Syntax Highlighting Modular Build System for NoWeb

Thomas J. Moore

Version 3.13.161
November 22, 2012

Abstract

This document describes, implements, and is built using a modular "literate programming[1]" build system with syntax highlighting, based on Norman Ramsey's NoWeb[2], GNU[3]make, and a number of other freely available support tools.

Contents

1 Overview

Most of my old personal programming projects started out as a general idea, along with numerous scattered pieces of paper with notes detaling some particular aspects of the design. Most of my professional programming projects instead started with formal design documents containing similar information. Over time, the

[1] http://www.literateprogramming.com/articles.html is a good starting point.
[2] http://www.eecs.harvard.edu/~nr/noweb/
[3] http://www.gnu.org

design documents and notes were lost and/or forgotten, and/or they diverged from the actual product. In any case, the organization of any design documentation rarely corresponds with the actual code, as efficient code follows much different rules than efficient human communication. Literate programming provides two main features that alleviate these problems: the design document is kept with the source code, and the code can be written in the order of the design document, to be reordered later for more efficient compilation.

Literate programming is a method of programming devised and named by Donald Knuth in 1983, along with the tools to help implement that method (Web). It mainly emphasizes writing a human-readable document with embedded code rather than the other way around. Over the years, as others have adopted and adapted the style, the same term has been used to mean slightly different things, and the tools have emphasized different features. To me, the important features are mixing of documentation and code, and reordering of code chunks to match the text, rather than the other way around. Systems which do not display the source code and documentation at the same time (e.g. Doxygen, perl's pod) and/or do not provide code reordering (e.g. dtx, lgrind, code2html) do not support literate programming. Knuth would probably be even pickier, and reject my use of large code chunks, short variable names for temporary variables, long sequences of multiple code chunks, and multiple source files and programs in a single document, as well as my lack of a language-specific symbol cross reference and index (in addition to the chunk index).

Knuth felt that programs should be pleasant to read, like documents which happen to have executable code in them. He believed that his method would allow creation of programs which are by their nature easier to maintain and less error-prone. Whether or not a document is pleasant to read is always very dependent on the writer (and, to a lesser extent, the reader). Easier maintenance and fewer bugs are but a pipe dream: people will in fact be even more tempted to ignore the source code in favor of the commentary, thus missing obvious bugs. Code reordering, and especially small code chunks may also make it difficult for some to follow the code to find out what *really* gets executed. My own reasons for doing this are mainly to keep my scattered design notes in one place (the source) and to keep the design document in sync with the implementation (since they are the same thing).

Creating programs as documents using literate programming principles requires a system that supports easy-to-write documentation, and code that is both structured like the documentation and actually part of the documentation. For simple programs, a set of tools which support only a single source language and a single program file can be used. As programs become more complicated, though, the number of source files may increase, and when designing entire products, the number of source languages may increase as well. Current common public documentation standards also require a system which produces at least HTML and PDF output. After long searching, I have not found any systems which meet these requirements. Instead, the closest is the NoWeb package. It supports LaTeX and HTML output, and an arbitrary number of output files and source languages. However, it does little to make the code look like part of the documentation. The code in this document serves to provide a build system that includes syntax-highlighted code, at the very least.

While the build system could simply be a carefully crafted shell script, a makefile is used instead. This allows use of implicit rules and the system-wide `CC` and `CFLAGS` defaults. The default chunk is set to the makefile, requiring no chunk name to extract. To build, place the NoWeb source files into their own directory (optionally extracting them from the document you are reading first), extract the makefile, and make using GNU make:

```
# if your PDF viewer supports attachments, save the attachment
# otherwise, use pdfdetach:
pdfdetach -saveall build.pdf
# or pdftk:
pdftk build.pdf unpack_files output .
# then extract the makefile and build
notangle -t8 build.nw > makefile
make install
```

If NoWeb is not available, but perl is, the following code can be copied and pasted into a text file using a reasonable document viewer and made executable as an imperfect, but adequate `notangle` replacement:

```
#!/bin/sh
#!perl
# This code barely resembles noweb's notangle enough to work for simple builds
eval 'exec perl -x -f "$0" "$@"'
  if 0;
my $chunk = '*';  my $tab = 0;

while($#ARGV > 0 && $ARGV[0] =~ /^-.*/) {
  if($ARGV[0] =~ /^-R/) {
    $chunk = $ARGV[0];
    $chunk =~ s/-R//;
  } elsif($ARGV[0] =~ /^-t(.*)/) {
    $tab = $1;
  }
  shift @ARGV;
}
my ($inchunk, %chunks);
while(<>) {
  if(/^<<(.*)>>=$/) {
    $inchunk = $1;
  } elsif(/^@( |$)/) {
    $inchunk = "";
  } elsif($inchunk ne "") {
    $chunks{$inchunk} .= $_;
  }
}
my $chunkout = $chunks{$chunk};
while($chunkout =~ /(?:^|\n)(|[^\n]*[^\n@])(?:<<((?:[^>\n]|>[^>\n])*)>>)/) {
  my $cv = $chunks{$2};
  my $ws = $1;
  $cv =~ s/\n$//;
  $ws =~ s/\S/ /g;
  $cv =~ s/\n/$&$ws/g;
  $chunkout =~ s/(^|[^@])(<<(([^>\n]|>[^>\n])*)>>)/$1$cv/;
}
$chunkout =~ s/@(<<|>>)/\1/g;
$chunkout =~ s/((^|\n)\t*) {$tab}/$1\t/g if($tab gt 0);
print $chunkout;
```

As noted below, additional makefile components are created in separate files. Additional build configuration can be done in `makefile.config` before installing (in particular, the install locations). This file can be generated using either `make makefile.config` or `make -n`. On the other hand, to avoid having to modify this file after cleaning, `makefile.config.local` can be created for this purpose instead.

1.1 Makefile

The makefile is for building the support files, executable binaries, and printable source code (HTML and PDF), maybe installing them, and cleaning up afterwards. Configuration information, variable definitions, and miscellaneous rules are extracted from not only this NoWeb file, but also any others that use it. The makefile itself can be used to include that information, but to simplify makefile extraction, all of that information is kept in separate files. Modules that wish to extend the makefile can do so using ⟨*makefile.config* 5c⟩ for user-adjustable variables, ⟨*makefile.vars* 5d⟩ for any other variable definitions, and ⟨*makefile.rules* 5b⟩ for new rules. Some configuration variables may need values before the configuration file is built, though, so defaults are included in the makefile, to be overridden by the configuration file. Note that any variable which does not depend on unknowns (noweb chunks or variables not yet defined) should be defined using GNU make's "simply expanded" variable flavor (i.e., `:=` instead of plain `=`) for improved performance. This has a disadvantage in that overriding simply defined variables will require overriding any of its dependents as well, but the performance improvement is usually worth it.

I have made some adjustments to the ordering of this in order to support Daniel Pfeiffer's Makepp[4], a

[4] http://makepp.sourceforge.net

GNU make clone which adds MD5 checksums as a file change detection mechanism. This magically fixes the problem of rebuilding every time, at the expense of dealing with Makepp-specific issues. The first of these is that include files cannot be automatically (re)built unless they appear after the rules which create them. This means that the two variables which control their creation can no longer come from the configuration file. They can only be set using the command line. Another, related issue is that even though Makepp solves the rebuild problem for most files, it makes the problem worse for the makefile and its include files. These are unconditionally rebuilt every single time makepp is invoked. The use of Makepp is detected by checking for the existence of a non-empty `$(MAKEPP_VERSION)`.

4a ⟨* 4a⟩≡

```
⟨makefile 4b⟩
```

4b ⟨*makefile* 4b⟩≡ (4a) 14a▷

```
# See makefile.config for variables to override
# Put local config in makefile.config or makefile.config.local
⟨Common NoWeb Warning 4c⟩

# default rule
all: misc bin doc

ifneq ($(MAKEPP_VERSION),)
# not much point, since it doesn't filter out #line directives
# and therefore most changes to NoWeb source will result in file changes
signature c_compilation_md5
endif

⟨Defaults for makefile.config 6a⟩

# This is the right place to include them, but makepp can't handle it
#-include makefile.config
-include makefile.config.local

⟨Build variables for makefile includes 6f⟩
⟨Build rules for makefile includes 6b⟩

-include makefile.config
# reinclude to ensure overrides
-include makefile.config.local

-include makefile.vars

# keep intermediate files
.SECONDARY:
# mark a few phonies - not really necessary
.PHONY: all misc bin doc install clean distclean

bin: $(LIB_FILES) $(EXEC_FILES)

doc: $(DOC_FILES)

misc: $(MISC_FILES)

-include makefile.rules
```

4c ⟨*Common NoWeb Warning* 4c⟩≡ (4 5 19d 22d 25a 32e 36d 46b 49f 50e)

```
# GENERATED FILE:    DO NOT EDIT OR READ THIS FILE
# Instead, read or edit the NoWeb file(s) from which this was generated,
# listed below.    Copyright notice and license terms can be found there.
# $Id: build.nw 161 2012-11-23 02:14:50Z darktjm $
```

5a ⟨*Sources* 5a⟩≡

```
$Id: build.nw 161 2012-11-23 02:14:50Z darktjm $
```

5b ⟨*makefile.rules* 5b⟩≡ 8a▷

```
⟨Common NoWeb Warning 4c⟩
install: misc bin
        mkdir -p $(DESTDIR)$(BIN_DIR)
        for x in $(EXEC_FILES); do \
           rm -f $(DESTDIR)$(BIN_DIR)/$$x; \
           cp -p $$x $(DESTDIR)$(BIN_DIR); \
         done
        ⟨Install other files 5e⟩

clean:
        ⟨Clean temporary files 9c⟩
        rm -f makefile.{config,vars,rules}

distclean: clean
        ⟨Clean built files 9b⟩
        rm -rf .makepp
        ⟨Remove makefile 6c⟩
```

5c ⟨*makefile.config* 5c⟩≡ 5g▷

```
# Installation prefix to apply to all install locations
DESTDIR:=
# Installation directory for binaries
BIN_DIR:=/usr/local/bin
```

5d ⟨*makefile.vars* 5d⟩≡ 8c▷

```
⟨Common NoWeb Warning 4c⟩
MAKEFILES:=makefile makefile.config makefile.vars makefile.rules
EXEC_FILES = ⟨Executables 11a⟩

LIB_FILES = ⟨Libraries 14c⟩

DOC_FILES = ⟨Source Code Documentation Files 20d⟩

MISC_FILES = ⟨Plain Files 5f⟩
```

5e ⟨*Install other files* 5e⟩≡ (5b)

5f ⟨*Plain Files* 5f⟩≡ (5d)

```
\
```

The makefile can make itself, as well. This is dependent only on its source file; there is little point in making this dependent on the included files, as they will be automatically rebuilt as needed, anyway. A quick check before writing out the file ensures that a blank or otherwise seriously invalid makefile will never be created due to errors in the source file.

Note that Makepp has issues with recursive make invocation for the verification of the makefile, so this step is skipped.

