you have a server type machine available (i.e., a Linux or UNIX box) that is running Apache or a Windows machine running IIS you can let it be the webserver for your repository. This has a number of advantages: this machine will be up all the time, it will probably be automatically backed up, and it can easily support multiple Fossil repositories.

I am not going into how to set up the webserver or how to enable CGI (Common Gateway Interface). See the following sites:

- For IIS see e.g. here https://www.ibm.com/docs/en/cognos-analytics/11.1.0?topic=services-configuring-cgi-gateway-iis-version-7-later and just ensure that your fossil.exe is in the path to be run by the cgi script.
- For Apache see here http://httpd.apache.org/docs/2.4/howto/cgi.html and ensure you know the directory where the CGI scripts are stored.

If you are not in control of the webserver you will need the help of the server admin to enable CGI and to copy your CGI scripts to the correct location.

3.2.1.2.1 CGI Script for hosted server If we assume an Apache server and, in my case, the cgi directory path is /Library/Webserver/CGI-Executables, then we have to write a script of the form:

```
#! <Fossil executable location> repository: <Fossil repository location>
```

and put it into the cgi script directory. I have put my Fossil executable into `/usr/local/bin` and I am putting my Fossil shared repository into `/Users/Shared/FOSSIL`. My script then becomes:

```
>  #!/usr/local/bin/fossil # Put the book repository on the web repository: /Users/Sh
```

After making the script I then copy it to the CGI directory and allow anyone to execute it.

```
sudo cp Book.cgi /Library/Webserver/CGI-Executables/Book.cgi
```

3.2.2 Test the setup

If all is in place then I should be able to access the webserver and get to this:

```
Web access to Fossil CGI hosted site - placeholder
```

3.3 User accounts

Serving a repository, either self hosting or the more complicated CGI method gets you to the same place as shown in Figure [fig:Web-access-to]. Now I have to set up user accounts for the other contributors to this book. Remember Fossil has automatically created an Anonymous user (see Figure [fig:User-Configuration]) thus others can access the site in a limited way, that is they can download the book but cannot commit changes. In this case I want to create a new account (Marilyn) that can make changes and commit changes as she is my editor.

To accomplish all this first I have to login by going to the log in page and entering my ID (jim) and my password. Now since I'm super-user I then go back to the User-Configuration page, Figure [fig:User-Configuration] and add a new user:

```
New Editor user - placeholdere
```

Since she is going to be an editor, this will be similar to a developer if we were doing code, so I picked the Developer privilege level. This way she can get the repository, check-in, check-out, and write and update tickets. I also added the attachments since she might need that to put on an image or other comment on actions she is doing. I also gave her a password so her access can be secured.

I could add other users at this point but don't need any others for this project, but you can see how easily this can be done. When you assign the user privileges just read carefully and don't give them any more than you think they need. If they have problems, you can easily modify their account in the future.

3.4 Multiple user operation

With the server set up and the user established the next thing to do is clone the repository. This means to make a copy of the webserver repository on my local machine. Once that is done this local repository uses the same commands and is very much like single the user use discussed in section 2. Fossil will synchronize your local repository with the one on the server.

3.4.1 Cloning

To clone a Fossil repository you have to know four things:

1. It's web address, for our repository. It is https://fossil-scm.org/FossilBook
2. Your account name, in my case it's jim
3. Your password (which I'm keeping to myself thank you...)
4. The local name of the repository, in this case I'm calling it FossilBook.fossil

You then go to where you want to keep the repository (in my case the FOSSIL directory) and use the clone command:

```
\$ fossil clone https://jim:@fossil-scm.org/FossilBook FossilBook.f
```

At this point I can go through the steps outlined in section 2 to set my user password and then open the Fossil repository on a working directory.

Now that I've moved everything to the new cloned repository I do a check in the end of the day which looks like this:

```
\$ fossil commit -m "Moved to clone repository"
```

As you see the files were committed locally and then the local repository was automatically synchronized with the repository on the server.

3.4.2 Keeping the code in sync

After doing all the setup described above I now have a distributed source control system. My co-worker, Marilyn has also cloned the repository and begun work. She is editing the book correcting my clumsy phrasing and fixing spelling mistakes. She is working independently and on the same files I use. We must use Fossil to prevent us from both modifying the same files but in different ways. Remember Fossil has no file locking, there is nothing to prevent her from editing and changing a file while I work on it.

This is where we both must follow procedures to prevent this sort of problem. Even though she edits files I cannot see the changes till they are committed. Two different versions of the same file won't be a problem till I try to commit with my version and her version is in the current leaf.

There are two problems:

1. Before I do any work I must be sure I have the current versions of all the files.
2. When I commit I must make sure what I am committing has only my changes and is not stepping on changes she has done.

The first is pretty obvious. You make sure you have the latest version before you do anything. We do that with the `fossil update` command. In figure [fig:Cloned-repository-checkin] I had done my latest check in. Before starting any more work I should ensure that Marilyn hasn't checked in something else. I could check the timeline but instead I'll do an update to my repository and source files. When I do the update I specify it should be updated from the trunk. This ensures I get it from the latest and greatest ... and not some branch.

```
\$ fossil update trunk
```

Ah ha! Marilyn has been at work and updated the book source and pdf. If I check the timeline from the webserver I see she has even documented it:

Now I know I have the current state of the world and I can proceed to add in new sections.

In the default configuration, Fossil operates in 'auto-sync' mode (configurable with the `fossil settings` command). This means that Fossil makes sure changes by others are synced from the Fossil server when the `fossil update` command is used. When doing a `commit`, Fossil will automatically push your changes back to the server you cloned from or most recently synced with. When doing an `update`, Fossil will first go to that same server and pull the recent changes to your local repository, then merge them into your local source tree.

3.4.3 Complications

The second problem described in section 3.4.2 is much harder. In this case I have diligently done my `fossil update` and started working. In the mean time Marilyn has also done her update and also started working. Now she is done and checks in her changes. I obviously don't know this since I am happily working on my changes. What happens next ...

```
\$ fossil commit -m "Commit that might fork"
```

Ah ha, that very thing has happened and Fossil warned me that my copy of the file differs from the master copy. If I had provided the option `-force` to the `commit` then the repository would generate a fork and Marilyn's future commits would be to her fork and my commits would be to mine. That would not be what we want since I want her to edit my copy of the book.

The next step would be to do as Fossil says and do an `update`. At this point you have to be careful since blindly updating the changed files could overwrite the stuff I've just done. So we do a trial update by using the `-n` and `-v` options to say "do a dry run" and show me the results_

```
\$ fossil update -n -v
```

That's a little more than I wanted as you can see almost everything is `UNCHANGED` but it shows that my file needs a MERGE and fossilbook.pdf needs an `UPDATE`. This is what I should expect, Marilyn has done edits to the ... file and so have I. So we have to merge the changes. But she has also updated the fossilbook.pdf which I have not. Before we go on if you are running on Linux or UNIX you can simplify this dry run by doing:

```
\$ fossil update -n -v \textbar{} grep -v UNCHANGED
```

By using the pipe and `grep` I can eliminate all those extra `UNCHANGED` lines.

3.4.4 Fixing the update file

First we fix the easy file, the fossilbook.pdf I can just `update` by itself so it matches the current repository. It doesn't need to be merged, so I just replace it. Before I do that I have to look at the repository time line

```
Placeholder for Timeline figure
```

I see that the current *Leaf* is ...[d44769cc23]... and it is tagged as *trunk*. I want to update the fossilbook.pdf from there. So I say:

```
\$ fossil update trunk fossilbook.pdf
```

and it's done.

3.4.5 Fixing the merge file

We can use the tools built into Fossil. In this case noticing that `commit` will cause a fork, Jim will use the `-force` option to cause the fork and will handle the merge later.

```
\begin{quote}
fossil commit -m "adding some changes of jim"
\end{quote}
```

Now the timeline looks like:

```
Placeholder for Timeline figure
```

To remove this fork (i.e. get the changes Marilyn did into the trunk) we use the Fossil `merge` command. We can use the `merge` because . . . fossilbook.lyx. . . is a text file and the merge markers are designed to work with text files. If it was a binary file we might have to use an external file or copy and paste between the two file versions using the handler program for the file.

```
\begin{quote}
fossil merge a91582b699
\end{quote}
```

Looking at the file (fossilbook.lyx) in a text editor (not LyX) we find:

```
> \>>>>>>> BEGIN MERGE CONFLICT

Placeholder for now
```

After the commit the timeline shows how the merge brought the fork back into the main *trunk*. Marilyn will then have to update to this new trunk. (See Section [sub:Updating-by-others])

3.5 Sharing other changes than code

3.5.1 Sharable content

When multiple users are working on the code, they may also want to use their local clone of the repository for tickets, the wiki and other stuff. So, while the local code changes are synchronized with the server using the fossil commands `update` and `commit` (or `pull` and `push` if you do it manually), other parts of the repository are not synchronized. If you want to see the current set of tickets or the wiki in your local copy of the repository as well, you need to synchronize those with the command `fossil sync`.

This command will synchronize all sharable content:

- check-ins
- wiki pages
- tickets
- forum posts
- tech notes
- unversioned content (using the additional option `-u`)

This is particularly useful if you cannot work on the remote server for a reason. You can start your local server with `fossil ui`, make changes to e.g. wiki pages and the sync the changes back to the remote server with `fossil sync`.

3.5.2 Synchronize the Fossil configuration

When cloning the remote repository, you also get the current configuration of the repository. This includes e.g. the skin and the ticket setup. When these settings change on the server, they will not automatically be synchronized with your local clone of the repository. So you may end up in a situation where you create a new ticket locally that does not reflect the structure of the tickets on the server because the server configuration has changed! So, if some important configuration changes on the server, you can (and should) pull those changes into your local clone. This is done using the `fossil configuration` command (see the section on the 7.5.4 command for all details). It can be used to change the configuration of the following areas:

- email
- interwiki
- project
- shun
- skin
- ticket
- user
- alias
- subscriber

So, to pull a changed ticket configuration into the local clone, you issue the following command:

```
\begin{quote}
fossil configuration pull ticket
\end{quote}
```

Using the `configuration command`, you can also import and export configurations via text files and even do a kind of merge operation. This makes it possible to edit more complex configurations in an editor of your choice (e.g. when you want to have some syntax highlighing). You can also push your local changes back to the server but that will require administeation priviliges on the remote server. Last but not least, you may want to do a 'real' sync, synchronizing configuration changes in the local repository with the remote repository. When you use the keyword 'all' for the area, all of the above areas will be synchronized:

```
\begin{quote}
fossil configuration sync all
\end{quote}
```

Chapter 4

Forks and branches

4.1 Introduction

This chapter will cover forking and branching in Fossil. **Forking** is where you *unintentially* create two places to check into a repository. **Branching** is where you *intentially* do this because you want to maintain two or more versions of the code in the same repository. We illustrated forking and it's solutions in section 3.4.3. If, instead of fixing (merging) the file then doing the commit, we forced the commit, Fossil would fork the repository.

Forking is something to avoid because it creates two checkin paths for the code. Thus different users will be on different paths and can check in contradictory changes. Branches on the other hand are forks that you desire. These occur when you want to have two different versions of the code base in development at the same time. A typical scenario is having a production verison of the code under maintenance and a development version both served from the same repository. In this case development changes should only be checked into the development branch. Maintanence changes might have to be checked into both.

Instead of using the book repository for these examples we will use a JSON parser program that has a number of files and documentation. This will make it simpler to illustrate branching and tagging.

There is a good discussion of these topics on the Fossil Web site: https://fossil-scm.org/home/doc/trunk/www/branching.wiki.

4.2 Forks, branch and merge

In this case the JSON code has just been placed in Fossil and two developers check out copies to work on. Jim wants to fix a number of compiler warnings that appear and Marilyn wants to fix the documentation. In both cases they

proceeed as shown in chapter 3. The JSON code has been placed in a distributed location, each of them clones the repository, and opens a working copy of the code.

4.2.1 Marilyn's actions

She looks through the documentation and finds a number of problems and fixes them (the documentation uses LyX and PDFs). When she is satisfied with what she has done, she checks the current version of the documentation in:

4.2.2 Jim's actions

At the same time, Jim gets a working copy of version `[6edbaf5fa8]` of the code, puts in a ticket `[d23bf4bbbb]` as shown in Figure [fig:Marilyn's-work]. After fixing the warnings, Jim is done and goes to commit. He does this *after* Marilyn has done her commit:

```
jim\$ fossil commit -m "{[}d23bf4bbbb{]} Remove warnings"
```

At this point Fossil recognizes that Marilyn has changed the repository (she updated the documentation) but Jim does not have these changes because he checked out an earlier version of the code. Jim says he must get his changes in so he does a FORCE to force fossil to accept the commit:

```
jim\$ fossil\{\} commit -m "{[}d23bf4bbbb{]} Remove warnings" -f
```

Looking at the timeline Jim sees this:

Not good, there are two Leafs and Marilyn would commit code to her fork and Jim would be commiting code to his. So Jim must fix this by merging the code. Jim wants to merge versions `[b72e96832e]` of Marilyn and his `[1beab85441]`.

4.2.3 Fixing the fork

So Jim who's checked out code is from Leaf `[1beab85441]` does a merge with Marilyn's leaf `[b72e96832e]` like so:

```
jim\$ fossil merge b72e96832e
```

As shown the two documentation files are updated, there are no merge conflicts as Jim didn't touch these files and Marilyn didn't touch the code files.

Next Jim does a commit to make this new merged set of files the new trunk. Remember doing the merge shown in Figure [fig:Merge-Operation] just updates your checked out code and does not change the repository till you check it in.

```
jim\$ fossil commit -m "after-merging-in-changes"
```

When we look at the timeline we have a single leaf for future code commits.

The only other thing remaining is that Marilyn does an `update` before proceeding so her checked out code matches the repository.

```
marilyn\$ fossil update
```

4.2.4 Commands used

fossil merge <fork> Used to merge a fork (specified by hash value) to current check out.

fossil update <version> Used to update current check out to specified version, if version not present use default tag for check out (see `fossil status`)

4.3 Merge without fork

In this case I will show how to merge in code changes from multiple users without causing a fork. In this case Marilyn has put in a BSD license text into all the code files while Jim is adding a help function to the code. In this case both of them put in tickets saying what they are doing but acting independently.

4.3.1 Check in attempt

Marilyn finished first and checks in her changes. Jim builds, tests and tries to check in his code and gets:

```
jim\$ make
```

4.3.2 Update

The next action Jim takes is to do the update but without doing changes, using the `-n` flag which tells it to just show what's going to happen without making any file changes.

```
jim\$ fossil update -n
```

This shows some files will be updated, i.e. be replaced by new text from the repository. The `main.c` file will be merged with the version from the repository. That is text from the repository will be mixed with the text from Jim's modified file. Note that it says MERGE meaning the two sets of text are a disjoint set. This means the merge can all be done by Fossil with no human intervention.

Jim can just do the update for real then commit the merged files to make a new leaf. So now we have Marilyn's and Jim changes combined in the lastest version.

4.3.3 Commands used

fossil update -n Does a dry run of an update to show what files will changed.

– UPDATE Implies file will be replaced by repository file – MERGE Implies file will be mixed text from repository and check out

4.4 Branching

4.4.1 Introduction

We have discussed this before but branching is the intential splitting of the code in the repository into multiple paths. This will usually be done with production

code where we have maintenance branch and a development branch. The maintenance branch is in use and would get bug fixes based on experience. The development branch would get those changes if applicable plus be modified to add features.

The JSON code parser has been tested and works so will be released to general use. Also we wish to modify it to add support for UTF-8 characters so it matches the JSON standard. The current version just works with ASCII 7 bit characters which is not standard. We wish to split the code into a VER-1.0 branch which is the current code in use and VER-2.0 branch which will add UTF-8 character support.

4.4.2 Branch the repository

Before proceeding we will make sure we have the current trunk code in our check out.

```
\$ fossil status
```

Seeing that matches the latest leaf in the time line we can proceed to branch the code.

```
\$ fossil branch new VER-1.0 trunk -bgcolor 0xFFC0FF
```

What was just done? We used the Fossil branch command to create two branches VER-1.0 and VER-2.0 and assigned each of them a color. We can see the timeline is now:

Instead of creating the branch explicitly, we could also have continued changing the code in the trunk. When doing the `commit` we can still decide that those new changes shouldn't go into trunk anyway but into a new branch. For this, we just need to specify the new branch with the `commit` command:

```
\$ fossil commit -branch VER-1.0 -branchcolor 0xFFC0FF
```

4.4.3 Color setup

As you see above the two branches have different colors in the timeline. This was due to the -bgcolor option added when we created each branch. (See Figure [fig:Branch-commands]). But we want this color to appear on subsequent checkins of each of these branches. To make that happen we have to set the options using the UI and picking a particular leaf on the timeline.

Under the 'Background Color' section I have checked 'Propagate color to descendants' so future checkins will have the same color.

4.4.4 Check out the branches

Now the the repository is branched we can check out the two sets of code into different directories. We create jsonp1 and jsonp2 and proceed to open the different branches into them.

```
\$ cd jschimpf/Public/jsonp1 \$ fossil open ../FOSSIL/jsonp.fossil
VER-1.0
```

Checking out VER-2.0 in the same way

```
\$ cd jschimpf/Public/jsonp2 \$ fossil open ../FOSSIL/jsonp.fossil
VER-2.0
```

Notice on both of these the tags show which branch we are attached to.

4.4.5 Correcting errors in both

After doing this work I found that the `main.c` file had a warning about an unused variable. I wanted to correct this in both branches. At this point all the files in both branches are the same so correcting the file in either branch and copying it to the other is possible. I put in a ticket for the change and edit `main.c`. I copy it to both checkouts for the both branches and then check both in.

```
\$ cd jschimpf/Public/jsonp1 \$ fossil commit -m "{[}2795e6c74d{]} Fix
unused variable"
```

Now the timeline looks like this:

4.4.6 Private branches

Sometimes you may want to create a branch but keep the changes in that branch private. E.g. , this may be some experimental code you do not want to share with the other users (yet). To do this, you just create a branch and declare it 'private'. This means, the branch will stay in your local copy of the repository but it will not be sychronized with the remote repository and thus not distributed to others during a commit:

```
\$ fossil commit -branch myTests-123 -private"
```

Of course, you can do the same with the `fossil branch` command. It also has the `--private` option. If you at some time decide to make your branch public, you can do so with the 7.3.17 command.