5g ⟨*makefile.config* 5c⟩+≡ ◁5c 6d▷

```
# The name of the file containing the makefile
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#BUILD_NOWEB:=build.nw
```

6a ⟨*Defaults for* `makefile.config` 6a⟩≡ (4b) 6e▷

```
BUILD_NOWEB=build.nw
```

6b ⟨*Build rules for makefile includes* 6b⟩≡ (4b) 8b▷

```
makefile: $(BUILD_NOWEB)
        notangle -t8 -R$@ $(BUILD_NOWEB) 2>/dev/null | grep -v '^$$' >/dev/null
        @#notangle -t8 -R$@ $(BUILD_NOWEB) 2>/dev/null | \ #
        @#      env -i $(MAKE) -n -f- /dev/null >/dev/null
        -notangle -t8 -R$@ $(BUILD_NOWEB) > $@
```

6c ⟨*Remove makefile* 6c⟩≡ (5b)

```
rm -f makefile
@echo
@echo Regenerate the makefile with:
@echo notangle -t8 $(BUILD_NOWEB) \> makefile
```

Generating the other files requires the ability to correctly assemble them from all the other NoWeb files.

6d ⟨*makefile.config* 5c⟩+≡ ◁5g 7c▷

```
# The name of the source files
# Any time you change this, makefile.* should be removed and rebuilt
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#NOWEB:=$(wildcard *.nw)
```

6e ⟨*Defaults for* `makefile.config` 6a⟩+≡ (4b) ◁6a 13e▷

```
NOWEB:=$(wildcard *.nw)
```

1.2 Merging Sources

From these files, an order of building must be derived. This is done using a dependency tree created from special comments in the source files. These comments are at the beginning of a line, and are of the form `%%% requires` *X*, where *X* is either the name of a NoWeb file or the name with the `.nw` removed. An explicit `\input{`*X*`.nw` creates a dependency as well. Top-level build files are those on which no others depend. The tree order consists of the top-level files, followed by their direct dependencies, in the order found in those files, followed by those files' direct dependencies, and so forth, with no file repeated.

Finding the dependency directives requires `egrep`, and removing the pattern requires `sed`. Due to the sloppiness of the patterns and other parsing, no file names can contain colons or spaces. This is all done using GNU make's internal functions so that the results can be easily used in other rules.

6f ⟨*Build variables for makefile includes* 6f⟩≡ (4b) 7a▷

```
NOWEB_DEPS:=$(shell egrep '^%%% requires |^\\input{.*\.nw}$$' $(NOWEB) /dev/null | \
                    sed -e 's/%%% requires //;s/\\input{\(.*\)}$$/\1/' \
                        -e 's/[[:space:]]*\(%.*\)*$$//;s/:/ /')
```

After the dependencies are found, they are separated into files which depend on others (NOWEB_UPPER), and files which are depended on (NOWEB_LOWER). The files whilch are depended on may be specified without the .nw extension, so the filesystem is checked for the file, and .nw is added if the file does not exist. If it still does not exist after tacking on .nw, the missing file is an error.

7a ⟨*Build variables for makefile includes* 6f⟩+≡ (4b) ◁6f 7b▷

```
# return rest of words of parm 2, starting at parm 1
rest = $(wordlist $1,$(words $2),$2)

# return every other word of parm, starting with first
ret_other = $(word 1,$1) $(if $(word 3,$1),$(call ret_other,$(call rest,3,$1)))

NOWEB_UPPER:=$(call ret_other,$(NOWEB_DEPS))

NOWEB_LOWER_BASE:=$(call ret_other,$(wordlist 2,$(words $(NOWEB_DEPS)),$(NOWEB_DEPS)))

NOWEB_LOWER:=$(foreach f,$(NOWEB_LOWER_BASE),$(if $(wildcard $f),$f,\
                                              $(if $(wildcard $f.nw),$f.nw, \
                                              $(error $f and $f.nw not found))))
```

Next, the tree is traversed, from top to bottom. The top is simply the list of files on which no other files depend. There must be at least one file at the top, or nothing will work correctly. Then, each file in the list is checked for as-yet unfulfilled dependencies to tack on. No dependency may appear before files upon which it depends, so the dependecies are repeatedly tacked onto the start of the list and stripped from the rest until the tree settles.

7b ⟨*Build variables for makefile includes* 6f⟩+≡ (4b) ◁7a 13g▷

```
NOWEB_HIGHEST:=$(filter-out $(NOWEB_LOWER),$(NOWEB))

$(if $(NOWEB_HIGHEST),,$(error Invalid dependency tree))

# return words from parm #3 in positions of parm #2 which match parm #1
match_words = $(if $(filter $1,$(word 1,$2)),$(word 1,$3)) \
  $(if $(word 2,$2),$(call match_words,$1,$(call rest,2,$2),$(call rest,2,$3)))

# return only unique words in parm, keeping only first occurrence
uniq = $(if $1,\
  $(firstword $1) $(call uniq,$(filter-out $(firstword $1),$(call rest,2,$1))))

# tack dependencies to left
predeps = $(call uniq, $(foreach f,$1,\
                $(call match_words,$f,$(NOWEB_UPPER),$(NOWEB_LOWER))) $1)

# true if lists not equal
listne = $(strip $(if $1,$(if $2,$(if $(filter $(word 1,$1),$(word 1,$2)), \
                $(call listne,$(call rest,2,$1),$(call rest,2,$2)), \
                  y1), y2), $(if $2, y3)))

# expand dependencies until there are no more
tree = $(if $(call listne,$1,$(call predeps,$1)), \
           $(call tree,$(call predeps,$1)), $1)

NOWEB_ORDER:=$(call tree,$(NOWEB_HIGHEST))

ifeq ($(PROJECT_NAME),)
PROJECT_NAME:=$(subst .nw,,$(firstword $(NOWEB_HIGHEST)))
endif
```

7c ⟨*makefile.config* 5c⟩+≡ ◁6d 11f▷

```
#Set to override the automatically determined project name
#PROJECT_NAME=
```

8a ⟨*makefile.rules* 5b⟩+≡ ◁5b 9a▷

```
prtree:
        @echo Project: $(PROJECT_NAME)
        @echo Deps: $(NOWEB_DEPS)
        @echo Highest: $(NOWEB_HIGHEST)
        @echo Upper: $(NOWEB_UPPER)
        @echo Lower: $(NOWEB_LOWER)
        @echo Order: $(NOWEB_ORDER)
```

So, to generate a file, all of these NoWeb files are concatenated, in reverse order, and passed into `notangle`. The makefile components in particular need to be checked for errors in mostly the same way as the main makefile.

8b ⟨*Build rules for makefile includes* 6b⟩+≡ (4b) ◁6b

```
makefile.config: $(NOWEB_ORDER)
        notangle -t8 -R$@ $^ 2>/dev/null | grep -v '^$$' >/dev/null
        @#notangle -t8 -R$@ $^ 2>/dev/null | env -i $(MAKE) -n -f- /dev/null >/dev/null
        -notangle -t8 -R$@ $^ > $@

makefile.vars: makefile.config
        notangle -t8 -R$@ $(NOWEB_ORDER) 2>/dev/null | grep -v '^$$' >/dev/null
        @#notangle -t8 -Rmakefile.config -R$@ $(NOWEB_ORDER) 2>/dev/null | \ #
        @#          env -i $(MAKE) -n -f- /dev/null >/dev/null
        -notangle -t8 -R$@ $(NOWEB_ORDER) > $@

makefile.rules: makefile.vars
        notangle -t8 -R$@ $(NOWEB_ORDER) 2>/dev/null | grep -v '^$$' >/dev/null
        @#notangle -t8 -Rmakefile.config -Rmakefile.vars -R$@ \ #
        @#                                                $(NOWEB_ORDER) 2>/dev/null | \ #
        @#          env -i $(MAKE) -n -f- /dev/null >/dev/null #
        -notangle -t8 -R$@ $(NOWEB_ORDER) > $@
```

Building plain files is done the same way, but without the complicated checks, assuming that no additional processing needs to be done. For files where additional processing is necessary, additional dependencies on the `misc` target, as well as ⟨*Install other files* 5e⟩ can be used to add files with special build rules.

Note that ⟨*Plain Files* 5f⟩ is intended for installable targets. For plain files generated as part of the build, use ⟨*Plain Build Files* 8d⟩ instead. For a subtle change, any files generated by means other than `notangle` can be added to ⟨*Plain Built Files* 8e⟩ in order to save making a clean rule for it.

8c ⟨*makefile.vars* 5d⟩+≡ ◁5d 9e▷

```
MISC_TEMP_FILES = ⟨Plain Build Files 8d⟩

GENERATED_TEMP_FILES = ⟨Plain Built Files 8e⟩
```

8d ⟨*Plain Build Files* 8d⟩≡ (8c)

```
\
```

8e ⟨*Plain Built Files* 8e⟩≡ (8c)

```
\
```

9a ⟨*makefile.rules* 5b⟩+≡ ◁8a 9d▷

```
$(MISC_FILES) $(MISC_TEMP_FILES): $(NOWEB_ORDER)
        -notangle -R$@ $^ >$@
```

9b ⟨*Clean built files* 9b⟩≡ (5b) 9f▷

```
rm -f $(MISC_FILES)
```

9c ⟨*Clean temporary files* 9c⟩≡ (5b) 12c▷

```
rm -f $(MISC_TEMP_FILES) $(GENERATED_TEMP_FILES)
```

1.3 Additional Features

It may also be useful to build a tar file for building on systems where NoWeb is not present. This is meant to be a convenience, for building binaries only, and not for distribution. That means neither the source documentation nor any means to build the source documentation will be included. Since it is not possible to distinguish between soft links created for building and soft links to other NoWeb files, no attempt will be made to force link dereferencing, either.

9d ⟨*makefile.rules* 5b⟩+≡ ◁9a 9g▷

```
$(PROJECT_NAME)-src.tar.gz: $(BUILD_SOURCE)
        @# needs GNU tar
        tar czf $(PROJECT_NAME)-src.tar.gz $(BUILD_SOURCE)
```

9e ⟨*makefile.vars* 5d⟩+≡ ◁8c 11b▷

```
BUILD_SOURCE=$(NOWEB) $(MAKEFILES) $(MISC_FILES) ⟨Build Source 11e⟩
```

9f ⟨*Clean built files* 9b⟩+≡ (5b) ◁9b 12d▷

```
rm -f $(PROJECT_NAME)-src.tar.gz
```

It may also be useful to get some source code statistics. Code counts include all code chunks, but do not include the chunk name/start line or the terminating `@`. Hidden sections are not included; they are delimited by <!--> and <--> on their own line.

9g ⟨*makefile.rules* 5b⟩+≡ ◁9d 10▷

```
count:
        @for x in $(NOWEB); do \
           echo $${x}:; \
           ⟨Count lines in $$x 9h⟩ \
         done
        @echo "Tangled output:"
        @wc -l $(MISC_FILES) ⟨Files to Count 12b⟩
              </dev/null | sort -k1n
```

9h ⟨*Count lines in* `$$x` 9h⟩≡ (9g)

```
tl='cat $$x | wc -l'; \
bl='grep -c '^[[:space:]]*$$' $$x'; \
hl='sed -n -e '/^<!-->$$/{:a p;n;/^<-->$$/!ba;}' < $$x | wc -l'; \
cl='sed -n -e '/^<<.*>>=/{:a n;/^@[^@]/b;/^@$$/b;/^[[:space:]]*$$/!p;ba;}' \
          -e '/^<!-->$$/{:b n;/^<-->$$/!bb;}' < $$x | wc -l' ; \
echo " Lines: $$tl:"; \
echo "  $$cl code, $$((tl-bl-hl-cl)) doc, $$hl hidden, $$bl blank";
```

Finally, there is a rule to check the consistency of the NoWeb source. Checking the list of chunks which are not specifically referenced requires human intervention, so all root chunks are printed out for review. The list of missing chunks should always be accurate, though. The terminating `@` for code chunks isn't required by the NoWeb syntax when writing multiple consecutive code chunks, but it is required by some of the document converters. The reinsertion code is stupid about unbalanced doc reinsertions, but instead of checking correctly there, it's flagged here.

10 ⟨*makefile.rules* 5b⟩+≡ ◁9g 12a▷

```
.PHONY: check
check:
        @$(NOROOTS) $(NOWEB) | sed 's/<<//;s/>>//;' | sort | while read x; do \
          echo "Root: $$x"; \
          notangle -R"$$x" $(NOWEB) >/dev/null; \
        done
        @for f in $(NOWEB); do \
           lno=1 gotat=y; \
           while IFS= read -r x; do \
             case "$$x" in \
                "<<"*">>"=) \
                   test "$$gotat" || echo "$$f: $$prev not terminated by @"; \
                   gotat=; \
                   prev="$${lno}: $$x";; \
                @|@[^@]*) \
                   test "$$gotat" && echo "$$f: $${lno}: extra @"; \
                   gotat=y; \
             esac; \
             lno=$$((lno+1)); \
           done < "$$f"; \
         done
        @for f in $(NOWEB); do \
           lno=1 std=0; \
           while IFS= read -r x; do \
             case "$$x" in \
                "% Begin-doc "*) \
                  std=$$((std+1)); eval bd$$std='"$${x#* * }"' bl$$std=$$lno; \
                  stp=1; while [ $$stp -lt $$std ]; do \
                    eval bd='"$$bd'$$stp\" bl=\$$bl$$stp; \
                    if [ "$${x#* * }" = "$$bd" ]; then \
                      echo "Recursive Begin-doc $$bd at $$bl/$$lno"; break; \
                    fi; \
                    stp=$$((stp+1)); \
                  done;; \
                "% End-doc "*) \
                  if [ $$std -eq 0 ]; then \
                    echo "$$f: unexpected end-doc at $$lno"; \
                  else \
                    eval bd='"$$bd'$$std\" bl=\$$bl$$std; \
                    if [ "$${x#* * }" != "$$bd" ]; then \
                      echo "$$f: Begin-doc $$bd at $$bl" \
                           "ended by $${x#% } at $$lno"; break; \
                    fi; \
                    std=$$((std-1)); \
                  fi;; \
```

10 ⟨*makefile.rules* 5b⟩+≡ ◁9g 12a▷

```
            esac; \
            lno=$$((lno+1)); \
          done < "$$f"; \
          while [ $$std -gt 0 ]; do \
            eval bd='"$$bd'$$std\" bl=\$$bl$$std; \
            echo "$$f: unended Begin-doc $$bd at $$bl"; \
            std=$$((std-1)); \
          done; \
        done
```

2 Binaries

Building the binaries is pretty simple, using automatic rules. Support is provided here for plain text executable scripts and C executables. In order to ensure a consistent build rule for all C files, the default C-to-executable rule is blanked out. Executables are assumed to have one mainline source, named the same as the executable but with a `.c` extenstion. All local libraries built from local sources are linked into every executable.

Since all C files share the same rule, a hook (`C_POSTPROCESS`) is provided for doing further modifications to the code before compilation. The hook is in the form of a GNU make function call, which takes the target file name as an argument. It is expected to return an appropriate pipeline for that target. Scripts, on the other hand, are easy enough to modify using separate rules.

11a ⟨*Executables* 11a⟩≡ (5d)

```
$(C_EXEC) $(SCRIPT_EXEC) \
```

11b ⟨*makefile.vars* 5d⟩+≡ ◁9e 12h▷

```
SCRIPT_EXEC=⟨Script Executables 12e⟩

C_EXEC=⟨C Executables 12f⟩

NOWEB_CFILES:=$(shell $(NOROOTS) $(NOWEB) | sed -n '/\.c>>/{s/<<//;s/>>//;p;}')
CFILES=⟨C Files 11c⟩

NOWEB_HEADERS:=$(shell $(NOROOTS) $(NOWEB) | sed -n '/\.h>>/{s/<<//;s/>>//;p;}')
HEADERS=⟨C Headers 11d⟩

COFILES=$(patsubst %.c, %.o, $(CFILES))
```

11c ⟨*C Files* 11c⟩≡ (11b)

```
$(NOWEB_CFILES) \
```

11d ⟨*C Headers* 11d⟩≡ (11b)

```
$(NOWEB_HEADERS) \
```

11e ⟨*Build Source* 11e⟩≡ (9e) 12g▷

```
$(SCRIPT_EXEC) \
```

11f ⟨*makefile.config* 5c⟩+≡ ◁7c 13f▷

```
# Set to -L for #line, but lose code indentation
USE_LINE:=-L
```

12a ⟨*makefile.rules* 5b⟩+≡ ◁10 13a▷

```
$(SCRIPT_EXEC): $(NOWEB)
        -notangle -R$@ $(NOWEB_ORDER) >$@
        chmod +x $@

# C_POSTPROCESS can be used to add boilerplate code
$(NOWEB_HEADERS): $(NOWEB)
        -notangle $(USE_LINE) -R$@ $(NOWEB_ORDER) $(call C_POSTPROCESS,$@) >$@

$(NOWEB_CFILES): $(NOWEB)
        -notangle $(USE_LINE) -R$@ $(NOWEB_ORDER) $(call C_POSTPROCESS,$@) >$@

# disable gmake built-in .c->exe rule
ifeq ($(MAKEPP_VERSION),)
%: %.c
endif

%.o: %.c
        $(CC) $(CFLAGS) $(EXTRA_CFLAGS) -c -o $@ $<

%: %.o
        $(CC) -o $@ $< $(LDFLAGS) -L. $(LOCAL_LIBS) $(EXTRA_LDFLAGS)
```

12b ⟨*Files to Count* 12b⟩≡ (9g)

```
$(CFILES) \
```

12c ⟨*Clean temporary files* 9c⟩+≡ (5b) ◁9c 13b▷

```
rm -f *.[choa]
```

12d ⟨*Clean built files* 9b⟩+≡ (5b) ◁9f 22b▷

```
rm -f $(EXEC_FILES)
```

12e ⟨*Script Executables* 12e⟩≡ (11b)

```
\
```

12f ⟨*C Executables* 12f⟩≡ (11b)

```
\
```

12g ⟨*Build Source* 11e⟩+≡ (9e) ◁11e 12i▷

```
$(CFILES) $(HEADERS) \
```

Scripts and C programs may also be generated as part of the build process, for example to create machine-generated code.

12h ⟨*makefile.vars* 5d⟩+≡ ◁11b 14e▷

```
BUILD_SCRIPT_EXEC=⟨Build Script Executables 13c⟩

BUILD_C_EXEC=⟨C Build Executables 13d⟩
```

12i ⟨*Build Source* 11e⟩+≡ (9e) ◁12g 18b▷

```
$(BUILD_SCRIPT_EXEC) \
```

13a ⟨*makefile.rules* 5b⟩+≡ ◁12a 13h▷

```
$(BUILD_SCRIPT_EXEC): $(NOWEB)
        -notangle -R$@ $(NOWEB_ORDER) >$@
        chmod +x $@
```

13b ⟨*Clean temporary files* 9c⟩+≡ (5b) ◁12c 14b▷

```
rm -f $(BUILD_SCRIPT_EXEC) $(BUILD_C_EXEC)
```

13c ⟨*Build Script Executables* 13c⟩≡ (12h) 22c▷

```
\
```

13d ⟨*C Build Executables* 13d⟩≡ (12h)

```
\
```

The local libraries are built using a chunk naming convention. ⟨*Library* `name` *Members* (imported)⟩ chunks contain a plain listing of included object files. No support for shared libraries is provided at this time. Ideally, this should ensure that libraries dependent on others are listed earlier in the library order. Instead, the list is printed in reverse order, so the least dependent libraries are printed first.

Since `tac` may not be available everywhere, an alternative may be specified in `makefile.config`.

13e ⟨*Defaults for* `makefile.config` 6a⟩+≡ (4b) ◁6e 16a▷

```
TACCMD=tac
```

13f ⟨*makefile.config* 5c⟩+≡ ◁11f 16b▷

```
# Where to find the non-standard tac command (GNU coreutils)
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#TACCMD:=sed -n -e '1!G;h;$$p'
```

13g ⟨*Build variables for makefile includes* 6f⟩+≡ (4b) ◁7b

```
LOCAL_LIBS_BASE:=$(shell $(NOROOTS) $(NOWEB_ORDER) | \
  sed -n 's/<<Library \[\[\(.*\)]] Members>>/\1/p' | \
    $(TACCMD))
LOCAL_LIBS:=$(LOCAL_LIBS_BASE:%=-l%)
LOCAL_LIB_FILES:=$(LOCAL_LIBS_BASE:%=lib%.a)
```

13h ⟨*makefile.rules* 5b⟩+≡ ◁13a 13i▷

```
$(C_EXEC): $(LOCAL_LIB_FILES)
```

While it would be nice to not have to generate yet another support file for this, notangle is required for this to work. One of the goals of this system is to be able to generate a source tarball that does not depend on notangle.

13i ⟨*makefile.rules* 5b⟩+≡ ◁13h 14d▷

```
#define build-lib
#lib$(1).a: $(2)
#	rm -f $$@
#	ar cr $$@ $$^
#	ranlib $$@
#endef
#
#$(foreach l,$(LOCAL_LIBS_BASE),$(eval $(call build-lib,$l, \
#	$(shell notangle -R'Library [[$l]] Members' $(NOWEB_ORDER) 2>/dev/null))))
```

Instead, `makefile.libs` is generated with the macros expanded.

14a ⟨*makefile* 4b⟩+≡ (4a) ◁4b

```
MAKEFILES+=makefile.libs
makefile.libs: makefile.rules
	for x in $(LOCAL_LIBS_BASE); do \
	   printf %s lib$$x.a:; \
	   notangle -R"Library [[$$x]] Members" $(NOWEB_ORDER) | tr '\n' ' '; \
	   printf '\n\trm -f $$@\n'; \
	   printf '\tar cr $$@ $$^\n'; \
	   printf '\tranlib $$@\n'; \
	done > $@
-include makefile.libs
```

14b ⟨*Clean temporary files* 9c⟩+≡ (5b) ◁13b 15i▷

```
rm -f makefile.libs
```

Of course it might be useful to also provide installation rules for selected libraries, as well as their support files (header files and API documentation), but that is left for a future revision.

14c ⟨*Libraries* 14c⟩≡ (5d)

```
$(LOCAL_LIB_FILES) \
```

In addition to distributed binaries, test binaries may be built and executed. Other than checking return codes, no interpretation of test results is supported. It is intended primarily for manual testing. Sample binaries may be placed under these rules, as well.