You can associate wiki pages with branchs using a name for the wiki page starting with `branch/`. Note though, although the code is private and not shared, the associated wiki page will be synchronized. Wiki pages are separate from all file-level content and do not take part in branching (or merging etc.). So, the wiki page is *associated* with that private branch but is not *a part of* that branch (in terms of the underlying artifact mechanism).
Source: Fossil forum post

4.4.7 Commands used

fossil branch Used to generate a branch of the repository. The command can optionally color the branch in the display.
fossil commit –branch Committing a new version into a new branch

Chapter 5

The Fossil user interface

Fossils graphical user interface is a web site directly served from the repository. It allows to inspect the history, documentation, tickets and other information on the repository. This interface is wah tusers will likely see in the first place when they want to learn about your software project.

5.1 fossil ui

{» ... about the 'fossil ui' command with reference to commands.md ... «}

5.2 Home

5.3 The timeline

The timeline presents a graphical overview of the development of your project. I t can list all kinds of events that happened within the project:

- Check-ins
- Tickets (and a list of new tickets)
- Wiki
- Tech notes
- Tags

...

5.3.1 Check-ins

Each check-in is presented as one point on the timeline.

...

5.4 File

5.4.1 The timeline of a single file

You can also display the timeline of a single file (as opposed to a check-in with multiple files). There are three possibilities to get there.

1. Navigating to the file, starting from the 'File' menu.

Browse to desired file. Click on the file link. you will see the content of the file if is is a text file. Below the main menu (on top), you will see a line starting with the word 'File' and followed by the file name (as a link) and the check-in context. Click on that file name link. This brings you to the `/finfo` page for this file showing the timeline for this single file.

2. Starting from the full timeline

Click on the check-in link (the hash value) in the timeline. You see some metadata (a section called 'Overview'), the timeline context (a section called 'Context') and below the diffs for files affected by this check-in (a section called 'Changes'). In that last section you will see some lines saying 'Modified . . . ' with a linked file name. Click on that link and you will get to the page showing the timeline for this single file.

There also is the possibility to see the history of all files in a specific directory (recursively). For this, go to the 'File' section, ensure you are in 'Flat View' (not a tree view that can expand/collapse) Browse to desired directory and then click on the 'History' link on second-level menu. You will see a timeline of check-ins that affected any file within that directory. You will, however, not get a list of the files affected during each check-in. For this, you need to click on the individual check-ins and then look into the 'Changes' section of that check-in.

5.5 Branches

5.6 Tags

5.7 Tickets

5.8 Wiki

5.9 Admin

5.10 Login

Chapter 6

Fossil configuration

Fossil can be configured or customized to you own needs in a variety of ways. This section cover the configuration of some aspects you can change via the Fossil `Admin` menu in the browser interface of Fossil.

6.1 Ticket configuration

6.2 Wiki configuration

Picking the Wiki item from the Fossil menu bar shows the wiki help page by default. You can change that. You may want to show the list of existing wiki pages instead so you can go to every wiki page from here. For this, you need to modify the Fossil skin. Pick `Admin` in the Fossil menu bar and click on 'Skins'. Follow the outlined procedure to initialize the skin editing process (steps 1 to 3; see section 6.4 for details). Then, edit the 'Header' (in Step 4). This will bring up the HTML and TH1 code for the Fossil menu (see section 9 for details on the TH1 language). The only thing you need to do is to change `\wiki` to `\wcontent` in the part of the code that handles showing the wiki menu item. After applying this change, you will need to end the skin editing by doing steps 7–5.

6.3 Users configuration

6.4 Skins configuration

6.5 Configure email notifications

Chapter 7

Fossil commands

7.1 Introduction

This section will go through the various Fossil command line commands. This will be divided into sections, the first will detail the must know commands. These are the ones you will be using all the time and will probably have memorized in short order. The other commands will be divided into Maintenance, Advanced, and Miscellaneous. These you will probably be checking as reference before use.

The most important command is help. You can always type fossil help at the command line and it will list out all the commands it has. Then typing fossil help <command> will print out the detailed information on that command. You always have that as your reference. This section of the book will try to supplement the built in help with some examples and further explanation of what a command does. All of the commands will be placed in the index for easy searching

NOTE: Fossil is a moving target, commands might be added and others removed future versions. Type fossil help on your version to get the latest list. The following applies to the fossil I used when I wrote this and your version might be different.

7.2 Basic

7.2.1 help

This command is used to dump the current command set and version of Fossil. It can also be used in the form fossil help <command> to get further information on any command.

Actually this will give you only a subset of the help commands, limited to the commands that are used most often. If you want to see all commands available then issue the fossil help –all command. Between versions you will see changes as to what is included in the help sub-set.

An example of using the help function to get further information about a particular command:

```
$ fossil help
Usage: fossil help TOPIC
Try "fossil help help" or "fossil help -a" for more options
Frequently used commands:
add          cat          extras       merge        rm           ui
addremove    changes      finfo        mv           settings     undo
all          clean        gdiff        open         sql          unversioned
amend        clone        grep         pull         stash        update
annotate     commit       help         push         status       version
bisect       dbstat       info         rebuild      sync
blame        delete       init         remote       tag
branch       diff         ls           revert       timeline
This is fossil version 2.14 [939a13d94f] 2020-11-20 19:28:05 UTC
```

Adding the optin `-a` (or `-all`) to the help command results in a listing of ALL the available commands:

```
$ fossil help -a
3-way-merge    clean          hook           rebuild        stash
add            clone          http           reconstruct    status
addremove      close          import         redo           sync
alerts         co             info           remote         tag
all            commit         init           remote-url     tarball
amend          configuration  interwiki      rename         ticket
annotate       dbstat         leaves         reparent       timeline
artifact       deconstruct    login-group    revert         tls-config
attachment     delete         ls             rm             touch
backoffice     descendants    md5sum         rss            ui
backup         diff           merge          scrub          undo
bisect         export         mv             search         unpublished
blame          extras         new            server         unset
branch         finfo          open           settings       unversioned
bundle         forget         pikchr         sha1sum        update
cache          fts-config     pop3d          sha3sum        user
cat            gdiff          praise         shell          uv
cgi            git            publish        smtpd          version
changes        grep           pull           sql            whatis
checkout       hash-policy    purge          sqlar          wiki
ci             help           push           sqlite3        zip
```

Use it to get further information about a particular command:

```
$ fossil help help
Usage: fossil help [OPTIONS] [TOPIC]

Display information on how to use TOPIC, which may be a command, webpage, or
setting.  Webpage names begin with "/".  If TOPIC is omitted, a list of
topics is returned.

The following options can be used when TOPIC is omitted:

   -a|--all          List both common and auxiliary commands
   -o|--options      List command-line options common to all commands
   -s|--setting      List setting names
   -t|--test         List unsupported "test" commands
   -x|--aux          List only auxiliary commands
   -w|--www          List all web pages
   -f|--full         List full set of commands (including auxiliary
                     and unsupported "test" commands), options,
                     settings, and web pages
   -e|--everything   List all help on all topics

These options can be used when TOPIC is present:

   -h|--html         Format output as HTML rather than plain text
   -c|--commands     Restrict TOPIC search to commands
```

As you can see from the description, `fossil help -a` doesn't even list all possibilities. With the `-f` option, you get even more!

7.2.2 add

The add command is used to add files into a repository. It is recursive and will pull in all files in subdirectories of the current. Fossil will not overwrite any of the files already present in the repository so it is safe to add all the files at any time. Only new files will be added.

```
$ fossil help add
Usage: fossil add ?OPTIONS? FILE1 ?FILE2 ...?

Make arrangements to add one or more files or directories to the
current checkout at the next commit.

When adding files or directories recursively, filenames that begin
with "." are excluded by default.  To include such files, add
the "--dotfiles" option to the command-line.
```

```
The --ignore and --clean options are comma-separated lists of glob patterns
for files to be excluded.  Example:  '*.o,*.obj,*.exe'  If the --ignore
option does not appear on the command line then the "ignore-glob" setting
is used.  If the --clean option does not appear on the command line then
the "clean-glob" setting is used.

If files are attempted to be added explicitly on the command line which
match "ignore-glob", a confirmation is asked first. This can be prevented
using the -f|--force option.

The --case-sensitive option determines whether or not filenames should
be treated case sensitive or not. If the option is not given, the default
depends on the global setting, or the operating system default, if not set.

Options:

   --case-sensitive BOOL   Override the case-sensitive setting.
   --dotfiles              include files beginning with a dot (".")
   -f|--force              Add files without prompting
   --ignore CSG            Ignore unmanaged files matching patterns from
                           the Comma Separated Glob (CSG) pattern list
   --clean CSG             Also ignore files matching patterns from
                           the Comma Separated Glob (CSG) list
   --reset                 Reset the ADDED state of a checkout, such
                           that all newly-added (but not yet committed)
                           files are no longer added. No flags other
                           than --verbose and --dry-run may be used
                           with --reset.

The following options are only valid with --reset:
   -v|--verbose            Outputs information about each --reset file.
   -n|--dry-run            Display instead of run actions.

See also: addremove, rm
```

Typing:

```
fossil add .
```

will add all files in the current directory and subdirectories.

Note none of these files are put in the repository untill a commit is done.

7.2.3 rm or del

The rm command is used to remove files from the repository. The file is not deleted from the file system but it will be dropped from the repository on the

next commit. This file will still be available in earlier versions of the repository but not in later ones.

```
$ fossil help rm
Usage: fossil rm|delete|forget FILE1 ?FILE2 ...?

Remove one or more files or directories from the repository.

The 'rm' and 'delete' commands do NOT normally remove the files from
disk.  They just mark the files as no longer being part of the project.
In other words, future changes to the named files will not be versioned.
However, the default behavior of this command may be overridden via the
command line options listed below and/or the 'mv-rm-files' setting.

The 'forget' command never removes files from disk, even when the command
line options and/or the 'mv-rm-files' setting would otherwise require it
to do so.

WARNING: If the "--hard" option is specified -OR- the "mv-rm-files"
         setting is non-zero, files WILL BE removed from disk as well.
         This does NOT apply to the 'forget' command.

Options:
  --soft                  Skip removing files from the checkout.
                          This supersedes the --hard option.
  --hard                  Remove files from the checkout.
  --case-sensitive BOOL   Override the case-sensitive setting.
  -n|--dry-run            If given, display instead of run actions.
  --reset                 Reset the DELETED state of a checkout, such
                          that all newly-rm'd (but not yet committed)
                          files are no longer removed. No flags other
                          than --verbose or --dry-run may be used with
                          --reset.
  --verbose|-v            Outputs information about each --reset file.
                          Only usable with --reset.

See also: addremove, add
```

You can delete groups of files by using wild-cards in their names. Thus if I had a group of files like com_tr.c, com_rx.c and com_mgmt.c I could remove them all with:

```
fossil rm com_*.c
```

By running a “fossil status” you can see what files will be deleted on the next commit.

7.2.4 rename or mv

This command is used to rename a file in the repository. This does not rename files on disk so is usually used after you have renamed files on the disk then want to change this in the repository.

```
$ fossil help rename
Usage: fossil mv|rename OLDNAME NEWNAME
   or: fossil mv|rename OLDNAME... DIR

Move or rename one or more files or directories within the repository tree.
You can either rename a file or directory or move it to another subdirectory

The 'mv' command does NOT normally rename or move the files on disk.
This command merely records the fact that file names have changed so
that appropriate notations can be made at the next commit.
However, the default behavior of this command may be overridden via
command line options listed below and/or the 'mv-rm-files' setting.

The 'rename' command never renames or moves files on disk, even when the
command line options and/or the 'mv-rm-files' setting would otherwise
require it to do so.

WARNING: If the "--hard" option is specified -OR- the "mv-rm-files"
         setting is non-zero, files WILL BE renamed or moved on disk
         as well.  This does NOT apply to the 'rename' command.

Options:
  --soft                    Skip moving files within the checkout.
                            This supersedes the --hard option.
  --hard                    Move files within the checkout.
  --case-sensitive BOOL     Override the case-sensitive setting.
  -n|--dry-run              If given, display instead of run actions.

See also: changes, status
```

Just like add or rm you can use wild cards in the names and rename groups of files. Like them “fossil status” will show you the current state.

7.2.5 status

The status command is used to show you the current state of your files relative to the repository. It will show you files added, deleted, and changed. This is only the condition of files that are already in the repository or under control of fossil. It also shows from where in the timeline you are checked out and where your repository is kept.

```
$ fossil help status
Usage: fossil changes|status ?OPTIONS? ?PATHS ...?

Report the change status of files in the current checkout.  If one or
more PATHS are specified, only changes among the named files and
directories are reported.  Directories are searched recursively.

... [snipped]

See also: extras, ls
```

```
$ fossil status
repository:   /home/chris/repos/fossilbook.fossil
local-root:   /home/chris/fossilbook/
config-db:    /home/chris/.config/fossil.db
checkout:     516e309ea1c86132e3a9be313e8331ec91f255ee 2020-11-21 16:26:12 UTC
parent:       030a38fe4b45b7088faa635982037d2eb05143b6 2020-11-21 16:23:34 UTC
tags:         trunk
comment:      Use .md file extension in book.md links. (user: chris)
EDITED     content/introduction.md
EDITED     content/rights.md
```

The listing above shows where my cloned copy of the repository is kept, where I am working, and the tags show me that I am checked out of the trunk. Finally it shows the status of the files: I am working on two of them.

7.2.6 changes

This lists the changed files like status but shows other information that status does not.

```
$ fossil help changes
Usage: fossil changes|status ?OPTIONS? ?PATHS ...?

Report the change status of files in the current checkout.  If one or
more PATHS are specified, only changes among the named files and
directories are reported.  Directories are searched recursively.

The status command is similar to the changes command, except it lacks
several of the options supported by changes and it has its own header
and footer information.  The header information is a subset of that
shown by the info command, and the footer shows if there are any forks.
Change type classification is always enabled for the status command.

Each line of output is the name of a changed file, with paths shown
according to the "relative-paths" setting, unless overridden by the
--abs-paths or --rel-paths options.
```

```
By default, all changed files are selected for display.  This behavior
can be overridden by using one or more filter options (listed below),
in which case only files with the specified change type(s) are shown.
As a special case, the --no-merge option does not inhibit this default.
This default shows exactly the set of changes that would be checked
in by the commit command.

If no filter options are used, or if the --merge option is used, the
artifact hash of each merge contributor check-in version is displayed at
the end of the report.  The --no-merge option is useful to display the
default set of changed files without the merge contributors.

If change type classification is enabled, each output line starts with
a code describing the file's change type, e.g. EDITED or RENAMED.  It
is enabled by default unless exactly one change type is selected.  For
the purposes of determining the default, --changed counts as selecting
one change type.  The default can be overridden by the --classify or
--no-classify options.

--edited and --updated produce disjoint sets.  --updated shows a file
only when it is identical to that of its merge contributor, and the
change type classification is UPDATED_BY_MERGE or UPDATED_BY_INTEGRATE.
If the file had to be merged with any other changes, it is considered
to be merged or conflicted and therefore will be shown by --edited, not
--updated, with types EDITED or CONFLICT.  The --changed option can be
used to display the union of --edited and --updated.

--differ is so named because it lists all the differences between the
checked-out version and the checkout directory.  In addition to the
default changes (excluding --merge), it lists extra files which (if
ignore-glob is set correctly) may be worth adding.  Prior to doing a
commit, it is good practice to check --differ to see not only which
changes would be committed but also if any files should be added.

If both --merge and --no-merge are used, --no-merge has priority.  The
same is true of --classify and --no-classify.

The "fossil changes --extra" command is equivalent to "fossil extras".

General options:
   --abs-paths       Display absolute pathnames.
   --rel-paths       Display pathnames relative to the current working
                     directory.
   --hash            Verify file status using hashing rather than
                     relying on file mtimes.
```

```
   --case-sensitive BOOL  Override case-sensitive setting.
   --dotfiles        Include unmanaged files beginning with a dot.
   --ignore <CSG>    Ignore unmanaged files matching CSG glob patterns.

Options specific to the changes command:
   --header          Identify the repository if report is non-empty.
   -v|--verbose      Say "(none)" if the change report is empty.
   --classify        Start each line with the file's change type.
   --no-classify     Do not print file change types.

Filter options:
   --edited          Display edited, merged, and conflicted files.
   --updated         Display files updated by merge/integrate.
   --changed         Combination of the above two options.
   --missing         Display missing files.
   --added           Display added files.
   --deleted         Display deleted files.
   --renamed         Display renamed files.
   --conflict        Display files having merge conflicts.
   --meta            Display files with metadata changes.
   --unchanged       Display unchanged files.
   --all             Display all managed files, i.e. all of the above.
   --extra           Display unmanaged files.
   --differ          Display modified and extra files.
   --merge           Display merge contributors.
   --no-merge        Do not display merge contributors.

See also: extras, ls
```

7.2.7 extras

The extras command is used to find files you have added to your working directory but are not yet under Fossil control. This is important because if you move your working directory or others attempt to use the repository they won't have these files.

```
$ fossil help extras
Usage: fossil extras ?OPTIONS? ?PATH1 ...?

Print a list of all files in the source tree that are not part of the
current checkout. See also the "clean" command. If paths are specified,
only files in the given directories will be listed.

Files and subdirectories whose names begin with "." are normally
ignored but can be included by adding the --dotfiles option.
```

```
Files whose names match any of the glob patterns in the "ignore-glob"
setting are ignored. This setting can be overridden by the --ignore
option, whose CSG argument is a comma-separated list of glob patterns.

Pathnames are displayed according to the "relative-paths" setting,
unless overridden by the --abs-paths or --rel-paths options.

Options:
   --abs-paths             Display absolute pathnames.
   --case-sensitive BOOL   Override case-sensitive setting
   --dotfiles              Include files beginning with a dot (".")
   --header                Identify the repository if there are extras
   --ignore CSG            Ignore files matching patterns from the argument
   --rel-paths             Display pathnames relative to the current working
                           directory.

See also: changes, clean, status
```

The –dotfiles option shows you any files starting with "." that are not under Fossil control. This would be important if you need those files in your repository. The last option –ignore allows you to ignore certain files you know don't belong in the repository. In the earlier version of this repository, when Lyx was the editor of choice, there was a file called fossilbook.lyx~ that is a LyX backup file that was not wanted, as it is temporary. So one can say

```
fossil extra -{}-ignore \*.lyx~
```

and only get:

```
$ fossil extra -{}-ignore \*.lyx\~

image/basic/help.png
```

instead of:

```
$ fossil extra

image/basic/help1.png
fossilbook.lyx~
```

7.2.8 revert

The revert command is used to take a file back to the value in the repository. This is useful when you make a error in editing or other mistake.

```
$ fossil help revert
Usage: fossil revert ?OPTIONS? ?FILE ...?

Revert to the current repository version of FILE, or to
```

```
the baseline VERSION specified with -r flag.

If FILE was part of a rename operation, both the original file
and the renamed file are reverted.

Using a directory name for any of the FILE arguments is the same
as using every subdirectory and file beneath that directory.

Revert all files if no file name is provided.

If a file is reverted accidentally, it can be restored using
the "fossil undo" command.

Options:
  -r|--revision VERSION    Revert given FILE(s) back to given
                           VERSION

See also: redo, undo, checkout, update
```

With no parameters it will revert the file to the current revision, see Figure [fig:status-run]. The -r option allows you to pick any revision from the time line.