14d ⟨*makefile.rules* 5b⟩+≡ ◁13i 15g▷

```
.PHONY: test test-bin
test-bin: $(TEST_EXEC) $(TEST_EXEC_SUPPORT)

$(TEST_EXEC):  $(LIB_FILES)

test: test-bin
	@set -e; for f in $(TEST_EXEC); do echo running $$f:; ./$$f; done
	⟨Additional Tests 15h⟩
```

14e ⟨*makefile.vars* 5d⟩+≡ ◁12h 18d▷

```
TEST_EXEC=⟨Test Executables 15a⟩

TEST_C_EXEC=⟨C Test Executables 15b⟩

TEST_SCRIPT_EXEC=⟨Test Scripts 15c⟩

TEST_EXEC_SUPPORT=⟨Test Support Executables 15d⟩

TEST_C_EXEC_SUPPORT=⟨C Test Support Executables 15e⟩

TEST_SCRIPT_EXEC_SUPPORT=⟨Test Support Scripts 15f⟩
```

15a ⟨*Test Executables* 15a⟩≡ (14e)

```
$(TEST_C_EXEC) $(TEST_SCRIPT_EXEC) \
```

15b ⟨*C Test Executables* 15b⟩≡ (14e)

```
\
```

15c ⟨*Test Scripts* 15c⟩≡ (14e)

```
\
```

15d ⟨*Test Support Executables* 15d⟩≡ (14e)

```
$(TEST_C_EXEC_SUPPORT) $(TEST_SCRIPT_EXEC_SUPPORT) \
```

15e ⟨*C Test Support Executables* 15e⟩≡ (14e)

```
\
```

15f ⟨*Test Support Scripts* 15f⟩≡ (14e)

```
\
```

15g ⟨*makefile.rules* 5b⟩+≡ ◁14d 18a▷

```
$(TEST_C_EXEC_SUPPORT) $(TEST_C_EXEC): $(LOCAL_LIB_FILES)
$(TEST_C_EXEC): $(TEST_C_EXEC_SUPPORT)

$(TEST_SCRIPT_EXEC) $(TEST_SCRIPT_EXEC_SUPPORT): $(NOWEB)
        -notangle -R$@ $(NOWEB_ORDER) >$@
        chmod +x $@
```

15h ⟨*Additional Tests* 15h⟩≡ (14d)

```

```

15i ⟨*Clean temporary files* 9c⟩+≡ (5b) ◁14b 18c▷

```
rm -f $(TEST_EXEC) $(TEST_C_EXEC:%=%.c) $(TEST_EXEC_SUPPORT) \
        $(TEST_C_EXEC_SUPPORT:%=%.c)
```

2.1 Support Files

Since `noroots` may not be on the target system, for example when using the tar file, a close equivalent is provided. This does not properly filter out non-root nodes, but most build operations require specific node names that are not likely to be used anywhere but the root level.

Makepp has trouble parsing this, so this is disabled for now.

16a ⟨*Defaults for* `makefile.config` 6a⟩+≡ (4b) ◁13e

```
NOROOTS=noroots
```

16b ⟨*makefile.config* 5c⟩+≡ ◁13f 20b▷

```
# The noroots command is used to extract file names from $(NOWEB)
# The following works well enough in most cases if noweb is missing
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#NOROOTS=sh -c "sed -n '/^<<.*>>=\$$/{s/=\$$//;p;}' \$$* \
#                                                  | sort -u" /dev/null
NOROOTS=noroots
```

In fact, when building just the code, it is likely that the only part of NoWeb required is `notangle`. For this, an equivalent can be provided in `perl`, which is installed on most modern UNIX systems by default. It is not a perfect emulation, but it should be close enough.

16c ⟨*perl-notangle* 16c⟩≡ 16d▷

```
#!/bin/sh
#!perl
# This code barely resembles noweb's notangle enough to work for simple builds
eval 'exec perl -x -f "$0" "$@"'
   if 0;
```

The only recognized command-line options are for tab size (`-t`) and chunk name (`-R`), which defaults to `*`. The remaining parameters are file names, and are left in `ARGV`.

16d ⟨*perl-notangle* 16c⟩+≡ ◁16c 16e▷

```
my $chunk = '*';    my $tab = 0;

while ($#ARGV > 0 && $ARGV[0] =~ /^-.*/) {
   if ($ARGV[0] =~ /^-R/) {
      $chunk = $ARGV[0];
      $chunk =~ s/-R//;
   } elsif ($ARGV[0] =~ /^-t(.*)/) {
      $tab = $1;
   }
   shift @ARGV;
}
```

Next, the files are read (<> just uses all files from `ARGV`) and raw chunks are extracted. They are stored in a hash keyed on chunk name.

16e ⟨*perl-notangle* 16c⟩+≡ ◁16d 17a▷

```
my ($inchunk, %chunks);
while (<>) {
  if (/^<<(.*)>>=$/) {
    $inchunk = $1;
  } elsif (/^@( |$)/) {
    $inchunk = "";
  } elsif ($inchunk ne "") {
    $chunks{$inchunk} .= $_;
  }
}
```

Then, the desired chunk is processed. Any chunk references are repeatedly extracted (using a regular expression) and replaced with their equivalent chunk. Multiline replacements get spaces prepended equivalent to the number of characters before the first line.

17a ⟨*perl-notangle* 16c⟩+≡ ◁16e 17b▷

```
my $chunkout = $chunks{$chunk};
while ($chunkout =~ /(?:^|\n)([^\n]*[^\n@])(?:<<((?:[^>\n]|>[^>\n])*)>>)/) {
  my $cv = $chunks{$2};
  my $ws = $1;
  $cv =~ s/\n$//;
  $ws =~ s/\S/ /g;
  $cv =~ s/\n/$&$ws/g;
  $chunkout =~ s/(^|[^@])(<<((?:[^>\n]|>[^>\n])*)>>)/$1$cv/;
}
```

Next, any final escapes are undone. NoWeb requires (and strips) escapes for « and ». Also, if the tab option was specified, initial groups of spaces equal to the tab size are replaced with tabs.

17b ⟨*perl-notangle* 16c⟩+≡ ◁17a 17c▷

```
$chunkout =~ s/@(<<|>>)/\1/g;
$chunkout =~ s/((^|\n)\t*) {$tab}/$1\t/g if ($tab gt 0);
```

Finally, the result is printed to standard output.

17c ⟨*perl-notangle* 16c⟩+≡ ◁17b

```
print $chunkout;
```

In order to avoid having to duplicate function prototypes in the main code and a header, the prototypes are generated automatically using cproto[5]. This uses the GNU make method of generating rules: `$(eval)`. Makepp does not support this.

The Makepp documentation claims that `$(eval)` can be replaced by `$[...]`, but my own experiments have shown me that this does not work: variables are not expanded correctly, and only one rule may be generated at a time (i.e., newlines are stripped out, even if they are explicitly added via perl code). In fact, using `$(...)` works fine for one rule, but is broken for multiple rules because, once again, newlines are stripped out. So, instead, it copies Makepp's code for adding a dependency directly. This is liable to break with different versions of Makepp, but it at least works with version 2.0.

Another Makepp issue triggered here was the original method of creating a temporary file, and then moving it to `cproto.h`. Makepp decided to add a rule dependency on the temporary file's name, which could not be removed or overridden. Now it just makes sure it got deleted on errors.

Of course there are times when blind prototype creation is undesirable. For example, libraries tend to place prototypes in headers. Some of them may use datatypes which are not defined in every C file. To allow

[5] http://invisible-island.net/cproto/

this, any prototypes detected in a header with the exact same syntax as automatically generated, up to the arguments' opening parenthesis, is removed from `cproto.h`. One which tends to vary from mainline to mainline is `main`, so it is always removed.

Also, local prototypes tend to rely on locally defined data types. To support this, the entire include of the static prototype file can be moved down to a location specified by a line containing only the comment `//static_proto`.

This uses the GNU `-i` extension, which may not be supported on all UNIX systems. If it doesn't, obtaining and using GNU sed is free and relatively painless. GNU sed also does not have a hidden line length limit or other misfeatures of other sed implementations, so I recommend using it all the time.

18a ⟨*makefile.rules* 5b⟩+≡ ◁15g 20e▷

```
$(COFILES): $(HEADERS) cproto.h

cproto.h: $(CFILES) $(HEADERS)
        echo >$@
        test -z "$(CFILES)" || ( \
           if cproto -E "$(CC) $(CFLAGS) $(EXTRA_CFLAGS) -E" \
                       $(CFILES) >$@.$$$$.h 2>/dev/null; then \
            sed -i -e '/^int main(/d' $@.$$$$.h; \
            grep -vn '^/\*' $@.$$$$.h | $(TACCMD) | \
              sed -e '⟨Strip down C prototype 19b⟩' | \
                while IFS=: read -r l p; do \
                  test -z "$$p" && continue; \
                  fgrep "$$p" $(HEADERS) | fgrep ":$$p" >/dev/null && \
                    sed -i -e "$${l}d" $@.$$$$.h; \
                done; \
            mv $@.$$$$.h $@; \
          else \
            rm -f $@ $@.$$$$.h; false; \
          fi )

ifeq ($(MAKEPP_VERSION),)
static_proto_rule=$1: $(1:%.o=%.c.static_proto)
$(foreach f,$(COFILES),$(eval $(call static_proto_rule,$f)))
else
sub f_adddep {
  my ($targ, $dep) = &args;
  my (undef, $mkfile, $mkline) = @_;
  my $tinfo = Mpp::File::file_info $targ, $mkfile->{CWD};
  Mpp::File::set_additional_dependencies($tinfo, $dep, $mkfile, $mkline);
  return '';
}
$(foreach f,$(COFILES),$(adddep $f,$(f:%.o=%.c.static_proto)))
endif
cproto.h: $(CFILES:%=%.static_proto)
%.c.static_proto: %.c
        ( \
          notangle -R$<.static_proto $(NOWEB_ORDER) 2>/dev/null || :; \
          cproto -S -E "$(CC) $(CFLAGS) $(EXTRA_CFLAGS) -E" $< 2>/dev/null \
        ) >$@ || (rm -f $@; false)
```

18b ⟨*Build Source* 11e⟩+≡ (9e) ◁12i 19a▷

```
cproto.h \
```

18c ⟨*Clean temporary files* 9c⟩+≡ (5b) ◁15i 22a▷

```
rm -f cproto.h.new *.c.static_proto{,.new}
```

18d ⟨*makefile.vars* 5d⟩+≡ ◁14e 20a▷

```
C_POSTPROCESS = | (tf="/tmp/.$$$$"; trap "rm $$tf" 0; cat >$$tf; \
                  if grep '^[ \t]*// static_proto$$' "$$tf" >/dev/null; then \
                    p='^[ \t]*// static_proto$$'; \
                  else \
                    p='^\#include "cproto.h"'; \
                  fi; \
                  sed -e "s%$$p%"'&\n\#include "$1.static_proto"%' $$tf)
```

19a ⟨*Build Source* 11e⟩+≡ (9e) ◁18b

```
$(CFILES:%=%.static_proto) \
```

Stripping down the C prototype requires proper handling of nested parentheses in the prameter list. This is done by repeatedly trying to strip the parameter list as if there were no nested parentheses, and if that fails, find the last pair of parentheses with no nested parentheses and remove them. The loop should eventually clear the parameter list of any nested parentheses, allowing the first substitution to succeed.

19b ⟨*Strip down C prototype* 19b⟩≡ (18a)

```
/(/{:a s/([^)]*);/(/;t;s/\(.*\)([^)]*)/\1/;ta;}
```

2.2 Support Chunks

In addition, C files can all take advantage of the fact that most headers are harmless to include, so it's easier to just include everything everywhere. The prototypes cannot be included in this list, because they really need to come last.