7.2.9 update

The update option will update a file or files to match the repository. With multiple users it should be done before you start working on any files. This ensures you have the latest version of all the files.

```
$ fossil help update
Usage: fossil update ?OPTIONS? ?VERSION? ?FILES...?

Change the version of the current checkout to VERSION.  Any
uncommitted changes are retained and applied to the new checkout.

The VERSION argument can be a specific version or tag or branch
name.  If the VERSION argument is omitted, then the leaf of the
subtree that begins at the current version is used, if there is
only a single leaf.  VERSION can also be "current" to select the
leaf of the current version or "latest" to select the most recent
check-in.

If one or more FILES are listed after the VERSION then only the
named files are candidates to be updated, and any updates to them
will be treated as edits to the current version. Using a directory
name for one of the FILES arguments is the same as using every
subdirectory and file beneath that directory.
```

```
If FILES is omitted, all files in the current checkout are subject
to being updated and the version of the current checkout is changed
to VERSION. Any uncommitted changes are retained and applied to the
new checkout.

The -n or --dry-run option causes this command to do a "dry run".
It prints out what would have happened but does not actually make
any changes to the current checkout or the repository.

The -v or --verbose option prints status information about
unchanged files in addition to those file that actually do change.

Options:
  --case-sensitive BOOL  Override case-sensitive setting
  --debug                Print debug information on stdout
  --latest               Acceptable in place of VERSION, update to
                         latest version
  --force-missing        Force update if missing content after sync
  -n|--dry-run           If given, display instead of run actions
  -v|--verbose           Print status information about all files
  -W|--width WIDTH       Width of lines (default is to auto-detect).
                         Must be more than 20 or 0 (= no limit,
                         resulting in a single line per entry).
  --setmtime             Set timestamps of all files to match their
                         SCM-side times (the timestamp of the last
                         checkin which modified them).
 -K|--keep-merge-files   On merge conflict, retain the temporary files
                         used for merging, named *-baseline, *-original,
                         and *-merge.

See also: revert
```

Update has a number of options, first you can tie the update to a particular version, if not picked then it just uses the latest. Second it can work on a single files or many files at once. That is you could say

```
$ fossil update \*.c
```

and it would update all C files.

Since this is a rather large set of changes it has a special “dry run” mode. If you add -n on the command it will just print out what will be done but not do it. This is very useful to do this trial if you are unsure what might happen. The -v command (which can be used with -n or alone) prints out the action for each file even if it does nothing.

The `update` command can be used to do a kind of cherrypicking. When you have a branch (or just some other version) and want to only merge a single file of that branch/version into the current checkout, you can do `fossil update other_branch other_file`. This will bring a copy of that other file into the current branch/version. However, this will not leave any trace in the timeline. Further, you will have to `fossil add` that file specifically yourself so it gets committed in the next `fossil commit`.

7.2.10 checkout or co

This command is similar to update.

```
$ fossil help checkout
Usage: fossil checkout ?VERSION | --latest? ?OPTIONS?
   or: fossil co ?VERSION | --latest? ?OPTIONS?

Check out a version specified on the command-line.  This command
will abort if there are edited files in the current checkout unless
the --force option appears on the command-line.  The --keep option
leaves files on disk unchanged, except the manifest and manifest.uuid
files.

The --latest flag can be used in place of VERSION to checkout the
latest version in the repository.

Options:
   --force           Ignore edited files in the current checkout
   --keep            Only update the manifest and manifest.uuid files
   --force-missing   Force checkout even if content is missing
   --setmtime        Set timestamps of all files to match their SCM-side
                     times (the timestamp of the last checkin which modified
                     them).

See also: update
```

7.2.11 undo

This is used to undo the last update, merge, or revert operation.

```
$ fossil help undo
Usage: fossil undo ?OPTIONS? ?FILENAME...?
   or: fossil redo ?OPTIONS? ?FILENAME...?

The undo command reverts the changes caused by the previous command
```

```
if the previous command is one of the following:
 * fossil update
 * fossil merge
 * fossil revert
 * fossil stash pop
 * fossil stash apply
 * fossil stash drop
 * fossil stash goto
 * fossil clean (*see note below*)

Note: The "fossil clean" command only saves state for files less than
10MiB in size and so if fossil clean deleted files larger than that,
then "fossil undo" will not recover the larger files.

If FILENAME is specified then restore the content of the named
file(s) but otherwise leave the update or merge or revert in effect.
The redo command undoes the effect of the most recent undo.

If the -n|--dry-run option is present, no changes are made and instead
the undo or redo command explains what actions the undo or redo would
have done had the -n|--dry-run been omitted.

If the most recent command is not one of those listed as undoable,
then the undo command might try to restore the state to be what it was
prior to the last undoable command, or it might be a no-op.  If in
doubt about what the undo command will do, first run it with the -n
option.

A single level of undo/redo is supported.  The undo/redo stack
is cleared by the commit and checkout commands.  Other commands may
or may not clear the undo stack.

Future versions of Fossil might add new commands to the set of commands
that are undoable.

Options:
  -n|--dry-run   do not make changes but show what would be done

See also: commit, status
```

It acts on a single file or files if specified, otherwise if no file given, it undoes all of the last changes.

7.2.12 diff

The diff command is used to produce a text listing of the difference of a file in the working directory and that same file in the repository. If you don't specify a file it will show the differences between all the changed files in the working directory vs the repository. If you use the –from and –to options you can specify which versions to check and to compare between two different versions in the repository. Not using the –to means compare with the working directory.

If you have configured an external diff program it will be used unless you use the -i option which uses the diff built into Fossil.

```
$ fossil help diff
Usage: fossil diff|gdiff ?OPTIONS? ?FILE1? ?FILE2 ...?

Show the difference between the current version of each of the FILEs
specified (as they exist on disk) and that same file as it was checked
out.  Or if the FILE arguments are omitted, show the unsaved changes
currently in the working check-out.

If the "--from VERSION" or "-r VERSION" option is used it specifies
the source check-in for the diff operation.  If not specified, the
source check-in is the base check-in for the current check-out.

If the "--to VERSION" option appears, it specifies the check-in from
which the second version of the file or files is taken.  If there is
no "--to" option then the (possibly edited) files in the current check-out
are used.

The "--checkin VERSION" option shows the changes made by
check-in VERSION relative to its primary parent.

The "-i" command-line option forces the use of the internal diff logic
rather than any external diff program that might be configured using
the "setting" command.  If no external diff program is configured, then
the "-i" option is a no-op.  The "-i" option converts "gdiff" into "diff".

The "-N" or "--new-file" option causes the complete text of added or
deleted files to be displayed.

The "--diff-binary" option enables or disables the inclusion of binary files
when using an external diff program.

The "--binary" option causes files matching the glob PATTERN to be treated
as binary when considering if they should be used with external diff program.
This option overrides the "binary-glob" setting.
```

```
Options:
  --binary PATTERN            Treat files that match the glob PATTERN
                              as binary
  --branch BRANCH             Show diff of all changes on BRANCH
  --brief                     Show filenames only
  --checkin VERSION           Show diff of all changes in VERSION
  --command PROG              External diff program. Overrides "diff-comman
  --context|-c N              Use N lines of context
  --diff-binary BOOL          Include binary files with external commands
  --exec-abs-paths            Force absolute path names on external commands
  --exec-rel-paths            Force relative path names on external commands
  --from|-r VERSION           Select VERSION as source for the diff
  --internal|-i               Use internal diff logic
  --new-file|-N               Show complete text of added and deleted files
  --numstat                   Show only the number of lines delete and added
  --side-by-side|-y           Side-by-side diff
  --strip-trailing-cr         Strip trailing CR
  --tclsh PATH                Tcl/Tk used for --tk (default: "tclsh")
  --tk                        Launch a Tcl/Tk GUI for display
  --to VERSION                Select VERSION as target for the diff
  --undo                      Diff against the "undo" buffer
  --unified                   Unified diff
  -v|--verbose                Output complete text of added or deleted files
  -w|--ignore-all-space       Ignore white space when comparing lines
  -W|--width N                Width of lines in side-by-side diff
  -Z|--ignore-trailing-space  Ignore changes to end-of-line whitespace
```

7.2.13 gdiff

This is the same as the diff command but uses (if configured) a graphical diff program you have on your system. See the settings command for details on how to set the graphical diff program.

```
$ fossil help gdiff
Usage: fossil diff|gdiff ?OPTIONS? ?FILE1? ?FILE2 ...?

Show the difference between the current version of each of the FILEs
specified (as they exist on disk) and that same file as it was checked
out.  Or if the FILE arguments are omitted, show the unsaved changes
currently in the working check-out.

If the "--from VERSION" or "-r VERSION" option is used it specifies
the source check-in for the diff operation.  If not specified, the
source check-in is the base check-in for the current check-out.

If the "--to VERSION" option appears, it specifies the check-in from
```

```
which the second version of the file or files is taken.  If there is
no "--to" option then the (possibly edited) files in the current check-out
are used.

The "--checkin VERSION" option shows the changes made by
check-in VERSION relative to its primary parent.

The "-i" command-line option forces the use of the internal diff logic
rather than any external diff program that might be configured using
the "setting" command.  If no external diff program is configured, then
the "-i" option is a no-op.  The "-i" option converts "gdiff" into "diff".

The "-N" or "--new-file" option causes the complete text of added or
deleted files to be displayed.

The "--diff-binary" option enables or disables the inclusion of binary files
when using an external diff program.

The "--binary" option causes files matching the glob PATTERN to be treated
as binary when considering if they should be used with external diff program.
This option overrides the "binary-glob" setting.

Options:
  --binary PATTERN            Treat files that match the glob PATTERN
                              as binary
  --branch BRANCH             Show diff of all changes on BRANCH
  --brief                     Show filenames only
  --checkin VERSION           Show diff of all changes in VERSION
  --command PROG              External diff program. Overrides "diff-command"
  --context|-c N              Use N lines of context
  --diff-binary BOOL          Include binary files with external commands
  --exec-abs-paths            Force absolute path names on external commands
  --exec-rel-paths            Force relative path names on external commands
  --from|-r VERSION           Select VERSION as source for the diff
  --internal|-i               Use internal diff logic
  --new-file|-N               Show complete text of added and deleted files
  --numstat                   Show only the number of lines delete and added
  --side-by-side|-y           Side-by-side diff
  --strip-trailing-cr         Strip trailing CR
  --tclsh PATH                Tcl/Tk used for --tk (default: "tclsh")
  --tk                        Launch a Tcl/Tk GUI for display
  --to VERSION                Select VERSION as target for the diff
  --undo                      Diff against the "undo" buffer
  --unified                   Unified diff
  -v|--verbose                Output complete text of added or deleted files
  -w|--ignore-all-space       Ignore white space when comparing lines
```

```
  -W|--width N                Width of lines in side-by-side diff
  -Z|--ignore-trailing-space  Ignore changes to end-of-line whitespace
```

7.2.14 ui

The ui command is used to start Fossil in a local webserver. The –port option is used to specify the port it uses, by default it uses 8080. It should automatically start the system's web browser and it will come up with the repository web page. If run within a working directory it will bring up the web page for that repository. If run outside the working directory you can specify the repository on the command line.

```
$ fossil help ui
Usage: fossil server ?OPTIONS? ?REPOSITORY?
   or: fossil ui ?OPTIONS? ?REPOSITORY?

Open a socket and begin listening and responding to HTTP requests on
TCP port 8080, or on any other TCP port defined by the -P or
--port option.  The optional argument is the name of the repository.
The repository argument may be omitted if the working directory is
within an open checkout.

The "ui" command automatically starts a web browser after initializing
the web server.  The "ui" command also binds to 127.0.0.1 and so will
only process HTTP traffic from the local machine.

The REPOSITORY can be a directory (aka folder) that contains one or
more repositories with names ending in ".fossil".  In this case, a
prefix of the URL pathname is used to search the directory for an
appropriate repository.  To thwart mischief, the pathname in the URL must
contain only alphanumerics, "_", "/", "-", and ".", and no "-" may
occur after "/", and every "." must be surrounded on both sides by
alphanumerics.  Any pathname that does not satisfy these constraints
results in a 404 error.  Files in REPOSITORY that match the comma-separated
list of glob patterns given by --files and that have known suffixes
such as ".txt" or ".html" or ".jpeg" and do not match the pattern
"*.fossil*" will be served as static content.  With the "ui" command,
the REPOSITORY can only be a directory if the --notfound option is
also present.

For the special case REPOSITORY name of "/", the list global configuration
database is consulted for a list of all known repositories.  The --repolist
option is implied by this special case.  See also the "fossil all ui"
command.

By default, the "ui" command provides full administrative access without
```

```
having to log in.  This can be disabled by turning off the "localauth"
setting.  Automatic login for the "server" command is available if the
--localauth option is present and the "localauth" setting is off and the
connection is from localhost.  The "ui" command also enables --repolist
by default.

Options:
  --baseurl URL       Use URL as the base (useful for reverse proxies)
  --create            Create a new REPOSITORY if it does not already exist
  --extroot DIR       Document root for the /ext extension mechanism
  --files GLOBLIST    Comma-separated list of glob patterns for static files
  --localauth         enable automatic login for requests from localhost
  --localhost         listen on 127.0.0.1 only (always true for "ui")
  --https             Indicates that the input is coming through a reverse
                      proxy that has already translated HTTPS into HTTP.
  --jsmode MODE       Determine how JavaScript is delivered with pages.
                      Mode can be one of:
                         inline       All JavaScript is inserted inline at
                                      the end of the HTML file.
                         separate     Separate HTTP requests are made for
                                      each JavaScript file.
                         bundled      One single separate HTTP fetches all
                                      JavaScript concatenated together.
                      Depending on the needs of any given page, inline
                      and bundled modes might result in a single
                      amalgamated script or several, but both approaches
                      result in fewer HTTP requests than the separate mode.
  --max-latency N     Do not let any single HTTP request run for more than N
                      seconds (only works on unix)
  --nocompress        Do not compress HTTP replies
  --nojail            Drop root privileges but do not enter the chroot jail
  --nossl             signal that no SSL connections are available (Always
                      set by default for the "ui" command)
  --notfound URL      Redirect
  --page PAGE         Start "ui" on PAGE.  ex: --page "timeline?y=ci"
  -P|--port TCPPORT   listen to request on port TCPPORT
  --th-trace          trace TH1 execution (for debugging purposes)
  --repolist          If REPOSITORY is dir, URL "/" lists repos.
  --scgi              Accept SCGI rather than HTTP
  --skin LABEL        Use override skin LABEL
  --usepidkey         Use saved encryption key from parent process.  This is
                      only necessary when using SEE on Windows.

See also: cgi, http, winsrv
```