19c ⟨*Common C Includes* 19c⟩≡ (19e)

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <errno.h>
```

19d ⟨*Common C Warning* 19d⟩≡ (19e)

```
/*
  ⟨Common NoWeb Warning 4c⟩
*/
```

19e ⟨*Common C Header* 19e⟩≡

```
⟨Common C Warning 19d⟩
⟨Common C Includes 19c⟩
#include "cproto.h"
static const char version_id[] = ⟨Version Strings 19f⟩;
```

19f ⟨*Version Strings* 19f⟩≡ (19e)

```
"$Id: build.nw 161 2012-11-23 02:14:50Z darktjm $\n"
```

Some of those prototypes are hard to enable, but here are a few extra flags to help.

20a ⟨*makefile.vars* 5d⟩+≡ ◁18d 20c▷

```
EXTRA_CFLAGS += -D_LARGEFILE_SOURCE -D_XOPEN_SOURCE=600 \
                -D_XOPEN_SOURCE_EXTENDED=1
```

3 PDF Code Documentation

Building the PDF code documentation is simple: just convert to LaTeX using `noweave`, and then convert to PDF using `pdflatex` or `xelatex`. The conversion needs to be invoked repeatedly until the cross references (and other things, such as `longtable` measurements) settle down. Of course the figures need to be converted to the appropriate format, too. Rules are provided for EPS, xfig, and dia diagrams, since they can be stored in-line in the NoWeb source; additional rules can be provided for other image formats. Once again, the use of `$(eval)` requires special processing for Makepp.

In order to support attachments, the `xelatex` backend may need to be informed of the file location. This should include the current directory by default, but a common "security" measure is to remove the current directory from the default configuration file search path, which removes it from the attachment search path as a side effect. Rather than hard-code the backend and configuration path, a note is added to the config file so the user can set the environment variable appropriately (which also coincidentally overrides the security feature, as well).

20b ⟨*makefile.config* 5c⟩+≡ ◁16b 25b▷

```
# The PDF command to use: must be pdflatex or xelatex.
LATEXCMD=pdflatex
# note: to use attachments with xetex, the following may be necessary:
#LATEXCMD=env DVIPDFMXINPUTS=.:\$$TEXMF/dvipdfmx/ xelatex
```

20c ⟨*makefile.vars* 5d⟩+≡ ◁20a 21a▷

```
NOWEB_NINCL:=$(shell \
  for x in $(NOWEB); do \
    grep '^\\input{'"$$x"'}$$' $(NOWEB) >/dev/null && continue; \
    echo "$$x"; \
  done)
```

20d ⟨*Source Code Documentation Files* 20d⟩≡ (5d) 44b▷

```
$(patsubst %.nw,%.pdf,$(NOWEB_NINCL)) \
```

20e ⟨*makefile.rules* 5b⟩+≡ ◁18a 21b▷

```
# returns names of \input'd files
inc_f=$(shell sed -n -e 's/^\\input{\(.*\.nw\)}$$/\1/p' $1)
# returns self and any files \input'd, recursively
inc_all=$1 $(foreach f,$(call inc_f,$1),$(call inc_all,$f))
figs=$(shell sed -n -e 's/^\\includegraphics.*{\(.*\)}$$/\1/p' \
       $(call inc_all,$1))
ifeq ($(MAKEPP_VERSION),)
pdf_figrule=$(if $2,$(basename $1).pdf: $(patsubst %,%.pdf,$2))
$(foreach f,$(NOWEB_NINCL),$(eval $(call pdf_figrule,$f,$(call figs,$f))))
else
$((foreach f,$(NOWEB_NINCL),
   $(foreach d,$(call figs,$f),$(adddep $(basename $f).pdf,$d.pdf))))
endif
```

20e ⟨*makefile.rules* 5b⟩+≡ ◁18a 21b▷

```
# attempt to pull in multi-line messages, each line starts with same char
# but remove xparse/.*define-command messages, even though they
# are the only messages known to look like that
LMSCMD = /xparse\/.*define-command/b; \
         s/LaTeX $1/&/i;T; \
         p; \
         s/^\([^ ]\) LaTeX.*/\1/;T; \
         h; \
       :a N; \
         s/\(.\)\n\1/\1/;T; \
         p;g;ba;

%.pdf: %.tex
        while $(LATEXCMD) -interaction batchmode $< >/dev/null; do \
          egrep $(LATEX_REDO_MSG) $*.log >/dev/null || break; \
          test -f $*.idx && makeindex $(INDEX_FLAGS) -q $*; \
          fgrep '\citation' $*.aux || continue; \
          fgrep '\bibdata' $*.aux && bibtex $*; \
        done
        @sed -n -e '$(call LMSCMD,info)' $*.log
        @fgrep -i overfull $*.log || :
        @fgrep -i underfull $*.log | fgrep -v vbox || :
        @sed -n -e '$(call LMSCMD,warning)' $*.log
        @# maybe removing output file is too drastic, but manual run of
        @# pdflatex is not that hard
        @if sed -n -e '/^!.*Error/,/^l./p' $*.log | grep .; then \
          rm -f $@; \
          false; \
        else \
          :; \
        fi
```

21a ⟨*makefile.vars* 5d⟩+≡ ◁20c 50b▷

```
# picks up changes in references or long tables
LATEX_REDO_MSG:=" ha(s|ve) changed. Rerun[ .]"

FIGS:=$(shell sed -n -e 's/^\\includegraphics.*{\(.*\)}$$/\1/p' $(NOWEB))
FIGS_EPS:=$(patsubst %, %.eps, $(FIGS))
FIGS_PDF:=$(patsubst %, %.pdf, $(FIGS))
FIGS_DIA:=$(shell $(NOROOTS) $(NOWEB) | sed -n '/\.dia>>/{s/<<//;s/>>//;p;}')
FIGS_XFIG:=$(shell $(NOROOTS) $(NOWEB) | sed -n '/\.fig>>/{s/<<//;s/>>//;p;}')
FIGS_EPS_RAW:=$(shell $(NOROOTS) $(NOWEB) | sed -n '/\.eps>>/{s/<<//;s/>>//;p;}')
```

21b ⟨*makefile.rules* 5b⟩+≡ ◁20e 36a▷

```
$(FIGS_DIA): $(NOWEB)
        notangle -R$@ $(NOWEB_ORDER) | gzip >$@

%.eps: %.dia
        dia -e $@ $<

$(FIGS_EPS_RAW): $(NOWEB)
        notangle -R$@ $(NOWEB_ORDER) >$@

$(FIGS_XFIG): $(NOWEB)
        notangle -R$@ $(NOWEB_ORDER) >$@

%.eps: %.fig
        fig2dev -L eps $< $@

%.pdf: %.eps
```

21b ⟨*makefile.rules* 5b⟩+≡ ◁20e 36a▷

```
        epspdf $< 2>/dev/null || epstopdf $<
        # epstopdf gives no return code on errors, so:
        pdfinfo $@ >/dev/null
```

22a ⟨*Clean temporary files* 9c⟩+≡ (5b) ◁18c 26e▷

```
rm -f *.{log,aux,out,toc,lof,tex,dia,eps} $(FIGS_PDF)
```

22b ⟨*Clean built files* 9b⟩+≡ (5b) ◁12d 44d▷

```
rm -f *.pdf
```

3.1 Pretty Printing Wrapper

Or at least it should be simple, but some additional adjustments of the standard NoWeb process need to be made. For one thing, NoWeb does not pretty-print code chunks natively, so pretty-printing the code chunks requires a bit of work. A generic code chunk formatter filter for NoWeb is used for this. Like all noweb filters, it just sucks in the marked up form of the NoWeb source and spits out the same, with modifications. The language is tracked so that the formatter can behave differently depending on language.

22c ⟨*Build Script Executables* 13c⟩+≡ (12h) ◁13c 24c▷

```
nwweavefilt \
```

22d ⟨*Common Perl Prefix* 22d⟩≡ (22e)

```
#!/bin/sh
#!perl
⟨Common NoWeb Warning 4c⟩
eval 'exec perl -x -f "$0" "$@"'
   if 0;

use strict;
```

22e ⟨*nwweavefilt* 22e⟩≡

```
⟨Common Perl Prefix 22d⟩

⟨Initialize nwweavefilt 22f⟩

while (<STDIN>) {
   ⟨Filter lines from NoWeb token stream 34d⟩
   print $_; # print most lines
   ⟨Process lines from NoWeb token stream 22g⟩
}
```

Language is determined by scanning for language settings from the listings package, which may be commented out if the listings package isn't currently being used.

22f ⟨*Initialize* nwweavefilt 22f⟩≡ (22e) 23b▷

```
my $lang = 'txt';
```

22g ⟨*Process lines from NoWeb token stream* 22g⟩≡ (22e) 23a▷

```
# [^{] is to prevent self-match
if (/lstset{language=(?:.*\])?([^}]*)[^{]*/) {
    $lang = lc $1;
}
```

A code chunk begins with the chunk name, which can be ignored. After that, all lines are sucked in as part of the code, until the end of the chunk.

23a ⟨*Process lines from NoWeb token stream* 22g⟩+≡ (22e) ◁22g

```
if (/^\@begin code/) {
    # skip @defn
    $_ = <STDIN> or last;
    ⟨Filter lines from NoWeb token stream 34d⟩
    print $_;
    # skip @nl
    $_ = <STDIN> or last;
    ⟨Filter lines from NoWeb token stream 34d⟩
    print $_;
    # accumulate code in $code
    my $code = '';
    ⟨Accumulate code for highlighting 23c⟩
    ⟨Highlight accumulated code 24b⟩
}
```

Chunk references (`@use`) are replaced with a marker in the code that should work with most dumb code formatters. The marker is later replaced by the `@use` in the output stream. The marker can be any unique string that isn't likely to appear in code. No matter what is chosen, some bits of code will probably need to be modified to ensure the pattern never appears. For now, this uses tripled caret (^) characters. Other characters that might've worked are doubled or tripled at signs (@), tildes (~), or backticks (`). Index definitions are moved to the bottom of the chunk rather than being marked and replaced like `@use`; this removes the immediate context, but also eliminates highlighter confusion.

23b ⟨*Initialize* `nwweavefilt` 22f⟩+≡ (22e) ◁22f 24a▷

```
my $marker = '^'.'^^';
```

23c ⟨*Accumulate code for highlighting* 23c⟩≡ (23a)

```
# accumulate @use in @use
my @use;
# accumulate index defns in $end
my $end = '';
while (<STDIN>) {
    ⟨Filter lines from NoWeb token stream 34d⟩
    if (/^\@text/) {
        s/^\@text //;
        s/\n//;
        $code .= $_;
    } elsif ($_ eq "\@nl\n") {
        $code .= "\n";
    } elsif (/^\@index ((local|)defn|nl)/) {
        $end .= $_;
    } elsif (/^\@(use|index|xref)/) {
        $code .= $marker;
        push @use, $_;
    } elsif (/^\@end/) {
        push @use, $end . $_;
        last;
    } else {
```

23c ⟨*Accumulate code for highlighting* 23c⟩≡ (23a)