7.2.15 server

This is a more powerful version of the ui command. This allows you to have multiple repositories supported by a single running Fossil webserver. This way you start the server and instead of a paricular repository you specify a directory where a number of repositories reside (all having the extension .fossil) then you can open and use any of them.

```
$ fossil help server
Usage: fossil server ?OPTIONS? ?REPOSITORY?
   or: fossil ui ?OPTIONS? ?REPOSITORY?

Open a socket and begin listening and responding to HTTP requests on
TCP port 8080, or on any other TCP port defined by the -P or
--port option.  The optional argument is the name of the repository.
The repository argument may be omitted if the working directory is
within an open checkout.

The "ui" command automatically starts a web browser after initializing
the web server.  The "ui" command also binds to 127.0.0.1 and so will
only process HTTP traffic from the local machine.

The REPOSITORY can be a directory (aka folder) that contains one or
more repositories with names ending in ".fossil".  In this case, a
prefix of the URL pathname is used to search the directory for an
appropriate repository.  To thwart mischief, the pathname in the URL must
contain only alphanumerics, "_", "/", "-", and ".", and no "-" may
occur after "/", and every "." must be surrounded on both sides by
alphanumerics.  Any pathname that does not satisfy these constraints
results in a 404 error.  Files in REPOSITORY that match the comma-separated
list of glob patterns given by --files and that have known suffixes
such as ".txt" or ".html" or ".jpeg" and do not match the pattern
"*.fossil*" will be served as static content.  With the "ui" command,
the REPOSITORY can only be a directory if the --notfound option is
also present.

For the special case REPOSITORY name of "/", the list global configuration
database is consulted for a list of all known repositories.  The --repolist
option is implied by this special case.  See also the "fossil all ui"
command.

By default, the "ui" command provides full administrative access without
having to log in.  This can be disabled by turning off the "localauth"
setting.  Automatic login for the "server" command is available if the
--localauth option is present and the "localauth" setting is off and the
connection is from localhost.  The "ui" command also enables --repolist
```

```
by default.

Options:
  --baseurl URL       Use URL as the base (useful for reverse proxies)
  --create            Create a new REPOSITORY if it does not already exist
  --extroot DIR       Document root for the /ext extension mechanism
  --files GLOBLIST    Comma-separated list of glob patterns for static files
  --localauth         enable automatic login for requests from localhost
  --localhost         listen on 127.0.0.1 only (always true for "ui")
  --https             Indicates that the input is coming through a reverse
                      proxy that has already translated HTTPS into HTTP.
  --jsmode MODE       Determine how JavaScript is delivered with pages.
                      Mode can be one of:
                         inline       All JavaScript is inserted inline at
                                      the end of the HTML file.
                         separate     Separate HTTP requests are made for
                                      each JavaScript file.
                         bundled      One single separate HTTP fetches all
                                      JavaScript concatenated together.
                      Depending on the needs of any given page, inline
                      and bundled modes might result in a single
                      amalgamated script or several, but both approaches
                      result in fewer HTTP requests than the separate mode.
  --max-latency N     Do not let any single HTTP request run for more than N
                      seconds (only works on unix)
  --nocompress        Do not compress HTTP replies
  --nojail            Drop root privileges but do not enter the chroot jail
  --nossl             signal that no SSL connections are available (Always
                      set by default for the "ui" command)
  --notfound URL      Redirect
  --page PAGE         Start "ui" on PAGE.  ex: --page "timeline?y=ci"
  -P|--port TCPPORT   listen to request on port TCPPORT
  --th-trace          trace TH1 execution (for debugging purposes)
  --repolist          If REPOSITORY is dir, URL "/" lists repos.
  --scgi              Accept SCGI rather than HTTP
  --skin LABEL        Use override skin LABEL
  --usepidkey         Use saved encryption key from parent process.  This is
                      only necessary when using SEE on Windows.

See also: cgi, http, winsrv
```

7.2.16 commit or ci

This is the command used to put the current changes in the working directory into the repository, giving this a new version and updating the timeline.

```
$ fossil help commit
Usage: fossil commit ?OPTIONS? ?FILE...?
   or: fossil ci ?OPTIONS? ?FILE...?

Create a new version containing all of the changes in the current
checkout.  You will be prompted to enter a check-in comment unless
the comment has been specified on the command-line using "-m" or a
file containing the comment using -M.  The editor defined in the
"editor" fossil option (see fossil help set) will be used, or from
the "VISUAL" or "EDITOR" environment variables (in that order) if
no editor is set.

All files that have changed will be committed unless some subset of
files is specified on the command line.

The --branch option followed by a branch name causes the new
check-in to be placed in a newly-created branch with the name
passed to the --branch option.

Use the --branchcolor option followed by a color name (ex:
'#ffc0c0') to specify the background color of entries in the new
branch when shown in the web timeline interface.  The use of
the --branchcolor option is not recommended.  Instead, let Fossil
choose the branch color automatically.

The --bgcolor option works like --branchcolor but only sets the
background color for a single check-in.  Subsequent check-ins revert
to the default color.

A check-in is not permitted to fork unless the --allow-fork option
appears.  An empty check-in (i.e. with nothing changed) is not
allowed unless the --allow-empty option appears.  A check-in may not
be older than its ancestor unless the --allow-older option appears.
If any of files in the check-in appear to contain unresolved merge
conflicts, the check-in will not be allowed unless the
--allow-conflict option is present.  In addition, the entire
check-in process may be aborted if a file contains content that
appears to be binary, Unicode text, or text with CR/LF line endings
unless the interactive user chooses to proceed.  If there is no
interactive user or these warnings should be skipped for some other
reason, the --no-warnings option may be used.  A check-in is not
allowed against a closed leaf.

If a commit message is blank, you will be prompted:
("continue (y/N)?") to confirm you really want to commit with a
blank commit message.  The default value is "N", do not commit.
```

```
The --private option creates a private check-in that is never synced.
Children of private check-ins are automatically private.

The --tag option applies the symbolic tag name to the check-in.

The --hash option detects edited files by computing each file's
artifact hash rather than just checking for changes to its size or mtime.

Options:
   --allow-conflict           allow unresolved merge conflicts
   --allow-empty              allow a commit with no changes
   --allow-fork               allow the commit to fork
   --allow-older              allow a commit older than its ancestor
   --baseline                 use a baseline manifest in the commit process
   --bgcolor COLOR            apply COLOR to this one check-in only
   --branch NEW-BRANCH-NAME   check in to this new branch
   --branchcolor COLOR        apply given COLOR to the branch
   --close                    close the branch being committed
   --date-override DATETIME   DATE to use instead of 'now'
   --delta                    use a delta manifest in the commit process
   --hash                     verify file status using hashing rather
                              than relying on file mtimes
   --integrate                close all merged-in branches
   -m|--comment COMMENT-TEXT  use COMMENT-TEXT as commit comment
   -M|--message-file FILE     read the commit comment from given file
   --mimetype MIMETYPE        mimetype of check-in comment
   -n|--dry-run               If given, display instead of run actions
   --no-prompt                This option disables prompting the user for
                              input and assumes an answer of 'No' for every
                              question.
   --no-warnings              omit all warnings about file contents
   --no-verify                do not run before-commit hooks
   --nosign                   do not attempt to sign this commit with gpg
   --override-lock            allow a check-in even though parent is locked
   --private                  do not sync changes and their descendants
   --tag TAG-NAME             assign given tag TAG-NAME to the check-in
   --trace                    debug tracing.
   --user-override USER       USER to use instead of the current default

DATETIME may be "now" or "YYYY-MM-DDTHH:MM:SS.SSS". If in
year-month-day form, it may be truncated, the "T" may be replaced by
a space, and it may also name a timezone offset from UTC as "-HH:MM"
(westward) or "+HH:MM" (eastward). Either no timezone suffix or "Z"
means UTC.
```

```
See also: branch, changes, update, extras, sync
```

It's a very good idea to always put a comment (-comment or -m) text on any commit. This way you get documentation in the timeline.

7.3 Maintenance

These commands you will probably use less often since the actions they perform are not needed in normal operation. You will have to use them and referring here or to fossil help will probably be required before use. Some of them like new or clone are only needed when you start a repository. Others like rebuild or reconstruct are only needed to fix or update a repository.

7.3.1 new

This command is used to create a new repository.

```
$ fossil help new
Usage: fossil new ?OPTIONS? FILENAME
   or: fossil init ?OPTIONS? FILENAME

Create a repository for a new project in the file named FILENAME.
This command is distinct from "clone".  The "clone" command makes
a copy of an existing project.  This command starts a new project.

By default, your current login name is used to create the default
admin user. This can be overridden using the -A|--admin-user
parameter.

By default, all settings will be initialized to their default values.
This can be overridden using the --template parameter to specify a
repository file from which to copy the initial settings.  When a template
repository is used, almost all of the settings accessible from the setup
page, either directly or indirectly, will be copied.  Normal users and
their associated permissions will not be copied; however, the system
default users "anonymous", "nobody", "reader", "developer", and their
associated permissions will be copied.

Options:
   --template      FILE         Copy settings from repository file
   --admin-user|-A USERNAME     Select given USERNAME as admin user
   --date-override DATETIME     Use DATETIME as time of the initial check-in
   --sha1                       Use an initial hash policy of "sha1"

DATETIME may be "now" or "YYYY-MM-DDTHH:MM:SS.SSS". If in
year-month-day form, it may be truncated, the "T" may be replaced by
```

```
a space, and it may also name a timezone offset from UTC as "-HH:MM"
(westward) or "+HH:MM" (eastward). Either no timezone suffix or "Z"
means UTC.

See also: clone
```

The file name specifies the new repository name. The options provided allow you to specify the admin user name if you want it to be different than your current login and the starting date if you want it to be different than now.

7.3.2 clone

The clone command is used to create your own local version of the master repository. If you are supporting multiple users via a network accessible version of the original repository (see Section[sub:Server-Setup]), then this command will copy that repository to your machine. Also it will make a link between your copy and the master, so that changes made in your copy will be propagated to the master.

```
$ fossil help clone
Usage: fossil clone ?OPTIONS? URI ?FILENAME?

Make a clone of a repository specified by URI in the local
file named FILENAME.  If FILENAME is omitted, then an appropriate
filename is deduced from last element of the path in the URL.

URI may be one of the following forms ([...] denotes optional elements):

 * HTTP/HTTPS protocol:

     http[s]://[userid[:password]@]host[:port][/path]

 * SSH protocol:

     ssh://[userid@]host[:port]/path/to/repo.fossil[?fossil=path/fossil.exe]

 * Filesystem:

     [file://]path/to/repo.fossil

For ssh and filesystem, path must have an extra leading
'/' to use an absolute path.

Use %HH escapes for special characters in the userid and
password.  For example "%40" in place of "@", "%2f" in place
of "/", and "%3a" in place of ":".
```

```
Note that in Fossil (in contrast to some other DVCSes) a repository
is distinct from a checkout.  Cloning a repository is not the same thing
as opening a repository.  This command always clones the repository.  This
command might also open the repository, but only if the --no-open option
is omitted and either the --workdir option is included or the FILENAME
argument is omitted.  Use the separate open command to open a
repository that was previously cloned and already exists on the
local machine.

By default, the current login name is used to create the default
admin user for the new clone. This can be overridden using
the -A|--admin-user parameter.

Options:
   --admin-user|-A USERNAME   Make USERNAME the administrator
   --httpauth|-B USER:PASS    Add HTTP Basic Authorization to requests
   --nocompress               Omit extra delta compression
   --no-open                  Clone only.  Do not open a check-out.
   --once                     Don't remember the URI.
   --private                  Also clone private branches
   --save-http-password       Remember the HTTP password without asking
   --ssh-command|-c SSH       Use SSH as the "ssh" command
   --ssl-identity FILENAME    Use the SSL identity if requested by the serve
   -u|--unversioned           Also sync unversioned content
   -v|--verbose               Show more statistics in output
   --workdir DIR              Also open a checkout in DIR

See also: init, open
```

As with **create**, you can specify the admin user for this clone with an option. The URL for the master repository is of the form:

```
https://user:password@domain
```

Where user and password are for a valid user of the selected repository. It is best to check the path with a browser before doing the clone. Make sure you can reach it, for example the repository for this book is:

```
https://www.fossil-scm.org/schimpf-book/home
```

Putting that into a browser should get you the home page for this book. (See Figure [fig:Web-access-to]). After you have verified that, then running the clone command should work.

Don't forget (as I always do) to put in the file name for the local repository, (see FILENAME above)

7.3.3 open

The open command is used to copy the files in a repository to a working directory. Doing this allows you to build or modify the product. The command also links this working directory to the repository so commits will go into the repository.

```
$ fossil help open
Usage: fossil open REPOSITORY ?VERSION? ?OPTIONS?

Open a new connection to the repository name REPOSITORY.  A checkout
for the repository is created with its root at the current working
directory, or in DIR if the "--workdir DIR" is used.  If VERSION is
specified then that version is checked out.  Otherwise the most recent
check-in on the main branch (usually "trunk") is used.

REPOSITORY can be the filename for a repository that already exists on the
local machine or it can be a URI for a remote repository.  If REPOSITORY
is a URI in one of the formats recognized by the clone command, then
remote repo is first cloned, then the clone is opened. The clone will be
stored in the current directory, or in DIR if the "--repodir DIR" option
is used. The name of the clone will be taken from the last term of the URI.
For "http:" and "https:" URIs, you can append an extra term to the end of
the URI to get any repository name you like. For example:

    fossil open https://fossil-scm.org/home/new-name

The base URI for cloning is "https://fossil-scm.org/home".  The extra
"new-name" term means that the cloned repository will be called
"new-name.fossil".

Options:
  --empty           Initialize checkout as being empty, but still connected
                    with the local repository. If you commit this checkout,
                    it will become a new "initial" commit in the repository.
  -f|--force        Continue with the open even if the working directory is
                    not empty.
  --force-missing   Force opening a repository with missing content
  --keep            Only modify the manifest and manifest.uuid files
  --nested          Allow opening a repository inside an opened checkout
  --repodir DIR     If REPOSITORY is a URI that will be cloned, store
                    the clone in DIR rather than in "."
  --setmtime        Set timestamps of all files to match their SCM-side
                    times (the timestamp of the last checkin which modified
                    them).
  --workdir DIR     Use DIR as the working directory instead of ".". The DIR
```

```
                    directory is created if it does not exist.

See also: close, clone
```

If you have multiple users or have a branched repository then it is probably wise to specify the particular version you want. When you run this it will create all the files and directories in the repository in your work area. In addition the files *FOSSIL*, manifiest and manifest.uuid will be created by Fossil.

You can have multiple open connections to the same repository at the same time. Every open connection just needs to be in its own separate working directory. This is very useul when you have many branches and do not want to commit your work just to switch over to another branch and continue or check something there. Just create a new open connection in another directory and you can `cd` between them and work on multiple branches at the same time. If you want to know where all the open checkouts of your repository are located, use `fossil info -v`.

7.3.4 close

This is the opposite of open, in that it breaks the connection between this working directory and the Fossil repository.

```
$ fossil help close
Usage: fossil close ?OPTIONS?

The opposite of "open".  Close the current database connection.
Require a -f or --force flag if there are unsaved changes in the
current check-out or if there is non-empty stash.

Options:
  --force|-f  necessary to close a check out with uncommitted changes

See also: open
```

This is useful if you need to abandon the current working directory. Fossil will not let you do this if there are changes between the current directory and the repository. With the force flag you can explicitly cut the connection even if there are changes.

7.3.5 version

This command is used to show the current version of fossil.

```
$ fossil help version
Usage: fossil version ?-verbose|-v?

Print the source code version number for the fossil executable.
```

```
If the verbose option is specified, additional details will
be output about what optional features this binary was compiled
with


$ fossil version
This is fossil version 2.14 [939a13d94f] 2020-11-20 19:28:05 UTC
```

The above figure shows the help and example of running the command. When you have problems with fossil it is very important to have this version information. You can then inquire of the Fossil news group about this problem and with the version information they can easily tell you if the problem is fixed already or is new.

7.3.6 rebuild

If you update your copy of Fossil you will want to run this command against all the repositories you have. This will automatically update them to the new version of Fossil.

```
$ fossil help rebuild
Usage: fossil rebuild ?REPOSITORY? ?OPTIONS?

Reconstruct the named repository database from the core
records.  Run this command after updating the fossil
executable in a way that changes the database schema.

Options:
  --analyze         Run ANALYZE on the database after rebuilding
  --cluster         Compute clusters for unclustered artifacts
  --compress        Strive to make the database as small as possible
  --compress-only   Skip the rebuilding step. Do --compress only
  --deanalyze       Remove ANALYZE tables from the database
  --force           Force the rebuild to complete even if errors are seen
  --ifneeded        Only do the rebuild if it would change the schema version
  --index           Always add in the full-text search index
  --noverify        Skip the verification of changes to the BLOB table
  --noindex         Always omit the full-text search index
  --pagesize N      Set the database pagesize to N. (512..65536 and power of 2)
  --quiet           Only show output if there are errors
  --randomize       Scan artifacts in a random order
  --stats           Show artifact statistics after rebuilding
  --vacuum          Run VACUUM on the database after rebuilding
  --wal             Set Write-Ahead-Log journalling mode on the database
```

7.3.7 all

This command is actually a modifier and when used before certain commands will run them on all the repositories.

```
$ fossil help all
Usage: fossil all SUBCOMMAND ...

The ~/.fossil file records the location of all repositories for a
user.  This command performs certain operations on all repositories
that can be useful before or after a period of disconnected operation.

On Win32 systems, the file is named "_fossil" and is located in
%LOCALAPPDATA%, %APPDATA% or %HOMEPATH%.

Available operations are:

   backup      Backup all repositories.  The argument must the name of
               a directory into which all backup repositories are written.

   cache       Manages the cache used for potentially expensive web
               pages.  Any additional arguments are passed on verbatim
               to the cache command.

   changes     Shows all local checkouts that have uncommitted changes.
               This operation has no additional options.

   clean       Delete all "extra" files in all local checkouts.  Extreme
               caution should be exercised with this command because its
               effects cannot be undone.  Use of the --dry-run option to
               carefully review the local checkouts to be operated upon
               and the --whatif option to carefully review the files to
               be deleted beforehand is highly recommended.  The command
               line options supported by the clean command itself, if any
               are present, are passed along verbatim.

   config      Only the "config pull AREA" command works.

   dbstat      Run the "dbstat" command on all repositories.

   extras      Shows "extra" files from all local checkouts.  The command
               line options supported by the extra command itself, if any
               are present, are passed along verbatim.

   fts-config  Run the "fts-config" command on all repositories.
```

```
   info        Run the "info" command on all repositories.

   pull        Run a "pull" operation on all repositories.  Only the
               --verbose option is supported.

   push        Run a "push" on all repositories.  Only the --verbose
               option is supported.

   rebuild     Rebuild on all repositories.  The command line options
               supported by the rebuild command itself, if any are
               present, are passed along verbatim.  The --force and
               --randomize options are not supported.

   sync        Run a "sync" on all repositories.  Only the --verbose
               and --unversioned options are supported.

   set|unset   Run the "setting", "set", or "unset" commands on all
               repositories.  These command are particularly useful in
               conjunction with the "max-loadavg" setting which cannot
               otherwise be set globally.

   server      Run the "ui" or "server" commands on all repositories.
   ui          The root URI gives a listing of all repos.


In addition, the following maintenance operations are supported:

   add         Add all the repositories named to the set of repositories
               tracked by Fossil.  Normally Fossil is able to keep up with
               this list by itself, but sometimes it can benefit from this
               hint if you rename repositories.

   ignore      Arguments are repositories that should be ignored by
               subsequent clean, extras, list, pull, push, rebuild, and
               sync operations.  The -c|--ckout option causes the listed
               local checkouts to be ignored instead.

   list | ls   Display the location of all repositories.  The -c|--ckout
               option causes all local checkouts to be listed instead.

Repositories are automatically added to the set of known repositories
when one of the following commands are run against the repository:
clone, info, pull, push, or sync.  Even previously ignored repositories
are added back to the list of repositories by these commands.

Options:
```

```
  --dry-run         If given, display instead of run actions.
  --showfile        Show the repository or checkout being operated upon.
  --stop-on-error   Halt immediately if any subprocess fails.
```