```
      print STDERR " ????? " . $_ . " ?????\n";
    }
}
```

The prettifier is passed in on the command line; it takes one (additional) argument - the name of the current language. It takes raw code (with markers for chunk references), and returns pretty code. Steps must be taken by the prettifier to ensure that the markers remain intact. The output of the prettifier is placed in the NoWeb stream as `@literal` lines[6]; markers are replaced by the appropriate `@use` line from `$use`.

The whole reason this was written in perl, rather than shell or awk, was to get speed (which awk already gave over the shell) and bidirectional pipes. This requires the `open2()` function. In addition, the command is passed as individual arguments, gaining a little more speed by not calling the shell to parse the command at every invocation.

24a ⟨*Initialize* `nwweavefilt` 22f⟩+≡ (22e) ◁23b 35a▷

```
use FileHandle;
use IPC::Open2;
# first arg is name of highlighting helper
my $filt = shift @ARGV;
```

24b ⟨*Highlight accumulated code* 24b⟩≡ (23a)

```
my ($pid, $reader, $writer);
$pid = open2($reader, $writer, $filt, $lang);

print $writer $code;
close $writer;

while (<$reader>) {
  while ((my $m = index($_, $marker)) >= 0) {
    if ($m > 0) {
      print '@literal ' . substr($_, 0, $m) . "\n";
    }
    print shift @use;
    $_ = substr($_, $m + length($marker));
  }
  if (length($_) > 1) { # 1 == just \n
    print '@literal ' . $_;
  }
  print "\@nl\n";
}
print shift @use; # the end
close $reader;
waitpid $pid, 0;
```

3.2 Pretty Printing with a highlighter and `framed`

One method of highlighting is to use an external highlighting program. Two related programs which work are `highlight`[7] and GNU `source-highlight`[8]. An additional wrapper script needs to be made to invoke the highlighter and massage the output a bit.

[6] Ramsey says I should write a new back end for this purpose, and has deprecated the `@literal` directive for this reason. Writing an entire back end just to reformat the code chunks a bit seems extreme, though. As of 2.11b, the directive still works.

[7] `http://www.andre-simon.de`

[8] `http://www.gnu.org/software/src-highlite/`

24c ⟨*Build Script Executables* 13c⟩+≡ (12h) ◁22c 32d▷

```
latexhl \
```

25a ⟨*latexhl* 25a⟩≡ 26a▷

```
#!/bin/sh
⟨Common NoWeb Warning 4c⟩
```

Since multiple highlighters are supported, the first thing to do is figure out which one to use. In previous versions of this system, only `highlight` version 2 was supported, so it is the preferred highlighter, if available. Since this needs to take advantage of a few low-level features, compatibility is not guaranteed. For `highlight` in particular, a check is made for major version 2 or 3 (early versions of 2 are not supported, but that is not checked). For either highlighter, it is difficult to come up with tests that ensure that this will actually work. If it doesn't, changes will need to be made here.

Rather than checking this for every chunk, the overall wrapper sets an environment variable to indicate which version to use. It also does any setup steps that do not need to be repeated for every chunk.

If the user wants, the check can be skipped entirely. Of course if the user setting does not match what the system actually has, there will be plenty of error messages while this runs.

25b ⟨*makefile.config* 5c⟩+≡ ◁20b 35d▷

```
# if non-blank, specify which highlighter to use:
#    highlight-2    named highlight, acts like highlight-2.x
#    highlight-3    named highlight, acts like highlight-3.x
#    source-highlight named source-highlight (probably 3.x)
# if blank, autodetect
HLPROG_TYPE=
```

25c ⟨*Prepare for weave* 25c⟩≡ (36d 50e) 29▷

```
⟨Check highlighter 25d⟩
```

25d ⟨*Check highlighter* 25d⟩≡ (25c 46b)

```
if [ -z "$HLPROG_TYPE" ]; then
  if type highlight >/dev/null 2>&1; then
    v=`highlight --version 2>&1 | fgrep highlight`
    v="${v##*version }"
    v="${v%%.*}"
    export HLPROG_TYPE=highlight-$v
    if [ $HLPROG_TYPE != highlight-2 -a $HLPROG_TYPE != highlight-3 ]; then
      # echo "Only version 2 or 3 of highlight supported." >&2
      # exit 1
      HLPROG_TYPE=
    fi
  elif type source-highlight >/dev/null 2>&1; then
    export HLPROG_TYPE=source-highlight
  else
    export HLPROG_TYPE=
  fi
else
  case "$HLPROG_TYPE" in
    highlight-[23]) type highlight >/dev/null 2>&1 || HLPROG_TYPE= ;;
    source-highlight) type source-highlight >/dev/null 2>&1 || HLPROG_TYPE= ;;
    *) HLPROG_TYPE= ;;
  esac
  if [ -z "$HLPROG_TYPE" ]; then
    echo "Invalid highlighter specified in HLPROG_TYPE" >&2
    exit 1
  fi
fi
```

Next, any preparations before running the source highlighter should be made, and the highlighter itself should be called.

26a ⟨*latexhl* 25a⟩+≡ ◁25a 26c▷

```
⟨Do per-chunk highlighting preparation for latexhl 26b⟩
case $HLPROG_TYPE in
  highlight-2)
    highlight ⟨highlight-2 options for latexhl 31e⟩ | \
      sed -n -e '⟨Adjust highlight-2 LaTeX 32a⟩' ;;
  highlight-3)
    highlight ⟨highlight-3 options for latexhl 31h⟩ | \
      sed -n -e '⟨Adjust highlight-3 LaTeX 32b⟩' ;;
  source-highlight)
    source-highlight ⟨source-highlight options for latexhl 31d⟩ | \
      sed -n -e '⟨Adjust source-highlight LaTeX 31i⟩' ;;
esac
```

To make the highlighter do its job, changing the block's background color needs to be supported as well. To do this, a box-type environment that sets background colors and can span pages is needed. The `framed` package comes close, but requires a bit of work to get the chunk to associate strongly with its header line. Without moving the code margin outside of the frame, it also extends the color box too far to the left. In the interest of simplicity, though, it will be used instead of writing an equivalent replacement. The background color itself is the named color `bgcolor`.

One problem to watch out for with `framed` is that if a code chunk starts at the bottom of a page, it is sometimes not broken at subsequent page boundaries. These chunks can be detected by scanning the LaTeX log for `Overfull \vbox` messages. This can be cured by inserting an explicit `\break` before the affected code chunk. All `\break` commands should be revisited regularly to make sure they are still necessary as the document changes. Of course this could also be taken as a sign that it is time to break the code chunk into smaller chunks, but sometimes breaking things up just for the sake of making them smaller just makes things messier.

26b ⟨*Do per-chunk highlighting preparation for* `latexhl` 26b⟩≡ (26a) 31a▷

```
case "$HLPROG_TYPE" in
  highlight-[23]) echo '\begin{framed}\advance\leftskip-\codemargin%' ;;
esac
```

26c ⟨*latexhl* 25a⟩+≡ ◁26a

```
case "$HLPROG_TYPE" in
  highlight-[23]) echo '\end{framed}%' ;;
esac
```

26d ⟨*Prepare for* `latexhl` 26d⟩≡ (36d) 27c▷

```
if [ $HLPROG_TYPE = source-highlight ]; then
printf %s\\n \
  'include "latexcolor.outlang"
   nodoctemplate
     "\begin{framed}\advance\leftskip-\codemargin%
"
     "
\end{framed}%
"
   end' >mylatex.outlang
fi
```

26e ⟨*Clean temporary files* 9c⟩+≡ (5b) ◁22a 27d▷

```
rm -f mylatex.outlang
```

27a ⟨*Framed Code Preamble* 27a⟩≡ (38c)

```
\usepackage{framed}
\setlength\FrameSep{0pt}
\setlength\FrameRule{0pt}
% nwdoit is the code header & flag indicating pass <= 2
% nwdoita is a flag indicating pass #1
\let\nwdoit\relax\let\nwdoita\relax
\def\FrameCommand#1{%
  \ifx\nwdoit\relax%
    % 3rd+ pass is subsequent pages - no code header
    \hskip\codemargin\colorbox{bgcolor}{#1}%
  \else%
    % 1st pass is invisible sizing pass, but only on older TeX.  How to tell?
    % 2nd pass is 1st page
    \vbox{\vbox{\nwdoit}\colorbox{bgcolor}{#1}}%
    \ifx\nwdoita\relax%
      \global\let\nwdoit\relax%
     % comment/uncomment next line if TeX newer/older
%   \else%
      \global\let\nwdoita\relax%
    \fi%
  \fi}
\renewenvironment{framed}%
  {\MakeFramed {\advance\hsize-2\fboxsep\advance\hsize-\codemargin%
   \FrameRestore}}%
  {\endMakeFramed}
```

27b ⟨*Insert code header into frame* 27b⟩≡ (36e)

```
/\\nwbegincode/s/\(sublabel{[^}]*}\)\(.*\)/\1\\def\\nwdoita{}\\def\\nwdoit{\2}%/
```

The actual colors come from a style file in `highlight`; there is actually no need to worry about which style to use when highlighting the chunks. This style file can be extracted by highlighting a dummy file, and parsing the output. If the background color is white, it is converted to a very light gray to ensure that code always looks a little different.

27c ⟨*Prepare for* `latexhl` 26d⟩+≡ (36d) ◁26d 27e▷

```
case "$HLPROG_TYPE" in
  highlight-[23])
    highlight ${theme:+-s $theme} -S c --out-format=latex -I </dev/null | \
      egrep 'newcommand|definecolor' | \
        sed '/bgcolor/s/{1,1,1}/{.98,.98,.98}/' > latexhl.sty.$$
    mv latexhl.sty.$$ latexhl.sty
esac
```

27d ⟨*Clean temporary files* 9c⟩+≡ (5b) ◁26e 28b▷

```
rm -f latexhl.sty{,.*}
```

On the other hand, `source-highlight` inserts the colors directly. However, since the colors don't change from chunk to chunk, it is better to extract them once, and use color names instead. This is done by creating an appropriate language definition and source file to get all possible entity types. In addition, `source-highlight` uses color names directly from HTML CSS files, which include invalid hexadecimal color specifiers. These need to be converted to valid decimal RGB triples.

27e ⟨*Prepare for* `latexhl` 26d⟩+≡ (36d) ◁27c

```
if [ $HLPROG_TYPE = source-highlight ]; then
  echo 'onestyle "\hl$style{$text}"' >> mylatex.outlang
  ⟨Get all source-highlight color names 28a⟩
  # outlang def for background color only, in same format as latexcolor
  echo 'nodoctemplate
        "\definecolor{bgcolor}\textcolor{$docbgcolor}"
        ""
        end' > mybg.outlang
  (
    source-highlight ${theme:+--style-css-file=$theme.css} -f latexcolor \
                     --lang-def=source-hl-elts.lang < source-hl-elts | \
      sed -n -e '/^\\mbox{}/{
                   s/\\mbox{}//;s/ \\\\//;s/\$\\_\$//g;
                   s/\(.*{\)\([^}]*\)\({*\)$/\\newcommand{\\hl\2}[1]{\1#1\3}/p;
                   s/^[^\\].*/\\newcommand{\\hl&}[1]{#1}/p;}'
    # fa is ff*.98
    source-highlight --lang-def=/dev/null --outlang-def=mybg.outlang \
                     ${theme:+--style-css-file=$theme.css} </dev/null | \
      sed 's/#ffffff/#fafafa/i;s/white/#f6f6f6/i'
  ) | ⟨Convert HTML color codes to LaTeX 28c⟩ | \
    sed -e '/^\\definecolor/{
              s/\\textcolor//
              # convert textcolor rgb to definecolor rgb
              s/\[rgb\]/{rgb}/;t
              # convert textcolor named to definecolor named
              s/{/{named}{/
            }' > latexhl.sty.$$
  mv latexhl.sty.$$ latexhl.sty
fi
```

28a ⟨*Get all* `source-highlight` *color names* 28a⟩≡ (27e 50e)