7.3.8 push

This command will push changes in the local repository to the master or remote repository.

```
$ fossil help push
Usage: fossil push ?URL? ?options?

Push all sharable changes from the local repository to a remote
repository.  Sharable changes include public check-ins, edits to
wiki pages, tickets, and tech-notes, as well as forum content.  Use
--private to also push private branches.  Use the "configuration
push" command to push website configuration details.

If URL is not specified, then the URL from the most recent clone, push,
pull, remote, or sync command is used.  See "fossil help clone" for
details on the URL formats.

Options:

  -B|--httpauth USER:PASS    Credentials for the simple HTTP auth protocol,
                             if required by the remote website
  --ipv4                     Use only IPv4, not IPv6
  --once                     Do not remember URL for subsequent syncs
  --proxy PROXY              Use the specified HTTP proxy
  --private                  Push private branches too
  -R|--repository REPO       Local repository to push from
  --ssl-identity FILE        Local SSL credentials, if requested by remote
  --ssh-command SSH          Use SSH as the "ssh" command
  -v|--verbose               Additional (debugging) output
  --verily                   Exchange extra information with the remote
                             to ensure no content is overlooked

See also: clone, config, pull, remote, sync
```

7.3.9 pull

This command will copy changes from the remote repository to the local repository. You could then use update to apply these changes to checked out files.

```
$ fossil help pull
Usage: fossil pull ?URL? ?options?
```

```
Pull all sharable changes from a remote repository into the local
repository.  Sharable changes include public check-ins, edits to
wiki pages, tickets, and tech-notes, as well as forum content.  Add
the --private option to pull private branches.  Use the
"configuration pull" command to pull website configuration details.

If URL is not specified, then the URL from the most recent clone, push,
pull, remote, or sync command is used.  See "fossil help clone" for
details on the URL formats.

Options:

  -B|--httpauth USER:PASS    Credentials for the simple HTTP auth protocol,
                             if required by the remote website
  --from-parent-project      Pull content from the parent project
  --ipv4                     Use only IPv4, not IPv6
  --once                     Do not remember URL for subsequent syncs
  --private                  Pull private branches too
  --project-code CODE        Use CODE as the project code
  --proxy PROXY              Use the specified HTTP proxy
  -R|--repository REPO       Local repository to pull into
  --ssl-identity FILE        Local SSL credentials, if requested by remote
  --ssh-command SSH          Use SSH as the "ssh" command
  -v|--verbose               Additional (debugging) output
  --verily                   Exchange extra information with the remote
                             to ensure no content is overlooked

See also: clone, config, push, remote, sync
```

7.3.10 sync

This command is used to sync a remote copy with the original copy of the repository, it does both a push and pull. This can also be used to switch a local repository to a different main repository by specifying the URL of a remote repository. If you want to run the update command with -n where it does a dry run, this does not do a sync first so doing fossil sync then fossil update -n will do that for you.

```
$ fossil help sync
Usage: fossil sync ?URL? ?options?

Synchronize all sharable changes between the local repository and a
remote repository.  Sharable changes include public check-ins and
edits to wiki pages, tickets, and technical notes.

If URL is not specified, then the URL from the most recent clone, push,
```

```
pull, remote, or sync command is used.  See "fossil help clone" for
details on the URL formats.

Options:

  -B|--httpauth USER:PASS    Credentials for the simple HTTP auth protocol,
                             if required by the remote website
  --ipv4                     Use only IPv4, not IPv6
  --once                     Do not remember URL for subsequent syncs
  --proxy PROXY              Use the specified HTTP proxy
  --private                  Sync private branches too
  -R|--repository REPO       Local repository to sync with
  --ssl-identity FILE        Local SSL credentials, if requested by remote
  --ssh-command SSH          Use SSH as the "ssh" command
  -u|--unversioned           Also sync unversioned content
  -v|--verbose               Additional (debugging) output
  --verily                   Exchange extra information with the remote
                             to ensure no content is overlooked

See also: clone, pull, push, remote
```

7.3.11 clean

This call can be used to remove all the “extra” files in a source tree. This is useful if you wish to tidy up a source tree or to do a clean build.

```
$ fossil help clean
Usage: fossil clean ?OPTIONS? ?PATH ...?

Delete all "extra" files in the source tree.  "Extra" files are files
that are not officially part of the checkout.  If one or more PATH
arguments appear, then only the files named, or files contained with
directories named, will be removed.

If the --prompt option is used, prompts are issued to confirm the
permanent removal of each file.  Otherwise, files are backed up to the
undo buffer prior to removal, and prompts are issued only for files
whose removal cannot be undone due to their large size or due to
--disable-undo being used.

The --force option treats all prompts as having been answered yes,
whereas --no-prompt treats them as having been answered no.

Files matching any glob pattern specified by the --clean option are
deleted without prompting, and the removal cannot be undone.
```

```
No file that matches glob patterns specified by --ignore or --keep will
ever be deleted.  Files and subdirectories whose names begin with "."
are automatically ignored unless the --dotfiles option is used.

The default values for --clean, --ignore, and --keep are determined by
the (versionable) clean-glob, ignore-glob, and keep-glob settings.

The --verily option ignores the keep-glob and ignore-glob settings and
turns on --force, --emptydirs, --dotfiles, and --disable-undo.  Use the
--verily option when you really want to clean up everything.  Extreme
care should be exercised when using the --verily option.

Options:
   --allckouts            Check for empty directories within any checkouts
                          that may be nested within the current one.  This
                          option should be used with great care because the
                          empty-dirs setting (and other applicable settings)
                          belonging to the other repositories, if any, will
                          not be checked.
   --case-sensitive BOOL  Override case-sensitive setting
   --dirsonly             Only remove empty directories.  No files will
                          be removed.  Using this option will automatically
                          enable the --emptydirs option as well.
   --disable-undo         WARNING: This option disables use of the undo
                          mechanism for this clean operation and should be
                          used with extreme caution.
   --dotfiles             Include files beginning with a dot (".").
   --emptydirs            Remove any empty directories that are not
                          explicitly exempted via the empty-dirs setting
                          or another applicable setting or command line
                          argument.  Matching files, if any, are removed
                          prior to checking for any empty directories;
                          therefore, directories that contain only files
                          that were removed will be removed as well.
   -f|--force             Remove files without prompting.
   -i|--prompt            Prompt before removing each file.  This option
                          implies the --disable-undo option.
   -x|--verily            WARNING: Removes everything that is not a managed
                          file or the repository itself.  This option
                          implies the --force, --emptydirs, --dotfiles, and
                          --disable-undo options. Furthermore, it
                          completely disregards the keep-glob
                          and ignore-glob settings.  However, it does honor
                          the --ignore and --keep options.
   --clean CSG            WARNING: Never prompt to delete any files matching
                          this comma separated list of glob patterns.  Also,
```

```
                          deletions of any files matching this pattern list
                          cannot be undone.
   --ignore CSG           Ignore files matching patterns from the
                          comma separated list of glob patterns.
   --keep <CSG>           Keep files matching this comma separated
                          list of glob patterns.
   -n|--dry-run           Delete nothing, but display what would have been
                          deleted.
   --no-prompt            This option disables prompting the user for input
                          and assumes an answer of 'No' for every question.
   --temp                 Remove only Fossil-generated temporary files.
   -v|--verbose           Show all files as they are removed.

See also: addremove, extras, status
```

7.3.12 branch

This command is used if you want to create or list branches in a repository. Previously we discussed forks (See Section [sub:Complications]); branches are the same idea but under user control. This would be where you have version 1.0 of something but want to branch off version 2.0 to add new features but want to keep a 1.0 branch for maintenance.

```
$ fossil help branch
Usage: fossil branch SUBCOMMAND ... ?OPTIONS?

Run various subcommands to manage branches of the open repository or
of the repository identified by the -R or --repository option.

   fossil branch current

       Print the name of the branch for the current check-out

   fossil branch info BRANCH-NAME

       Print information about a branch

   fossil branch list|ls ?OPTIONS?

       List all branches. Options:
         -a|--all      List all branches.  Default show only open branches
         -c|--closed   List closed branches.
         -r            Reverse the sort order
         -t            Show recently changed branches first

   fossil branch new BRANCH-NAME BASIS ?OPTIONS?
```

```
       Create a new branch BRANCH-NAME off of check-in BASIS.
       Supported options for this subcommand include:
       --private             branch is private (i.e., remains local)
       --bgcolor COLOR       use COLOR instead of automatic background
       --nosign              do not sign contents on this branch
       --date-override DATE  DATE to use instead of 'now'
       --user-override USER  USER to use instead of the current default

       DATE may be "now" or "YYYY-MM-DDTHH:MM:SS.SSS". If in
       year-month-day form, it may be truncated, the "T" may be
       replaced by a space, and it may also name a timezone offset
       from UTC as "-HH:MM" (westward) or "+HH:MM" (eastward).
       Either no timezone suffix or "Z" means UTC.

Options valid for all subcommands:

   -R|--repository FILE       Run commands on repository FILE
```

7.3.13 merge

This command does the opposite of branch, it brings two branches together.

```
$ fossil help merge
Usage: fossil merge ?OPTIONS? ?VERSION?

The argument VERSION is a version that should be merged into the
current checkout.  All changes from VERSION back to the nearest
common ancestor are merged.  Except, if either of the --cherrypick or
--backout options are used only the changes associated with the
single check-in VERSION are merged.  The --backout option causes
the changes associated with VERSION to be removed from the current
checkout rather than added.

If the VERSION argument is omitted, then Fossil attempts to find
a recent fork on the current branch to merge.

Only file content is merged.  The result continues to use the
file and directory names from the current checkout even if those
names might have been changed in the branch being merged in.

Options:

  --backout               Do a reverse cherrypick merge against VERSION.
                          In other words, back out the changes that were
                          added by VERSION.
```

```
  --baseline BASELINE     Use BASELINE as the "pivot" of the merge instead
                          of the nearest common ancestor.  This allows
                          a sequence of changes in a branch to be merged
                          without having to merge the entire branch.

  --binary GLOBPATTERN    Treat files that match GLOBPATTERN as binary
                          and do not try to merge parallel changes.  This
                          option overrides the "binary-glob" setting.

  --case-sensitive BOOL   Override the case-sensitive setting.  If false,
                          files whose names differ only in case are taken
                          to be the same file.

  --cherrypick            Do a cherrypick merge VERSION into the current
                          checkout.  A cherrypick merge pulls in the changes
                          of the single check-in VERSION, rather than all
                          changes back to the nearest common ancestor.

  -f|--force              Force the merge even if it would be a no-op.

  --force-missing         Force the merge even if there is missing content.

  --integrate             Merged branch will be closed when committing.

  -K|--keep-merge-files   On merge conflict, retain the temporary files
                          used for merging, named *-baseline, *-original,
                          and *-merge.

  -n|--dry-run            If given, display instead of run actions

  -v|--verbose            Show additional details of the merge
```

7.3.14 tag

This command can be used to control "tags" which are attributes added to any entry in the time line. You can also add/delete/control these tags from the UI by going into the timeline, picking an entry then doing an edit. See Figure [fig:Remove-Leaf].

```
$ fossil help tag
Usage: fossil tag SUBCOMMAND ...

Run various subcommands to control tags and properties.

  fossil tag add ?OPTIONS? TAGNAME CHECK-IN ?VALUE?
```

```
      Add a new tag or property to CHECK-IN. The tag will
      be usable instead of a CHECK-IN in commands such as
      update and merge.  If the --propagate flag is present,
      the tag value propagates to all descendants of CHECK-IN

      Options:
        --raw                       Raw tag name.
        --propagate                 Propagating tag.
        --date-override DATETIME    Set date and time added.
        --user-override USER        Name USER when adding the tag.
        --dryrun|-n                 Display the tag text, but do not
                                    actually insert it into the database.

      The --date-override and --user-override options support
      importing history from other SCM systems. DATETIME has
      the form 'YYYY-MMM-DD HH:MM:SS'.

fossil tag cancel ?--raw? TAGNAME CHECK-IN

      Remove the tag TAGNAME from CHECK-IN, and also remove
      the propagation of the tag to any descendants.  Use the
      the --dryrun or -n options to see what would have happened.

      Options:
        --raw                       Raw tag name.
        --date-override DATETIME    Set date and time deleted.
        --user-override USER        Name USER when deleting the tag.
        --dryrun|-n                 Display the control artifact, but do
                                    not insert it into the database.

fossil tag find ?OPTIONS? TAGNAME

      List all objects that use TAGNAME.  TYPE can be "ci" for
      check-ins or "e" for events. The limit option limits the number
      of results to the given value.

      Options:
        --raw           Raw tag name.
        -t|--type TYPE  One of "ci", or "e".
        -n|--limit N    Limit to N results.

fossil tag list|ls ?OPTIONS? ?CHECK-IN?

      List all tags, or if CHECK-IN is supplied, list
      all tags and their values for CHECK-IN.  The tagtype option
```

```
        takes one of: propagated, singleton, cancel.

        Options:
          --raw           List tags raw names of tags
          --tagtype TYPE  List only tags of type TYPE
          -v|--inverse    Inverse the meaning of --tagtype TYPE.

The option --raw allows the manipulation of all types of tags
used for various internal purposes in fossil. It also shows
"cancel" tags for the "find" and "list" subcommands. You should
not use this option to make changes unless you are sure what
you are doing.

If you need to use a tagname that might be confused with
a hexadecimal baseline or artifact ID, you can explicitly
disambiguate it by prefixing it with "tag:". For instance:

  fossil update decaf

will be taken as an artifact or baseline ID and fossil will
probably complain that no such revision was found. However

  fossil update tag:decaf

will assume that "decaf" is a tag/branch name.
```

7.3.15 settings

This command is used to set or unset a number of properties for fossil.

```
$ fossil help settings
Usage: fossil settings ?SETTING? ?VALUE? ?OPTIONS?
   or: fossil unset SETTING ?OPTIONS?

The "settings" command with no arguments lists all settings and their
values.  With just a SETTING name it shows the current value of that setting
With a VALUE argument it changes the property for the current repository.

Settings marked as versionable are overridden by the contents of the
file named .fossil-settings/PROPERTY in the check-out root, if that
file exists.

The "unset" command clears a setting.

Settings can have both a "local" repository-only value and "global" value
that applies to all repositories.  The local values are stored in the
```

```
"config" table of the repository and the global values are stored in the
configuration database.  If both a local and a global value exists for a
setting, the local value takes precedence.  This command normally operates
on the local settings.  Use the --global option to change global settings.

Options:
  --global   set or unset the given property globally instead of
             setting or unsetting it for the open repository only.

  --exact    only consider exact name matches.

See also: configuration
```

7.3.16 info

```
fossil info ?VERSION | REPOSITORY_FILENAME? ?OPTIONS?
```

This command show various information on the repository, a bit like the `status` command does. Using the command without any arguments, it shows information on the current checkout:

- name of the project and its code (hash value)
- path to the repository
- path to Fossil's global configuration database for the user (typically `~./fossil`)
- path to the local open checkout
- id and time stamp of the last checkout, its parent and child
- tags
- last commit message

When you do not want information on the last check-in, but on other versions, you just supply the version on the command line. You can also explicitly provide the repository file with the `-R` option. Then you will only get the project name and code and the number of chek-ins.

The `-v` option is helpful since it provides not only the above information on the current checkout but also tells you the locations of all other known checkouts there are (if you have multiple open checkouts of the same repository in different directories). It also lists the possible access URLs that have been used over time (for the ui).

7.3.17 publish

```
fossil publish ?--only? TAGS...
```

This command causes artifacts identified by the given `TAGS...` (these can be more than one tag) to be published (i.e. made non-private). This can be used

(for example) to convert a private branch into a public branch, or to publish a bundle that was imported privately.

If any of `TAGS` names a branch, then all check-ins on the most recent instance of that branch are included, not just the most recent check-in.

If any of `TAGS` name check-ins then all files and tags associated with those check-ins are also published automatically. If the `--only` option is used, then only the specific artifacts identified by `TAGS` are published.

If a `TAG` is already public, this command is a harmless no-op.

7.4 Miscellaneous

These are commands that don't seem to fit in any category but are useful.

7.4.1 zip

You can do what this command does from the web based user interface. In Figure [fig:Timeline-Detail] you can download a ZIP archive of the particular version of the files. This command lets you do it from the command line.

```
$ fossil help zip
Usage: fossil zip VERSION OUTPUTFILE [OPTIONS]

Generate a ZIP archive for a check-in.  If the --name option is
used, its argument becomes the name of the top-level directory in the
resulting ZIP archive.  If --name is omitted, the top-level directory
name is derived from the project name, the check-in date and time, and
the artifact ID of the check-in.

The GLOBLIST argument to --exclude and --include can be a comma-separated
list of glob patterns, where each glob pattern may optionally be enclosed
in "..." or '...' so that it may contain commas.  If a file matches both
--include and --exclude then it is excluded.

Options:
  -X|--exclude GLOBLIST   Comma-separated list of GLOBs of files to exclude
  --include GLOBLIST      Comma-separated list of GLOBs of files to include
  --name DIRECTORYNAME    The name of the top-level directory in the archive
  -R REPOSITORY           Specify a Fossil repository
```

7.4.2 user

This command lets you modify user information. Again this is a command line duplication of what you can do from the user interface in the browser, see Figure [fig:New-Editor-user].

```
$ fossil help user
Usage: fossil user SUBCOMMAND ...  ?-R|--repository FILE?

Run various subcommands on users of the open repository or of
the repository identified by the -R or --repository option.

  fossil user capabilities USERNAME ?STRING?

       Query or set the capabilities for user USERNAME

  fossil user contact USERNAME ?CONTACT-INFO?

       Query or set contact information for user USERNAME

  fossil user default ?USERNAME?

       Query or set the default user.  The default user is the
       user for command-line interaction.

  fossil user list
  fossil user ls

       List all users known to the repository

  fossil user new ?USERNAME? ?CONTACT-INFO? ?PASSWORD?

       Create a new user in the repository.  Users can never be
       deleted.  They can be denied all access but they must continue
       to exist in the database.

  fossil user password USERNAME ?PASSWORD?

       Change the web access password for a user.
```

7.4.3 finfo

This command will print the history of any particular file. This can be useful if you need this history in some other system. You can pass this text file to the other system which can than parse and use the data.

```
$ fossil help finfo
Usage: fossil finfo ?OPTIONS? FILENAME

Print the complete change history for a single file going backwards
in time.  The default mode is -l.
```

```
For the -l|--log mode: If "-b|--brief" is specified one line per revision
is printed, otherwise the full comment is printed.  The "-n|--limit N"
and "--offset P" options limits the output to the first N changes
after skipping P changes.

In the -s mode prints the status as <status> <revision>.  This is
a quick status and does not check for up-to-date-ness of the file.

In the -p mode, there's an optional flag "-r|--revision REVISION".
The specified version (or the latest checked out version) is printed
to stdout.  The -p mode is another form of the "cat" command.

Options:
  -b|--brief           display a brief (one line / revision) summary
  --case-sensitive B   Enable or disable case-sensitive filenames.  B is a
                       boolean: "yes", "no", "true", "false", etc.
  -l|--log             select log mode (the default)
  -n|--limit N         Display the first N changes (default unlimited).
                       N less than 0 means no limit.
  --offset P           skip P changes
  -p|--print           select print mode
  -r|--revision R      print the given revision (or ckout, if none is given)
                       to stdout (only in print mode)
  -s|--status          select status mode (print a status indicator for FILE
  -W|--width N         Width of lines (default is to auto-detect). Must be
                       more than 22 or else 0 to indicate no limit.

See also: artifact, cat, descendants, info, leaves
```

An example would be to run it on the outline.txt file in the earlier version of our book directory:

```
$ fossil finfo outline.txt

    History of outline.txt

    2010-05-17 [0272dc0169] Finished maintenance commands (user: jim, artifa

    {}           [25b6e38e97])

    2010-05-12 [5e5c0f7d55] End of day commit (user: jim, artifact: [d1a1d31

    2010-05-10 [e924ca3525] End of day update (user: jim, artifact: [7cd1907

    2010-05-09 [0abb95b046] Intermediate commit, not done with basic command

    {}           (user: jim, artifact: [6f7bcd48b9])
```

```
    2010-05-07 [6921e453cd] Update outline & book corrections (user: jim,
    {}           artifact: [4eff85c793])
    2010-05-03 [158492516c] Moved to clone repository (user: jim, artifact:
    {}           [23b729cb66])
    2010-05-03 [1a403c87fc] Update before moving to server (user: jim, artifact:
    {}           [706a9d394d])
    2010-04-30 [fa5b9247bd] Working on chapter 1 (user: jim, artifact:
    {}           [7bb188f0c6])
    2010-04-29 [51be6423a3] Update outline (user: jim, artifact: [7cd39dfa06])
    2010-04-27 [39bc728527] [1665c78d94] Ticket Use (user: jim, artifact:
    {}           [1f82aaf41c])
    2010-04-26 [497b93858f] Update to catch changes in outline (user: jim,
    {}           artifact: [b870231e48])
    2010-04-25 [8fa0708186] Initial Commit (user: jim, artifact: [34a460a468])
```

7.4.4 timeline

This prints out the timeline of the project in various ways. The command would be useful if you were building a GUI front end for Fossil and wanted to display the timeline. You could issue this command and get the result back and display it in your UI. There are a number of options in the command to control the listing.

```
$ fossil help timeline
Usage: fossil timeline ?WHEN? ?CHECKIN|DATETIME? ?OPTIONS?

Print a summary of activity going backwards in date and time
specified or from the current date and time if no arguments
are given.  The WHEN argument can be any unique abbreviation
of one of these keywords:

    before
```

```
    after
    descendants | children
    ancestors | parents

The CHECKIN can be any unique prefix of 4 characters or more. You
can also say "current" for the current version.

DATETIME may be "now" or "YYYY-MM-DDTHH:MM:SS.SSS". If in
year-month-day form, it may be truncated, the "T" may be replaced by
a space, and it may also name a timezone offset from UTC as "-HH:MM"
(westward) or "+HH:MM" (eastward). Either no timezone suffix or "Z"
means UTC.


Options:
  -n|--limit N         If N is positive, output the first N entries.  If
                       N is negative, output the first -N lines.  If N is
                       zero, no limit.  Default is -20 meaning 20 lines.
  -p|--path PATH       Output items affecting PATH only.
                       PATH can be a file or a sub directory.
  --offset P           skip P changes
  --sql                Show the SQL used to generate the timeline
  -t|--type TYPE       Output items from the given types only, such as:
                           ci = file commits only
                           e  = technical notes only
                           t  = tickets only
                           w  = wiki commits only
  -v|--verbose         Output the list of files changed by each commit
                       and the type of each change (edited, deleted,
                       etc.) after the check-in comment.
  -W|--width N         Width of lines (default is to auto-detect). N must be
                       either greater than 20 or it ust be zero 0 to
                       indicate no limit, resulting in a single line per
                       entry.
  -R REPO_FILE         Specifies the repository db to use. Default is
                       the current checkout's repository.
```

7.4.5 wiki

This command allows you to have command line control of the wiki. Again this is useful if you were writing a shell to control Fossil or wanted to add a number of computer generated pages to the Wiki.

```
$ fossil help wiki
Usage: fossil wiki (export|create|commit|list) WikiName
```

```
Run various subcommands to work with wiki entries or tech notes.

  fossil wiki export ?OPTIONS? PAGENAME ?FILE?
  fossil wiki export ?OPTIONS? -t|--technote DATETIME|TECHNOTE-ID ?FILE?

      Sends the latest version of either a wiki page or of a tech
      note to the given file or standard output.  A filename of "-"
      writes the output to standard output.  The directory parts of
      the output filename are created if needed.
      If PAGENAME is provided, the named wiki page will be output.

      Options:
        --technote|-t DATETIME|TECHNOTE-ID
                    Specifies that a technote, rather than a wiki page,
                    will be exported. If DATETIME is used, the most
                    recently modified tech note with that DATETIME will
                    output.
        -h|--html   The body (only) is rendered in HTML form, without
                    any page header/foot or HTML/BODY tag wrappers.
        -H|--HTML   Works like -h|-html but wraps the output in
                    <html><body>...</body></html>.
        -p|--pre    If -h|-H is used and the page or technote has
                    the text/plain mimetype, its HTML-escaped output
                    will be wrapped in <pre>...</pre>.

  fossil wiki (create|commit) PAGENAME ?FILE? ?OPTIONS?

      Create a new or commit changes to an existing wiki page or
      technote from FILE or from standard input. PAGENAME is the
      name of the wiki entry or the timeline comment of the
      technote.

      Options:
        -M|--mimetype TEXT-FORMAT   The mime type of the update.
                                    Defaults to the type used by
                                    the previous version of the
                                    page, or text/x-fossil-wiki.
                                    Valid values are: text/x-fossil-wiki,
                                    text/x-markdown and text/plain. fossil,
                                    markdown or plain can be specified as
                                    synonyms of these values.
        -t|--technote DATETIME      Specifies the timestamp of
                                    the technote to be created or
                                    updated. When updating a tech note
                                    the most recently modified tech note
                                    with the specified timestamp will be
```

```
                                  updated.
       -t|--technote TECHNOTE-ID  Specifies the technote to be
                                  updated by its technote id.
       --technote-tags TAGS       The set of tags for a technote.
       --technote-bgcolor COLOR   The color used for the technote
                                  on the timeline.

  fossil wiki list ?OPTIONS?
  fossil wiki ls ?OPTIONS?

      Lists all wiki entries, one per line, ordered
      case-insensitively by name.

      Options:
        -t|--technote              Technotes will be listed instead of
                                   pages. The technotes will be in order
                                   of timestamp with the most recent
                                   first.
        -s|--show-technote-ids     The id of the tech note will be listed
                                   along side the timestamp. The tech note
                                   id will be the first word on each line.
                                   This option only applies if the
                                   --technote option is also specified.

DATETIME may be "now" or "YYYY-MM-DDTHH:MM:SS.SSS". If in
year-month-day form, it may be truncated, the "T" may be replaced by
a space, and it may also name a timezone offset from UTC as "-HH:MM"
(westward) or "+HH:MM" (eastward). Either no timezone suffix or "Z"
means UTC.

The "Sandbox" wiki pseudo-page is a special case. Its name is
checked case-insensitively and either "create" or "commit" may be
used to update its contents.
```

7.5 Advanced

These are commands that you will rarely have to use. These are functions that are needed to do very complicated things with Fossil. If you have to use these you are probably way beyond the audience for this book.

7.5.1 scrub

This is used to removed sensitive information like passwords from a repository. This allows you to then send the whole repository to someone else for their use.

```
$ fossil help scrub
Usage: fossil scrub ?OPTIONS? ?REPOSITORY?

The command removes sensitive information (such as passwords) from a
repository so that the repository can be sent to an untrusted reader.

By default, only passwords are removed.  However, if the --verily option
is added, then private branches, concealed email addresses, IP
addresses of correspondents, and similar privacy-sensitive fields
are also purged.  If the --private option is used, then only private
branches are removed and all other information is left intact.

This command permanently deletes the scrubbed information. THE EFFECTS
OF THIS COMMAND ARE NOT REVERSIBLE. USE WITH CAUTION!

The user is prompted to confirm the scrub unless the --force option
is used.

Options:
  --force     do not prompt for confirmation
  --private   only private branches are removed from the repository
  --verily    scrub real thoroughly (see above)
```

7.5.2 search

This is used to search the timeline entries for a pattern. This can also be done in your browser on the timeline page.

```
$ fossil help search
Usage: fossil search [-all|-a] [-limit|-n #] [-width|-W #] pattern...

Search for timeline entries matching all words provided on the
command line. Whole-word matches scope more highly than partial
matches.

Note:  The command only search the EVENT table.  So it will only
display check-in comments or other comments that appear on an
unaugmented timeline.  It does not search document text or forum
messages.

Outputs, by default, some top-N fraction of the results. The -all
option can be used to output all matches, regardless of their search
score.  The -limit option can be used to limit the number of entries
returned.  The -width option can be used to set the output width used
when printing matches.
```

```
Options:

    -a|--all          Output all matches, not just best matches.
    -n|--limit N      Limit output to N matches.
    -W|--width WIDTH  Set display width to WIDTH columns, 0 for
                      unlimited. Defaults the terminal's width.
```

7.5.3 sha3sum

This can compute the sha1 value for a particular file. These sums are the labels that Fossil uses on all objects and should be unique for any file.

```
$ fossil help sha3sum
Usage: fossil sha3sum FILE...

Compute an SHA3 checksum of all files named on the command-line.
If a file is named "-" then take its content from standard input.

To be clear:  The official NIST FIPS-202 implementation of SHA3
with the added 01 padding is used, not the original Keccak submission.

Options:

   --224              Compute a SHA3-224 hash
   --256              Compute a SHA3-256 hash (the default)
   --384              Compute a SHA3-384 hash
   --512              Compute a SHA3-512 hash
   --size N           An N-bit hash.  N must be a multiple of 32 between
                      128 and 512.
   -h, --dereference  If FILE is a symbolic link, compute the hash on
                      the object pointed to, not on the link itself.

See also: md5sum, sha1sum
```

7.5.4 configuration

This command allows you to save or load a custom configuration of Fossil.

```
$ fossil help configuration
Usage: fossil configuration METHOD ... ?OPTIONS?

Where METHOD is one of: export import merge pull push reset.  All methods
accept the -R or --repository option to specify a repository.

   fossil configuration export AREA FILENAME

        Write to FILENAME exported configuration information for AREA.
```

```
        AREA can be one of:

            all email interwiki project shun skin
            ticket user alias subscriber

   fossil configuration import FILENAME

        Read a configuration from FILENAME, overwriting the current
        configuration.

   fossil configuration merge FILENAME

        Read a configuration from FILENAME and merge its values into
        the current configuration.  Existing values take priority over
        values read from FILENAME.

   fossil configuration pull AREA ?URL?

        Pull and install the configuration from a different server
        identified by URL.  If no URL is specified, then the default
        server is used.  Use the --overwrite flag to completely
        replace local settings with content received from URL.

   fossil configuration push AREA ?URL?

        Push the local configuration into the remote server identified
        by URL.  Admin privilege is required on the remote server for
        this to work.  When the same record exists both locally and on
        the remote end, the one that was most recently changed wins.

   fossil configuration reset AREA

        Restore the configuration to the default.  AREA as above.

   fossil configuration sync AREA ?URL?

        Synchronize configuration changes in the local repository with
        the remote repository at URL.

Options:
   -R|--repository FILE       Extract info from repository FILE

See also: settings, unset
```

7.5.5 descendants

This is used to find where the checked out files are in the time line.

```
$ fossil help descendants
Usage: fossil descendants ?CHECKIN? ?OPTIONS?

Find all leaf descendants of the check-in specified or if the argument
is omitted, of the check-in currently checked out.

Options:
   -R|--repository FILE       Extract info from repository FILE
   -W|--width N               Width of lines (default is to auto-detect).
                              Must be greater than 20 or else 0 for no
                              limit, resulting in a one line per entry.

See also: finfo, info, leaves
```

Chapter 8

Pikchr

This section needs completion.

Chapter 9

TH1 Scripting language

TH1 is designed to be the scripting language of Fossil. It is used in several places within Fossil. As a typical user, you will probably first meet TH1 when cusomizing the ticket system (see 5.7 for an overview of the ticket system and 6.1 on how to customize the ticket system). You can customize the behaviour of the ticket web pages that show

- the form for creating new tickets
- existing tickets
- the form for editing existing tickets
- the list of ticket reports

If you are changing the overall look and feel of Fossil's web pages, you can use TH1 when designing the skin. Both the "Content Header" and the "Content Footer" can contain TH1 code (also the optional Javascript for the footer). This includes the possibility to change Fossil's default "Content Secutiry Policy" (CSP) which can be overridden by custom TH1 code.

Further, 8 scripts can include TH1 scripts. However, this is restricted to the Fossil command `pikchr` on the command line. TH1 scripts are not enabled within Fossil's web pages (also including `/pikchrshow`).

Finally, there are two place where TH1 can be used when Fossil is explicitly enabled to do so at compile-time:

- when compiled with the `--with-th1-docs` option, Fossil's embedded documentation facility via the `/doc` web pages can containt TH1 scripts. It is a compile-time option since it opens Fossil scripting to anyone who has commit capabilities on the repository and thus can commit documentation files containing arbitrary TH1 code. This can be dangerous and must therefore be enabled already at compile-time. In addition, this option must be enabled in Fossil's settings (option `th1-docs` in the `Admin` menu under `Settings`; see section 6). Note, that TH1 scripts only work in

embedded documentation files having the mime type "application/x-th1". The ideal situation is to use a file having a file name ending in `.th1` and containing HTML. The TH1 code is then embedded into the special `<th1>` tag.
- when compiled with the `--with-th1-hooks` option, TH1 scripts can be used to monitor, create, alter, or cancel the execution of Fossil commands and web pages (additionally, the option `th1-hooks` must be enabled at runtime (in the `Admin` menu under `Settings`)). Hooks can then be set to run before and/or after Fossil command execution, and before and/or after the rendering of a web page.

9.1 Introduction to TH1

9.1.1 TH1 is a Tcl-like language

TH1 is a string-based command language closely based on the Tcl language. The language has only a few fundamental constructs and relatively little syntax which is meant to be simple. TH1 is an interpreted language and it is parsed, compiled and executed when the script runs. In Fossil, TH1 scripts are typically run to build web pages, often in response to web form submissions.

The basic mechanisms of TH1 are all related to strings and string substitutions. The TH1/Tcl way of doing things is a little different from some other programming languages with which you may already be familiar, so it is worth making sure you understand the basic concepts.

9.2 The Hello world program

The traditional starting place for a language introduction is the classic "Hello, World" program. In TH1 this program has only a single line:

```
puts "Hello, world\n"
```

The command to output a string in TH1 is the `puts` command (the same command name as in Tcl). A single unit of text after the `puts` command will be printed to the output stream. If the string has more than one word, you must enclose the string in double quotes or curly brackets. A set of words enclosed in quotes or curly brackets is treated as a single unit, while words separated by white space are treated as multiple arguments to the command.

9.3 TH1 structure and syntax

9.3.1 Datatypes

TH1 has at its core only a single data type which is string. All values in TH1 are strings, variables hold strings and the procedures return strings. Strings in TH1 consist of single byte characters and are zero terminated. Characters outside of the ASCII range, i.e. characters in the 0x80-0xff range have no TH1 meaning at all: they are not considered digits or letters, nor are they considered white space.

Depending on context, TH1 can interpret a string in four different ways. First of all, strings can be just that: text consisting of arbitrary sequences of characters. Second, a string can be considered a list, an ordered sequence of words separated by white space. Third, a string can be a command. A command is a list where the first word is interpreted as the name of a command, optionally followed by further argument words. Fourth, and last, a string can be interpreted as an expression.

The latter three interpretations of strings are discussed in more detail below.