```
# need to sort in reverse order because source-highlight uses first match
# rather than longest match
for x in $(source-highlight --lang-list | cut -d\  -f3 | sort -u); do
  source-highlight --show-lang-elements=$x
done | sort -ur | fgrep -v 'named subexps' > source-hl-elts
# lang def simply makes a type its name
sed 's/.*/& = "&"/' < source-hl-elts > source-hl-elts.lang
```

28b ⟨*Clean temporary files* 9c⟩+≡ (5b) ◁27d

```
rm -f mybg.outlang source-hl-elts*
```

In order to convert HTML colors to LaTeX colors, the HTML color needs to be parsed and converted to decimal. Awk can do that, as long as it is POSIX awk. GNU awk does not work in non-POSIX mode because it does not support converting hexadecimal strings to numbers. To enable POSIX mode, either the non-portable `-posix` option can be given, or the slightly more portable `POSIXLY_CORRECT` environment variable can be set.

28c ⟨*Convert HTML color codes to LaTeX* 28c⟩≡ (27e)

```
POSIXLY_CORRECT=1 awk '⟨Convert HTML color codes to LaTeX using awk 28d⟩'
```

28d ⟨*Convert HTML color codes to LaTeX using* `awk` 28d⟩≡ (28c)

```
# change color styles from HTML color IDs to rgb color triples
# save RE in variable to avoid having to repeat
# note that this assumes 6-digit hex string, not 3-digit.
BEGIN { re1 = "\\\\textcolor\\{#[0-9a-fA-F]{6}\\}"
```

28d ⟨*Convert HTML color codes to LaTeX using* awk 28d⟩≡ (28c)

```
        re2 = "\\\\colorbox\\{#[0-9a-fA-F]{6}\\}" }
# print non-matching lines first
$0 !~ re1 "|" re2 { print }
# helper to convert to decimal triple
function todec3(start,   val, rv) {
    val = "0x" substr($0, RSTART + start, 2);
    rv = (val / 255) ",";
    val = "0x" substr($0, RSTART + start + 2, 2);
    rv = rv (val / 255) ",";
    val = "0x" substr($0, RSTART + start + 4, 2);
    rv = rv (val / 255);
    return rv
}
$0 ~ re1 "|" re2 {
  # loop to match all unconverted colors in this line
  while (match($0, re1)) {
    # capture groups would be real handy now
    # but instead, we extract each digit pair manually and append to
    # replacement string (ns)
    ns = "\\textcolor[rgb]{" todec3(12) "}";
    # convert string to be replaced to a regex by escaping \ { }
    nsre = "\\" substr($0, RSTART, RLENGTH);
    gsub("[{}]", "\\\\&", nsre);
    # globally replace all occurrences of this color
    gsub(nsre, ns);
  }
  while (match($0, re2)) {
    ns = "\\begingroup\\definecolor{boxcolor}{rgb}{" todec3(11) "}" \
         "\\colorbox{boxcolor}";
    # match the entire box so \endgroup can be placed after it
    match($0, re2 "\\{([^\\\\]|\\\\[^a-zA-Z]|\\\\[a-z]+\\{\\})*\\}");
    $0 = substr($0, 1, RSTART - 1) ns \
         substr($0, RSTART + 18, RLENGTH - 18) "\\endgroup" \
         substr($0, RSTART + RLENGTH);
  }
  print
}
```

A few syntax changes need to be made to the default language definitions. First, a language definition for highlighting sed comments is provided. As a special hack, `#include` is detected and not considered a comment (this is due to my use of HDF in a particular project, which uses this format). Second, the C language gets multi-line macros highlighted like regular source (already there in `source-highlight`), and support for 64-bit integer constants (already there in `highlight-3`). Finally, data types are being highlighted using the listings package's `moreemph` directive, so these directives are extracted and inserted into the keyword highlight list. This is actually done by using the name of the chunk containing all of these data types, so they are properly combined ahead of time.

29 ⟨*Prepare for weave* 25c⟩+≡ (36d 50e) ◁25c 33e▷

```
trap "rm -rf /tmp/latexhl.*.$$" 0
export HLDIR=/tmp/latexhl.hl.$$
mkdir -p $HLDIR
case "$HLPROG_TYPE" in
  source-highlight)
    # sed language only supports comments
    # #include supported for HDF
    echo "comment start '#(?!include\\W)'" > $HLDIR/sed.lang
    # c language needs a lot of help
    cat <<"EOF" > $HLDIR/myc.lang
      include "c.lang"
```

29 ⟨*Prepare for weave* 25c⟩+≡ (36d 50e) ◁25c 33e▷

```
      subst number =
'\<[+-]?((0x[[:xdigit:]]+)|(([[:digit:]]*\.)?
[[:digit:]]+([eE][+-]?[[:digit:]]+)?))[uUlL]*\>'
type =
EOF
    printf %s '⟨Known Data Types 33d⟩' | egrep -v '^[[:space:]]*(%|$)' | \
      sed -e 's/[[:space:]]//g;s/,/|/g;s/%.*//;s/^/"/;s/|$/"/;$!s/$/,/' \
          >>$HLDIR/myc.lang
    ;;
  highlight-2)
    mkdir -p $HLDIR/langDefs
    # sed language only supports comments
    # #include supported for HDF
    echo '$SL_COMMENT=regex(^\s*(#(?!include\W)))' > $HLDIR/langDefs/sed.lang
    # c language needs a lot of help
    hl_kw="`printf %s '⟨Known Data Types 33d⟩' | egrep -v '^[[:space:]]*(%|$)' | \
           sed -e 's/[[:space:]]//g;s/,/ /g;s/%.*//;\$!s/\$/\\\\/' `"
    langdir=`highlight --print-config | sed -n '/Language def/s/.*: *//p' `
    sed < ${langdir:-/usr/share/highlight/langDefs}/c.lang -e '
      # remove cont to fully highlight multi-line macros
      s/.*CONTINUATIONSYMBOL.*//
      # fix preproc expression
      /DIRECTIVE/s/#/regex(^\\s*(#))/
      # add some kws to kwb using old listings package moreemph decls
      /KW_LIST(kwb)/a\'"${hl_kw}"'
      # same thing, but newer versions of highlight
      /KEYWORDS(kwb)/a\'"${hl_kw}"'
      # change number RE to include UL, LL, ULL, etc.
      $a\$DIGIT=regex((?:0x|0X)[0-9a-fA-F]+|\\d*[.]?\\d+(?:[eE][\\-+]\\d+)?[lLuU]*)
      ' > $HLDIR/langDefs/c.lang
    ;;
  highlight-3)
    # sed language only supports comments
    # #include supported for HDF
    echo 'Description="sed"
          Keywords={}
          Comments={
            {Block=false, Delimiter={[[^\s*(#(?!include\W))]]}}
          }' > $HLDIR/sed.lang
    # c language needs a lot of help
    langdir=`highlight --print-config | sed -n '/Language def/s/.*: *//p' `
    echo "dofile('${langdir}c.lang')" > $HLDIR/myc.lang
    cat <<"EOF" >> $HLDIR/myc.lang
      -- fix preproc expression and remove cont to highlight multi-line macros
      Preprocessor = { Prefix=[[^\\s*(#)]] }
      --add some kws to ID=2 using old listings package moreemph decls
      for i, v in ipairs({
EOF
    printf %s '⟨Known Data Types 33d⟩' | egrep -v '^[[:space:]]*(%|$)' | \
      sed -e 's/[[:space:]]//g;s/,/","/g;s/%.*//;s/$/"/;s/^/"/;s/""//;' \
         >> $HLDIR/myc.lang
    cat <<"EOF" >>$HLDIR/myc.lang
                          }) do
        table.insert(Keywords[2].List, v)
      end
EOF
    ;;
esac
```

To actually invoke the highlighter, the language specified on the command line must be passed in. If it is one of the overrides, the override should be used instead. For `highlight-2`, this is done using the `-E` option. For the other two, it is done by specifying the language definition file directly. Since these require checking the language argument anyway, they can also change the language to a different one if necessary. I

have not tested all languages available to the listings package, which is used to determine legal language names, but any which mismatch what is supported by the external highlighter should be listed here.

31a ⟨*Do per-chunk highlighting preparation for* latexhl 26b⟩+≡ (26a) ◁26b

```
⟨Do generic per-chunk highlighting preparation 31b⟩
```

31b ⟨*Do generic per-chunk highlighting preparation* 31b⟩≡ (31a 49f) 31f▷

```
if [ source-highlight = $HLPROG_TYPE ]; then
  case $1 in
    ⟨Translate listings source type to source-highlight source type 31c⟩
    *) larg="-s $1"
  esac
fi
```

31c ⟨*Translate listings source type to source-highlight source type* 31c⟩≡ (31b)

```
[cC]) larg="--lang-def=$HLDIR/myc.lang" ;;
sed) larg="--lang-def=$HLDIR/sed.lang" ;;
make) larg="-s makefile" ;;
```

31d ⟨source-highlight *options for* latexhl 31d⟩≡ (26a)

```
--outlang-def=mylatex.outlang --failsafe $larg
```

31e ⟨highlight-*2 options for* latexhl 31e⟩≡ (26a)

```
--out-format=latex -f -S $1 -E $HLDIR
```

31f ⟨*Do generic per-chunk highlighting preparation* 31b⟩+≡ (31a 49f) ◁31b

```
if [ highlight-3 = $HLPROG_TYPE ]; then
  case $1 in
    ⟨Translate listings source type to highlight-3 source type 31g⟩
    *) larg="-S $1"
  esac
fi
```

31g ⟨*Translate listings source type to highlight-3 source type* 31g⟩≡ (31f)

```
[cC]) larg="--config-file=$HLDIR/myc.lang" ;;
sed) larg="--config-file=$HLDIR/sed.lang" ;;
```

31h ⟨highlight-*3 options for* latexhl 31h⟩≡ (26a)

```
--out-format=latex -f $larg
```

Finally, the output needs to be adjusted. All three need a strut on the first line to ensure consistent appearance. All three generate a final blank line, which is just excess whitespace. All three encode the triple-hat sequence which is used to indicate chunk references as LaTeX, which must be decoded. All three also add unnecessary garbage at the end of each line. The highlight program also terminages LaTeX commands with a space rather than curly braces, which has caused me problems in some places. The last line should also be terminated by a percent sign to remove excess whitespace.

31i ⟨*Adjust* `source-highlight` LaTeX 31i⟩≡ (26a)

```
# remove blank line at end
/^\\mbox{}$/d
# detexify triple-^
s/\(\\textasciicircum{}\)\1\1/^"^^/g
# remove unnecessary EOL stuff
s/ \\\\$//
# fix _ in command names
s/\(\\hl[a-z]*\)_/\1/g
# add strut to first line
2s/$/\\strut{}/
# add a % to 2nd-to-last line
x;${s/$/%/;G};1!p
```

32a ⟨*Adjust* `highlight-2` LaTeX 32a⟩≡ (26a 32b)