9.3.2 Lists

A list in TH1 is an ordered sequence of items, or words separated by white space. In TH1 the following characters constitute white space:

```
' '    0x20
'\t'   0x09
'\n'   0x0A
'\v'   0x0B
'\f'   0x0C
'\r'   0x0D
```

A word can be any sequence of characters delimited by white space. It is not necessary that a word is alphanumeric at all: `".%*"` is a valid word in TH1. If a word needs to contain embedded white space characters, it needs to be quoted, with either a double quotes or with opening/closing curly brackets. Quoting has the effect of grouping the quoted content into a single list element.

Words cannot start with one of the TH1 special characters `{ } [ ] \ ;` and `"`. Note that a single quote `'` is not a special character in TH1. To use one of these characters to start a word it must be escaped, which is discussed further later on.

TH1 offers several built-in commands for working with lists, such as counting the number of words in a list, retrieving individual words from the list by index and appending new items to a list. These commands are discussed in the section "Working with lists".

9.3.3 Commands

TH1 casts everything into the mold of a command, even programming constructs like variable assignment and procedure definition. TH1 adds a tiny amount of syntax needed to properly invoke commands, and then it leaves all the hard work up to the command implementation.

Commands are a special form of list. The basic syntax for a TH1 command is:

```
command arg1 arg2 arg3 ...
```

The command is either the name of a built-in command or a TH1 procedure.

White space is used to separate the command name and its arguments, and a newline character or semicolon is used to terminate a command. TH1 comments are lines with a `#` character at the beginning of the line, or with a `#` character after the semicolon terminating a command.

9.3.4 Grouping & substitution

TH1 does not interpret the arguments to the commands except to perform grouping, which allows multiple words in one argument, and substitution, which is used to deal with special characters, to fetch variables and to perform nested command calls. Hence, the behavior of the TH1 command processor can be summarized in three basic steps:

1. Argument grouping.
2. Value substitution of backslash escapes, variables and nested commands
3. Command invocation.

Note that there is no step to evaluate the arguments to a command. After substitution, arguments are passed verbatim to the command and it is up to the command to evaluate its arguments as needed.

9.3.5 Argument grouping

TH1 has two mechanisms for grouping multiple words into a single item:

- Double quotes, `" "`
- Curly brackets, `{ }`

Whilst both have the effect of grouping a set of words, they have different impact on the next phase of substitution. In brief, double quotes only group their content and curly brackets group and prevent all substitution on their content.

Grouping proceeds from left to right in the string and is not affected by the subsequent substitution. If a substitution leads to a string which would be grouped differently, it has no effect, as the grouping has already been decided in the preceding grouping phase.

9.3.6 Value substitutions

TH1 performs three different substitutions (see the th.c/thSubstWord code for details)

- Backslash escape substitution
- Variable substitution
- Nested command substitution

Like grouping, substitution proceeds from left to right and is performed only once: if a substitution leads to a string which could again be substituted such this not happen.

9.3.7 Backslash escape substitution

In general, the backslash (`\`) disables substitution for the single character immediately following the backslash. Any character immediately following the backslash will stand as literal. This is useful to escape the special meaning of the characters `{ } [ ] \ ;` and `"`.

There are two specific strings which are replaced by specific values during the substitution phase. A backslash followed by the letter `n` gets replaced by the newline character, as in C. A backslash followed by the letter `x` and two hexadecimal digits gets replaced by the character with that value, i.e. writing `\x20` is the same as writing a space. Note that the `\x` substitution does not "keep going" as long as it has hex digits as in Tcl, but insists on two digits. The word `\x2121` is not a single exclamation mark, but the 3 letter word !21.

9.3.8 Variable substitution

Like any programming language, TH1 has a concept of variables. TH1 variables are named containers that hold string values. Variables are discussed in more detail later in this document, for now we limit ourselves to variable substitution.

The dollar sign (`$`) may be used as a special shorthand form for substituting variable values. If $ appears in an argument that isn't enclosed in curly brackets then variable substitution will occur. The characters after the $, up to the first character that isn't a number, letter, or underscore, are taken as a variable name and the string value of that variable is substituted for the name. For example, if variable foo has the value test, then the command `puts $foo.c` is equivalent to the command:

```
puts test.c
```

There are two special forms for variable substitution. If the next character after the name of the variable is an open parenthesis, then the variable is assumed to be an array name, and all of the characters between the open parenthesis and the next close parenthesis are taken as an index into the array. Command substitutions and variable substitutions are performed on the information between

the parentheses before it is used as an index. For example, if the variable x is an array with one element named first and value 87 and another element named 14 and value more, then the command

```
puts xyz$x(first)zyx
```

is equivalent to the command

```
puts xyz87zyx
```

If the variable index has the value '14', then the command

```
puts xyz$x($index)zyx
```

is equivalent to the command

```
puts xyzmorezyx
```

See the section Variables and arrays below for more information on arrays.

The second special form for variables occurs when the dollar sign is followed by an open curly bracket. In this case the variable name consists of all the characters up to the next curly bracket. Array references are not possible in this form: the name between curly brackets is assumed to refer to a scalar variable. For example, if variable foo has the value 'test', then the command

```
set a abc${foo}bar
```

is equivalent to the command

```
set a abctestbar
```

A dollar sign followed by something other than a letter, digit, underscore, or left parenthesis is treated as a literal dollar sign. The following prints a single character $.

```
puts x $
```

9.3.9 Command substitution

The last form of substitution is command substitution. A nested command is delimited by square brackets, `[ ]`. The TH1 interpreter takes everything between the brackets and evaluates it as a command. It rewrites the outer command by replacing the square brackets and everything between them with the result of the nested command.

Example:

```
puts \[string length foobar]

=> 6
```

In the example, the nested command is: string length foobar. This command returns the length of the string foobar. The nested command runs first. Then, command substitution causes the outer command to be rewritten as if it were:

```
puts 6
```

If there are several cases of command substitution within a single command, the interpreter processes them from left to right. As each right bracket is encountered, the command it delimits is evaluated. This results in a sensible ordering in which nested commands are evaluated first so that their result can be used in arguments to the outer command.

9.3.10 Argument grouping revisited

During the substitution phase of command evaluation, the two grouping operators, the curly bracket and the double quote are treated differently by the TH1 interpreter.

Grouping words with double quotes allows substitutions to occur within the double quotes. A double quote is only used for grouping when it comes after white space. The string `a"b"` is a normal 4 character string, and not the two character string ab.

```
puts a"b"

=> a"b"
```

Grouping words within curly brackets disables substitution within the brackets. Again, A opening curly bracket is only used for grouping when it comes after white space. Characters within curly brackets are passed to a command exactly as written, and not even backslash escapes are processed.

Note that curly brackets have this effect only when they are used for grouping (i.e. at the beginning and end of a sequence of words). If a string is already grouped, either with double quotes or curly brackets, and the curly brackets occur in the middle of the grouped string (e.g. “foo{bar”), then the curly brackets are treated as regular characters with no special meaning. If the string is grouped with double quotes, substitutions will occur within the quoted string, even between the brackets.

The square bracket syntax used for command substitution does not provide grouping. Instead, a nested command is considered part of the current group. In the command below, the double quotes group the last argument, and the nested command is just part of that group:

```
puts "The length of $s is [string length $s]."
```

If an argument is made up of only a nested command, you do not need to group it with double-quotes because the TH1 parser treats the whole nested command as part of the group. A nested command is treated as an unbroken

sequence of characters, regardless of its internal structure. It is included with the surrounding group of characters when collecting arguments for the main command.

9.3.11 Summary

The following rules summarize the fundamental mechanisms of grouping and substitution that are performed by the TH1 interpreter before it invokes a command:

- Command arguments are separated by white space, unless arguments are grouped with curly brackets or double quotes as described below.
- Grouping with curly brackets, `{}`, prevents substitutions. Curly brackets nest. The interpreter includes all characters between the matching left and right brace in the group, including newlines, semicolons, and nested curly brackets. The enclosing (i.e., outermost) curly brackets are not included in the group's value.
- Grouping with double quotes, `" "`, allows substitutions. The interpreter groups everything until another double quote is found, including newlines and semicolons. The enclosing quotes are not included in the group of characters. A double-quote character can be included in the group by quoting it with a backslash, (i.e. `\"`).
- Grouping decisions are made before substitutions are performed, which means that the values of variables or command results do not affect grouping.
- A dollar sign, $, causes variable substitution. Variable names can be any length, and case is significant. If variable references are embedded into other strings, or if they include characters other than letters, digits, and the underscore, they can be distinguished with the `${varname}` syntax.
- Square brackets, `[]`, cause command substitution. Everything between the brackets is treated as a command, and everything including the brackets is replaced with the result of the command. Nesting is allowed.
- The backslash character, `\`, is used to quote special characters. You can think of this as another form of substitution in which the backslash and the next character or group of characters is replaced with a new character.
- Substitutions can occur anywhere unless prevented by curly bracket grouping. Part of a group can be a constant string, and other parts of it can be the result of substitutions. Even the command name can be affected by substitutions.
- A single round of substitutions is performed before command invocation. The result of a substitution is not interpreted a second time. This rule is important if you have a variable value or a command result that

contains special characters such as spaces, dollar signs, square brackets, or curly brackets. Because only a single round of substitution is done, you do not have to worry about special characters in values causing extra substitutions.

9.3.12 Caveats

- A common error is to forget a space between arguments when grouping with curly brackets or quotes. This is because white space is used as the separator, while the curly brackets or quotes only provide grouping. If you forget the space, you will get syntax errors about the wrong number of arguments being applied. The following is an error because of the missing space between } and {:

```
if {$x > 1}{    puts "x = $x"}
```

- When double quotes are used for grouping, the special effect of curly brackets is turned off. Substitutions occur everywhere inside a group formed with double quotes. In the next command, the variables are still substituted:

```
set x xvalue

set y "foo {$x} bar"

=> foo {xvalue} bar
```

- Spaces are not required around the square brackets used for command substitution. For the purposes of grouping, the interpreter considers everything between the square brackets as part of the current group.

9.4 TH1 expressions

The TH1 interpreter itself does not evaluate math expressions. TH1 just does grouping, substitutions and command invocations. However, several built-in commands see one of more of their arguments as expressions and request the interpreter to calculate the value of such expressions.

The expr command is the simplest such command and is used to parse and evaluate expressions:

```
puts [expr 7.4/2]

=> 3.7
```

Note that an expression can contain white space, but if it does it must be grouped in order to be recognized as a single argument.

Within the context of expression evaluation TH1 works with three datatypes: two types of number, integer and floating point, and string. Integer values are promoted to floating point values as needed. The Boolean values True and False are represented by the integer values 1 and 0 respectively. The implementation of expr is careful to preserve accurate numeric values and avoid unnecessary conversions between numbers and strings.

Before expression evaluation takes place, both variable and command substitution is performed on the expression string. Hence, you can include variable references and nested commands in math expressions, even if the expression string was originally quoted with curly brackets. Note that backslash escape substitution is not performed by the expression evaluator.

A TH1 expression consists of a combination of operands, operators, and parentheses. White space may be used between the operands and operators and parentheses; it is ignored by the expression processor. Where possible, operands are interpreted as integer values. If an operand is not in integer format, then it is treated as a floating-point number if that is possible. Floating-point numbers may be specified in any of the ways accepted by an ANSI-compliant C compiler. For example, all of the following are valid floating-point numbers: `2.1`, `3.`, `6e4`, `7.91e+16`. If no numeric interpretation is possible, then an operand is left as a string (and only a limited set of operators may be applied to it).

Operands may be specified in any of the following ways:

- As a numeric value, either integer or floating-point.
- As a string enclosed in curly brackets. The characters between the opening bracket and matching closing bracket are used as the operand without any substitutions.
- As a string enclosed in double quotes. The expression parser performs variable and command substitutions on the information between the quotes, and uses the resulting value as the operand.
- As a TH1 variable, using standard $ notation. The variable's value is used as the operand.
- As a TH1 command enclosed in square brackets. The command will be executed and its result will be used as the operand.

Where substitutions occur above (e.g. inside quoted strings), they are performed by the expression processor. However, an additional layer of substitution may already have been performed by the command parser before the expression processor was called. As discussed below, it is usually best to enclose expressions in curly brackets to prevent the command parser from performing substitutions on the contents.

The valid operators are listed below, grouped in decreasing order of precedence:

All of the binary operators group left-to-right within the same precedence level. For example, the expression `4*2 < 7` evaluates to 0.

All internal computations involving integers are done with the C type int, and all internal computations involving floating-point are done with the C type double. Conversion among internal representations for integer, floating-point, and string operands is done automatically as needed. For arithmetic computations, integers are used until some floating-point number is introduced, after which floating-point is used.

String values may be used as operands of the comparison operators, although the expression evaluator tries to do comparisons as integer or floating-point when it can. If one of the operands of a comparison is a string and the other has a numeric value, the numeric operand is converted back to a string.

9.5 TH1 variables

Like almost all programming languages TH1 has the concept of variables. TH1 variables bind a name to a string value. A variable name must be unique in its scope, either the global scope or a local scope. TH1 supports two types of variables: scalars and arrays.

TH1 allows the definition of variables and the use of their values either through `$`-style variable substitution, the set command, or a few other mechanisms. Variables need not be declared: a new variable will automatically be created each time a new variable name is used.

9.5.1 Working with variables

TH1 has two key commands for working with variables, set and unset:

```
set varname ?value?

unset varname

info exists varname
```

The `set` command returns the value of variable varname. If the variable does not exist, then an error is thrown. If the optional argument value is specified, then the set command sets the value of varname to value, creating a new variable in the current scope if one does not already exist, and returns its value.

The `unset` command removes a variable from its scope. The argument varname is a variable name. If varname refers to an element of an array, then that element is removed without affecting the rest of the array. If varname consists of an array name with no parenthesized index, then the entire array is deleted. The unset command returns an empty string as result. An error occurs if the variable doesn't exist.

The `info exists` command returns `1` if the variable named varname exists in the current scope, either the global scope or the current local scope, and returns `0` otherwise.

9.5.2 Scalar variables and array variables

TH1 supports two types of variables: scalars and arrays. A scalar variable has a single value, whereas an array variable can have any number of elements, each with a name (called its "index") and a value. TH1 arrays are one-dimensional associative arrays, i.e. the index can be any single string.

If varname contains an open parenthesis and ends with a close parenthesis, then it refers to an array element: the characters before the open parenthesis are the name of the array, and the characters between the parentheses are the index within the array. Otherwise varname refers to a scalar variable.

For example, the command `set x(first) 44` will modify the element of x whose index is first so that its new value is 44. Two-dimensional arrays can be simulated in TH1 by using indices that contain multiple concatenated values. For example, the commands

```
set a(2,3) 1

set a(3,6) 2
```

set the elements of a whose indices are 2,3 and 3,6.

In general, array elements may be used anywhere in TH1 that scalar variables may be used. If an array is defined with a particular name, then there may not be a scalar variable with the same name. Similarly, if there is a scalar variable with a particular name then it is not possible to make array references to the variable. To convert a scalar variable to an array or vice versa, remove the existing variable with the unset command.

9.5.3 Variable scope

Variables exist in a scope. The TH1 interpreter maintains a global scope that is available to and shared by all commands executed by it. Each invocation of a user defined command creates a new local scope. This local scope holds the arguments and local variables of that user command invocation and only exists as long a the user command is executing.

If not in the body of user command, then references to varname refer to a global variable, i.e. a variable in the global scope. In contrast, in the body of a user defined command references to varname refer to a parameter or local variable of the command. However, in the body of a user defined command, a global variable can be explicitly referred to by preceding its name by ::.

TH1 offers a special command to access variables not in the local scope of the current command but in the local scope of the call chain of commands that leas to the current command. This is the `upvar` command:

```
upvar ?frame? othervar myvar ?othervar myvar ...?
```

The `upvar` command arranges for one or more local variables in the current procedure to refer to variables in an enclosing procedure call, or to global variables. If frame is an integer, then it gives a distance (up the command calling stack) to move. The argument frame may be omitted if othervar is not an integer (frame then defaults to `1`). For each othervar argument, the upvar command makes the variable by that name in the local scope identified by the frame offset accessible in the current procedure by the name given in the corresponding myvar argument. The variable named by othervar need not exist at the time of the call; it will be created the first time myvar is referenced, just like an ordinary variable. The upvar command is only meaningful from within user defined command bodies. Neither othervar nor myvar may refer to an element of an array. The upvar command returns an empty string. The upvar command simplifies the implementation of call-by-name procedure calling and also makes it easier to build new control constructs as TH1 commands. For example, consider the following procedure:

```
proc incr {name} {
    upvar $name x
    set x [expr $x+1]
}
```

incr is invoked with an argument giving the name of a variable, and it adds one to the value of that variable.

9.6 TH1 commands, scripts and program flow

In TH1 there is actually no distinction between commands (often known as 'statements' and 'functions' in other languages) and "syntax". There are no reserved words (like if and while) as exist in C, Java, Python, Perl, etc. When the TH1 interpreter starts up there is a list of built-in, known commands that the interpreter uses to parse a line. These commands include for, set, puts, and so on. They are, however, still just regular TH1 commands that obey the same syntax rules as all TH1 commands, both built-in, and those that you create yourself with the proc command.

9.6.1 Commands revisited

Like Tcl, TH1 is build up around commands. A command does something for you, like outputting a string, computing a math expression, or generating HTML to display a widget on the screen.

Commands are a special form of list. The basic syntax for a TH1 command is:

```
command arg1 arg2 arg3 ...
```

The command is either the name of a built-in command or a TH1 procedure.

White space is used to separate the command name and its arguments, and a newline character or semicolon is used to terminate a command. TH1 comments are lines with a # character at the beginning of the line, or with a # character after the semicolon terminating a command.

9.6.2 Scripts

Normally, control in TH1 flows from one command to the next. The next command is either in the same list (if the current command is terminated with a semicolon) or in the next input line. A TH1 program is thus a TH1 list of commands.

Such a list of commands is referred to as a "script". A script is hence a self contained code fragment containing one or more commands. The commands in a script are analogous to statements in other programming languages.

Some commands take one or more scripts as arguments and run those scripts zero or more times depending on the other arguments. For example, the if command executes either the then script or the else script once, depending on the if expression being true or false. The command that takes a script will perform the normal grouping and substitution as part of executing the script.

Note that the script always needs to be enclosed in curly brackets to prevent substitution taking place twice: once as part of the execution of the top level command and once again when preparing the script. Forgetting to enclose a script argument in curly brackets is common source of errors.

A few commands (return, error, break and continue) immediately stop execution of the current script instead of passing control to the next command in the list. Control is instead returned to the command that initiated the execution of the current script.

9.6.3 Command result codes

Each command produces two results: a result code and a string. The code indicates whether the command completed successfully or not, and the string gives additional information. The valid codes are defined in `th.h`, and are:

TH1 programmers do not normally need to think about return codes, since TH1_OK is almost always returned. If anything else is returned by a command, then the TH1 interpreter immediately stops processing commands and returns to its caller. If there are several nested invocations of the TH1 interpreter in progress, then each nested command will usually return the error to its caller,

until eventually the error is reported to the top-level application code. The application will then display the error message for the user.

In a few cases, some commands will handle certain "error" conditions themselves and not return them upwards. For example, the for command checks for the TH1_BREAK code; if it occurs, for stops executing the body of the loop and returns TH1_OK to its caller. The for command also handles TH1_CONTINUE codes and the procedure interpreter handles TH1_RETURN codes. The catch command allows TH1 programs to catch errors and handle them without aborting command interpretation any further.

9.6.4 Flow control commands

The flow control commands in TH1 are:

```
if expr1 body1 ?elseif expr2 body2? ? ?else? bodyN?

for init condition incr script

break    ?value?

continue ?value?

error    ?value?

catch script ?varname?
```

Below each command is discussed in turn

The if command has the following syntax:

```
if expr1 body1 ?elseif expr2 body2? ? ?else? bodyN?
```

The `expr` arguments are expressions, the body arguments are scripts and the elsif and else arguments are keyword constant strings. The if command optionally executes one of its body scripts.

The `expr` arguments must evaluate to an integer value. If it evaluates to a non-zero value the following body script is executed and upon return from that script processing continues with the command following the if command. If an expr argument evaluates to zero, its body script is skipped and the next option is tried. When there are no more options to try, processing also continues with the next command.

The if command returns the value of the executed script, or "0" when no script was executed.

The for command has the following syntax:

```
for init condition incr body
```

The `init`, `incr` and `body` arguments are all scripts. The `condition` argument is an expression yielding an integer result. The for command is a looping command, similar in structure to the C for statement.

The `for` command first invokes the TH1 interpreter to execute init. Then it repeatedly evaluates condition as an expression; if the result is non-zero it invokes the TH1 interpreter on body, then invokes the TH1 interpreter on incr, then repeats the loop. The command terminates when test evaluates to zero.

If a `continue` command is invoked within execution of the body script then any remaining commands in the current execution of body are skipped; processing continues by invoking the TH1 interpreter on incr, then evaluating condition, and so on. If a break command is invoked within body or next, then the for command will return immediately. The operation of break and continue are similar to the corresponding statements in C.

The `for` command returns an empty string.

The `break` command has the following syntax:

```
break ?value?
```

The `break` command returns immediately from the current procedure (or top-level command), with value as the return value and TH1_BREAK as the result code. If value is not specified, the string "break" will be returned as result.

The continue command has the following syntax:

```
continue ?value?
```

The `continue` command returns immediately from the current procedure (or top-level command), with value as the return value and TH1_CONTINUE as the result code. If value is not specified, the string "continue" will be returned as result.

The `error` command has the following syntax:

```
error ?value?
```

The `error` command returns immediately from the current procedure (or top-level command), with value as the return value and TH1_ERROR as the result code. If value is not specified, the string "error" will be returned as result.

The `catch` command has the following syntax:

```
catch script ?varname?
```

The `catch` command may be used to prevent errors from aborting command interpretation. The catch command calls the TH1 interpreter recursively to execute script, and always returns a TH1_OK code, regardless of any errors that might occur while executing script.

The return value from `catch` is a decimal string giving the code returned by the TH1 interpreter after executing script. This will be `0`(TH1_OK) if there were

no errors in command; otherwise it will have a non-zero value corresponding to one of the exceptional result codes. If the varname argument is given, then it gives the name of a variable; catch sets the value of the variable to the string returned from running script (either a result or an error message).

9.6.5 Creating user defined commands

The `proc` command creates a new command. The syntax for the `proc` command is:

```
proc name args body
```

The `proc` command creates a new TH1 command procedure, name, replacing any existing command there may have been by that name. Whenever the new command is invoked, the contents of body will be executed by the TH1 interpreter.

The parameter `args` specifies the formal arguments to the procedure. It consists of a list, possibly empty, each of whose elements specifies one argument. Each argument specifier is also a list with either one or two fields. If there is only a single field in the specifier, then it is the name of the argument; if there are two fields, then the first is the argument name and the second is its default value. Curly brackets and backslashes may be used in the usual way to specify complex default values.

The `proc` command returns the null string.

9.6.6 Execution of user defined commands

If a command is a user defined command (i.e. a command created with the proc command), then the TH1 interpreter creates a new local variable context, binds the formal arguments to their actual values (i.e. TH1 uses call by value exclusively) and loads the body script. Execution then proceeds with the first command in that script. Execution ends when the last command has been executed or when one of the returning commands is executed. When the script ends, the local variable context is deleted and processing continues with the next command after the user defined command.

More in detail, when a user defined command is invoked, a local variable is created for each of the formal arguments to the procedure; its value is the value of corresponding argument in the invoking command or the argument's default value. Arguments with default values need not be specified in a procedure invocation. However, there must be enough actual arguments for all the formal arguments that don't have defaults, and there must not be any extra actual arguments.

There is one special case to permit procedures with variable numbers of arguments. If the last formal argument has the name args, then a call to the procedure may contain more actual arguments than the procedure has formals. In this case,

all of the actual arguments starting at the one that would be assigned to args are combined into a list (as if the list command had been used); this combined value is assigned to the local variable args.

When body is being executed, variable names normally refer to local variables, which are created automatically when referenced and deleted when the procedure returns. One local variable is automatically created for each of the procedure's arguments. Global variables can be accessed by using the :: syntax.

When a procedure is invoked, the procedure's return value is the value specified in a return command. If the procedure doesn't execute an explicit return, then its return value is the value of the last command executed in the procedure's body. If an error occurs while executing the procedure body, then the procedure as a whole will return that same error.

The syntax for the `return` command is:

```
return ?-code code? ?value?
```

The optional argument pair `-code code` allows to change the return status code from the default of TH1_OK to another status code. This code has to be specified with its numeric value.

9.6.7 Special commands

TH1 includes three core commands that assist with working with commands. They are:

```
breakpoint args

rename oldcmd newcmd

uplevel ?level? script
```

The `breakpoint` command does nothing. It is used as placeholder to place breakpoints during debugging.

The `rename` command renames a user defined or a built-in command. The old name is removed and the new name is inserted in the interpreter's command table.

The `uplevel` command executes a command in the variable scope of a command higher up in the call chain. The script argument is evaluated in the variable scope indicated by level. The `uplevel` command returns the result of that evaluation. If level is an integer, then it gives a distance (up the procedure calling stack) to move before executing the command. If level is omitted then it defaults to '1'.

For example, suppose that procedure a was invoked from top-level, and that it called b, and that b called c. Suppose that c invokes the uplevel command. If level is `1` or omitted, then the command will be executed in the variable context

of b. If level is `2` then the command will be executed in the variable context of a. If level is `3` then the command will be executed at top-level (i.e. only global variables will be visible).

The `uplevel` command causes the invoking procedure to disappear from the procedure calling stack while the command is being executed. In the above example, suppose c invokes the command

```
uplevel 1 {set x 43; d}
```

where d is another TH1 procedure. The set command will modify the variable x in the context of b, and d will execute at level 3, as if called from b. If it in turn executes the command

```
uplevel {set x 42}
```

then the set command will modify the same variable x in the context of b context: the procedure c does not appear to be on the call stack when d is executing.

9.7 Working with strings

TH1 provides the `string` command to facilitate working with strings. The `string` command is a single command with seven subcommands, identified by the first argument. The first argument serves no purpose other than to identify the subcommand. If the first argument does not match a subcommand, an error is thrown.

The seven `string` subcommands are:

```
string length string

string compare string1 string2

string first needle haystack ?startindex?

string last needle haystack ?startindex?

string range string first last

string repeat string count

string is alnum string
```

The `string length` subcommand takes one parameter, which is a string. It returns the decimal string with the length of the string. As TH1 uses a single byte character encoding the string size is both the size in characters and in bytes.

The `string compare` subcommand performs a character-by-character comparison of argument strings string1 and string2 in the same way as the C strcmp

procedure. It returns a decimal string with value -1, 0, or 1, depending on whether string1 is lexicographically less than, equal to, or greater than string2.

The `string first` subcommand searches argument haystack for a sequence of characters that exactly match the characters in argument needle. If found, it returns a decimal string with the index of the first character in the first such match within haystack. If not found, it returns return -1. The optional integer argument startindex specifies the position where the search begins; the default value is 0, i.e. the first character in haystack.

The `string last` subcommand searches argument haystack for a sequence of characters that exactly match the characters in argument needle. If found, it returns a decimal string with the index of the first character in the last such match within haystack. If not found, it returns return -1. The optional integer argument startindex specifies the position where the search begins; the default value is 0, i.e. the first character in haystack.

The `string range` subcommand returns a range of consecutive characters from argument string, starting with the character whose index is first and ending with the character whose index is last. An index of zero refers to the first character of the string. last may be end to refer to the last character of the string. If first is less than zero then it is treated as if it was zero, and if last is greater than or equal to the length of the string then it is treated as if it were end. If first is greater than last then an empty string is returned.

The `string repeat` subcommand returns a string that is formed by repeating the argument string for count times. The argument count must be an integer. If count is zero or less the empty string is returned.

The `string is alnum` subcommand tests whether the argument string is an alphanumeric string, i.e. a string with only alphanumeric characters. It returns a decimal string with value 1 if the string is alphanumeric, and with value 0 it is not.

9.8 Working with lists

The list is the basic TH1 data structure. A list is simply an ordered collection of items, numbers, words, strings, or other lists. For instance, the following string is a list with four items. The third item is a sub-list with two items:

```
{first second {a b} fourth}
```

TH1 has three core commands to work with lists:

```
list ?arg1 ?arg2? ...?

lindex list index

llength list
```

The `list` command returns a list comprising all the args. Braces and backslashes get added as necessary, so that the lindex command may be used on the result to re-extract the original arguments. For example, the command

```
list a b {c d e} {f {g h}}
```

will return

```
a b {c d e} {f {g h}}
```

The `lindex` command treats argument list as a TH1 list and returns the element with index number index from it. The argument index must be an integer number and zero refers to the first element of the list. In extracting the element, the lindex command observes the same rules concerning braces and quotes and backslashes as the TH1 command interpreter; however, variable substitution and command substitution do not occur. If index is negative or greater than or equal to the number of elements in value, an empty string is returned.

The `llength` command treats argument list as a list and returns a decimal string giving the number of elements in it.

Chapter 10

Chiselapp

Chiselapp is a website that is like github but hosts Fossil repositories. This way you can have your repository on a internet accessible host. This works like the Apache hosted repositories described in [par:Server-hosted] but Chiselapp supplies the server and the host is on the internet not a local area network.

After you set up a FREE account you can then push your repository to them and zap you are on the internet at:

https://chiselapp.com/user/<your account>/repository/<Project>

10.1 Create an account

Your first step is to create an account. The Chiselapp home page is:

Fill out the form with your information in my case I used my name and my Gmail account to set it up and my account is jschimpf.

10.2 Repositories

You can create repositories on the site and then copy one of your local repositories there. You have the choice of making public or private repositories. Public are visible to anyone visiting the site and private are visible only to you. In addition you do the standard Fossil assignment of users and privileges so once someone accesses the repository they only can do what you allow.(Figure:[fig:User-Configuration])

The rest of this section will show how I am putting the repository NULLMODEM http://chiselapp.com/user/jschimpf/repository/NULMODEM/index on to ChiselApp.

10.2.1 Create Repository

The first step is to pick the option Create New Repository on your login page. This will give you the following screen:

So I fill in the name as NULLMODEM and I put in my repository password but what is Project Code ? Here you have to run Fossil to extract this information from your repository as follows:

```
500 FOSSIL> fossil info -R NULMODEM.fossil
```

The form is now filled in and we can create the repository

and you get this:

10.2.2 Moving data

The next step is moving the repository on my disk to Chiselapp. This is done via a push command in Fossil. I am doing this command in the directory where NULMODEM.fossil lives so I don't need to type a path. Note the command is complete but I'm hiding my password when you do this type you password in full where I have .

```
501 FOSSIL> fossil push https://jschimpf:<passwd>@chiselapp.com/user/jschimp
```

10.3 Fixing Data

When you go to your new repository things are a bit messed up. You get:

Whoa where's all my nice formatting and pointers to my documentation ? They are hidden and you have to get them back:

Go to the timeline view:

And see the top checkin that is the initial empty check-in this is an artifact of how Chiselapp creates your repository and you have to SHUN it

You will then be taken to another page where it will ask you if you really want to do this and pick Shun again.

Not quite there yet, you have to log into the project (Remember your name and password from Figure:[fig:Filled-in-form]) log in with this information and go to the Admin->Configuration page. Put in the same information you had on your local repository and ZAP your home page is back.

10.4 Final Fixes

The home page is now restored and we are ready to go.

The only problem now is the System Manual link doesn't work. The original was:

The fix is to change the link to just /doc/tip/DOCS/NULMODEM.pdf

10.5 Syncing

When you created the repository on ChiselApp you used this command:

```
fossil push https://jschimpf:<passwd>@chiselapp.com/user/jschimpf/repository/NULMODEM
```

when you did that the repository was created but it did not sync with your local one. This is probably not a good idea as you want the ChiselApp repository to stay up to date. If you leave off the –once then it will sync locally. If it isn't synching now and you want to reverse this at any time just type:

```
fossil push https://jschimpf:<passwd>@chiselapp.com/user/jschimpf/repository/NULMODEM
```

i.e. the command without the `-once` and you are syncing again.

10.6 Final Result

Now you can go to https://chiselapp.com/user/jschimpf/repository/NULMODEM and view the repository and do what an anonymous user can do.

Chapter 11

Advanced uses

Fossil can be used in various ways. It is flexible and can be adapted and customized. Here are some examples for advanced usage.

11.1 Additional tables in the repository

It is possible to add more tables to the sqlite database that Fossil uses to store the repository into. If you add tables having names starting with `fx_` they will not be not be touched and also not deleted during a rebuild of the database.

However, these tables won't be synched and not be copied when cloning the database. They are only meant for local data. Still, you could add those tables in a server database and use the data to serve some specific content of the repository to all users accessing the repository over the web frontend.

Chapter 12

What's next ?

12.1 Learning more

This book so far has covered how to use the many features of Fossil and has, I hope, interested you in using it. The question "what's next" now comes up. First go to the Fossil website. While there you can go to the Docs link and view the list pages. There are all sorts of topics covered in depth. If that still doesn't help, you can join the Fossil forum and ask a question. I have found the forum contributors, including members of the development team, to be very helpful and have had my questions asked very quickly.

In the Fossil forum you will see suggestions for changes to be made to Fossil, some of these are accepted very quickly and will appear within hours in the Fossil source code. Others engender long discussions and it is interesting to read the pros and cons of suggested changes.

Fossil is an evolving program but if you get a version that has all the features you need you can stick with that version as long as you like. Going to a new version though is simple and just requires a rebuild of your current repositories. The developers have been very careful to preserve the basic structure so it is easy and safe to switch versions.

12.2 Contributing

Finally if you wish to contribute to the project there are many things to do, often triggered by forum comments.

Chapter 13

Revision history

Use the following table to track a history of this document's revisions. An entry should be made into this table for each version of the document.

Version	Author	Description	Issue Date
0.1	js	Initial Version	24-Apr-2010
0.2	js	Finishing up Single User Chapter	27-Apr-2010
0.3	js	Working on Introduction Chapter	30-Apr-2010
0.4	js	Adding multiuser chapter	1-May-2010
0.5	mn	Adding editorial corrections [ebf40b842a]	4-May-2010
0.6	js	Adding Command sections [e11399d575]	8-May-2010
0.7	js	English & spelling corrections	19-May-2010
0.8	js	Spelling fixes	30-May-2010
0.9	ws	Using Fossil merge [db6c734300]	2-Jun-2010
1.0	js/ws	Put Fossil merge first in handling fork	3-Jun-2010
1.1	mn	Fixes in multiple user chapter [e8770d3172]	4-Jun-2010
1.2	js	Start advanced use chapter [2abc23dae5]	4-Jun-2010
1.3	mn	English corrections Chapter 1 [8b324dc900]	5-Jun-2010
1.4	mn	Sections 2.1 & 2.2 corrections [0b34cb6f04]	7-Jun-2010
1.5	js	Move close leaf to adv use [2abc23dae5]	7-Jun-2010
1.6	js	Convert Advanced chapter to forks and branching	13-Jun-2010
1.7	js/tr	Add note about IP port usage [a62efa8eba]	8-Jul-2010
1.71	javelin	Check on mispelling section 1.1 [637d974f62]	15-Sep-2011
1.72	anon	Fix absolute path in image regs [d54868853b]	15-Sep-2011
1.73	anon	Fix fossil create section 2.2.5 [36772d90a5]	15-Sep-2011
1.74	anon	Push/Pull described incorrectly [1b930fced6]	15-Sep-2011
1.75	arnel	Commands might be changed [4aaf1f78bb]	15-Sep-2011
2.0	FvD	Updated and matched to fossil 1.25	March 2013
3.0	ctp	Migrated from Lyx to Markdown	Nov-2020

Version	Author	Description	Issue Date
3.x	tb	Aim to cover fossil 2.18 ... still much to do	Feb-2022