```
# removes 1st 2 lines (setting font & spacing - already done)
1,2d
# remove last few lines (just extra whitespace & restoring font/spacing)
/^\\mbox{}$/,$d
# since multiple lines were deleted, just start sed over so $ works right
p' | sed -e '
# detexify triple-^
s/\(\\textasciicircum \)\1\1/^"^^/g
# remove unnecessary fill on every line
s/\\hspace\*{\\fill}\\\\$//
# use {} instead of single space to terminate TeX cmds
s/\(\\[a-z][a-z]*\) /\1{}/g
# add strut to first line
1s/$/\\strut{}/
# remove last CR by tacking on % to last line
$s/$/%/
```

32b ⟨*Adjust* `highlight-3` LaTeX 32b⟩≡ (26a)

```
⟨Adjust highlight-2 LaTeX 32a⟩
```

3.3 Pretty Printing with `listings`

Or, since the commands from the listings package are used anyway, the listings package could be used instead. To do this, `nwbegincode` and `nwendcode` need to be changed to the listings package equivalents. Then, all commands within the code are escaped using ctrl-G, a character not likely in text and easily specified within sed. The escape character must be activated using `\lstset{escapechar=^^G}` in the preamble.

32c ⟨*Change* `nwcode` *to* `listings` 32c⟩≡ (36e)

```
s/^\(.*\)\(\\nwbegincode{[0-9]*}\)\(.*\)$/\1\\begin{lstlisting}\n\a\3\a/
/\\def\\nwend/!s/^\(.*\)\(\\nwendcode\)\(.*\)$/\\end{lstlisting}\1\3/
```

32d ⟨*Build Script Executables* 13c⟩+≡ (12h) ◁24c 36b▷

```
addlistings \
```

32e ⟨*addlistings* 32e⟩≡

```
#!/bin/sh
⟨Common NoWeb Warning 4c⟩

# use the ctrl-g escape (\a = ann = bell)
sed -e 's/\^\^/\a&\a/g
        # insert dummy escapes into potential listing ender tokens
        s/\\end{lstlisting}/\\\a\aend{lstlisting}/g
        s/\\nwbegincode/\\\a\anwbegincode/g
        s/\\nwendcode/\\\a\anwendcode/g'
```

33a ⟨*Preamble Adjustments for* `listings` 33a⟩≡ (38c) 33b▷

```
% Use ^G for escape char - not likely in document
\catcode`\^^G=13
\lstset{escapechar=^^G}
```

The appropriate language definitions need to be loaded at the end of the LaTeX preamble. This includes comment highlighting for `sed` and selecting the appropriate default sublanguages.

33b ⟨*Preamble Adjustments for* `listings` 33a⟩+≡ (38c) ◁33a 33c▷

```
% Stuff for the listings package
% note: lstloadlanguages seems broken in some versions
%\lstloadlanguages{C,sh,[gnu]make,[LaTeX]TeX} %,sed
% Most of sed can't be expressed in the listings package - no regexes
\lstdefinelanguage{sed}{morecomment=[f]\#}
\lstdefinelanguage{txt}{}
\lstset{defaultdialect=[gnu]make,defaultdialect=[LaTeX]TeX}
%The supplied HTML def has no dialect, so setting this creates infinite loop
%\lstset{defaultdialect=[ext]HTML}
\lstset{language=C}
```

A few additional adjustments need to be made for those languages. In particular, some keywords are missing, and some standard data types are missing. Also, the data types provided by the used libraries need to be highlighted. The data type adjustments are applied when using `highlight` as well, since the `moreemph` directives are parsed out.

33c ⟨*Preamble Adjustments for* `listings` 33a⟩+≡ (38c) ◁33b 34b▷

```
% HTML doesn't bold enctype
\lstdefinelanguage[ext]{HTML}[]{HTML}{morekeywords={enctype,public,xml}}
% Known data types
\lstset{moreemph={⟨Known Data Types 33d⟩
}}
```

33d ⟨*Known Data Types* 33d⟩≡ (29 33c)

```
% C
regex_t,regmatch_t,FILE,va_list,pollfd,pthread_t,%
pthread_mutex_t,pthread_cond_t,uid_t,gid_t,mode_t,pid_t,time_t,socklen_t,%
sockaddr_in,sockaddr,off_t,utimbuf,%
```

Since there are not really any styles for this package, the highlighter program's styles can be used instead. If they are present, they need to be converted to the appropriate directives. If there is no highlighter program available, a dummy `latexhl.sty` needs to be created as well. The format of a listings style command is a declaration rather than a command, so the style file needs to be converted as well.

33e ⟨*Prepare for weave* 25c⟩+≡ (36d 50e) ◁29 36e▷

```
if [ -z "$HLPROG_TYPE" ]; then
  echo >latexhl.sty
fi
```

34a ⟨*Change* `highlight` *style to* `listings` *style* 34a⟩≡ (36e)

```
s/\[1\]//;s/#1//g
s/^[^{]*{[^}]*}$/&{}/
# colorbox is not supported, so silently delete
s/\\colorbox{[^}]*}{\(.*\)}/\1/
s/\\text\(color[^}]*}\){\(.*\)}/\\\1\2/
s/\\bf{\(.*\)}/\\bfseries\1/
s/\\textbf{\(.*\)}/\\bfseries\1/
s/\\it{\(.*\)}/\\itshape\1/
s/\\textit{\(.*\)}/\\itshape\1/
# underline is not supported, so silently turn into sans-serif
s/\\underline{\(.*\)}/\\sffamily\1/
s/\\texttt{\(.*\)}/\\ttfamily\1/
```

34b ⟨*Preamble Adjustments for* `listings` 33a⟩+≡ (38c) ◁33c 34c▷

```
\lstset{backgroundcolor=\color{bgcolor}}
\ifx\hlstd\undefined\else
\lstset{basicstyle=\hlstd\footnotesize,keywordstyle=\hlkwa,emphstyle=\hlkwb,%
        commentstyle=\hlcom,stringstyle=\hlstr,directivestyle=\hldir}
\fi
\ifx\hlnormal\undefined\else
\lstset{basicstyle=\hlnormal\footnotesize,keywordstyle=\hlkeyword,%
        emphstyle=\hltype,commentstyle=\hlcomment,stringstyle=\hlstring,%
        directivestyle=\hlpreproc}
\fi
```

Finally, some general appearance adjustments are made. The entire listing is shifted over a bit, and the font layout is set for minimum mangling.

34c ⟨*Preamble Adjustments for* `listings` 33a⟩+≡ (38c) ◁34b

```
\lstset{columns=[l]fixed,framexleftmargin=1.5em,framexrightmargin=1em}
```

3.4 Additional Output Adjustments

Now that code chunks can be pretty-printed, a few more adjustments need to be made. Underscores in chunk names (such as when the chunk name is a file name) need to be escaped. The easiest place to do this is in a filter, which has the chunk names separated out already. For maximum flexibility, this is done in a perl script.

34d ⟨*Filter lines from NoWeb token stream* 34d⟩≡ (22 23) 35b▷

```
if (/^\@use / or /^\@defn /) {
  s/_/\\_/g unless /\[\[/;
}
```

Chunk names referring to chunks in other files should be indicated as such. For now, this is done by prepending the chunk name with the source file in parentheses. The parentheses are to help force the index entry to the top of the list. The printed source file is the top-most file which includes this definition; in other words, if a file is included by another, its definitions are owned by the includer. This can be disabled for user-level documentation by adding the commment:

```
%%% no-ext-ref
```

35a ⟨*Initialize* `nwweavefilt` 22f⟩+≡ (22e) ◁24a

```
my %ch;
# final args are $noweb_order
my %feqv;
my $disext;
# find included files
for my $a (@ARGV) {
  open(A, $a);
  while(<A>) {
    $disext = /^%%% no-ext-ref$/;
    last if $disext;
    /^\\input{(.*\.nw)}$/ or next;
    $feqv{$1} = $a;
  }
}
unless($disext) {
  my $myf = pop @ARGV; # my name is always last; skip it
  # now loop over files, making included files look like includer
  for my $a (@ARGV) {
    my $f = $a;
    while($feqv{$f}) { $f = $feqv{$f}; }
    next if($f eq $myf);
    open(A, $a);
    while(<A>) {
      if(/^<<(.*)>>=$/) {
        $ch{$1} = $f unless $ch{$1};
      }
    }
  }
}
```

35b ⟨*Filter lines from NoWeb token stream* 34d⟩+≡ (22 23) ◁34d

```
if(!$disext and (/^(\@use )(.*)/ or /^(\@defn )(.*)/) and $ch{$2}) {
  $_ = "$1($ch{$2}) $2\n";
}
```

There is little point in separating consecutive code chunks with a lot of white space, so the excess whitespace added by NoWeb needs to be removed.

35c ⟨*Reduce interchunk whitespace* 35c⟩≡ (36d)

```
/\\nwendcode{}\\nwbegindocs/{
  N;N
  /\\lstset/{s/$/%/;N;}
  /\\nwenddocs/{s/\\nwbegindocs.*docspar\n\n//;s/\\nwenddocs{}//;}
}
```

Then, the LaTeX preamble needs to be added. Except for the overall document style, this is mostly the same for all documents. This means that the first line, the document style, must come before the common preamble. Rather than looking for that particular line, a special comment is replaced:

```
%%% latex preamble
```

The following script does this, and brings all of the above together. In order to merge sources correctly, the dependency order (tree) needs to be passed in as well as the input and output file names. The highlight theme can be selected using a command-line parameter. Selection of the listings package versus an external highlighter is done by prefixing the highlight theme name with `list:`.

35d ⟨*makefile.config* 5c⟩+≡ ◁25b 57f▷

```
# Highlight theme for PDF/HTML formatted source code
# Blank uses default highlight style; prefix w/ list: to use listings package
HL_THEME:=
```

36a ⟨*makefile.rules* 5b⟩+≡ ◁21b 36c▷

```
%.tex: %.nw nw2latex $(call tree,%.nw)
        ./nw2latex $< $@ "$(call tree,$<)" "$(HL_THEME)" "$(HLPROG_TYPE)"
```

36b ⟨*Build Script Executables* 13c⟩+≡ (12h) ◁32d 46a▷

```
nw2latex \
```

36c ⟨*makefile.rules* 5b⟩+≡ ◁36a 43a▷

```
nw2latex: latexhl nwweavefilt addlistings
```

36d ⟨*nw2latex* 36d⟩≡

```
#!/bin/sh
⟨Common NoWeb Warning 4c⟩

noweb="$1"
outf="$2"
noweb_order="$3"
theme="$4"
export HLPROG_TYPE="$5"

# prefix theme with "list:" to enable lstlistings
uselst="${theme%%:*}"
if [ "list" = "$uselst" ]; then
  theme="${theme#list:}"
else
  uselst=
fi

⟨Prepare for weave 25c⟩
⟨Prepare for latexhl 26d⟩
(
  ln=`grep -n '^%%% latex preamble' "$noweb" | cut -d: -f1 | head -n 1`
  head -n $ln "$noweb"
  cat <<"EOF"
⟨preamble.tex 37⟩
EOF
  tail -n +$((ln+1)) "$noweb"
) | ⟨Pre-process before weave 39b⟩
  noweave -filter "./nwweavefilt $filt $noweb_order" \
          -delay -index - | ⟨Post-process latex after weave 41a⟩
            sed -e '⟨Reduce interchunk whitespace 35c⟩' -e "$postproc" >"$outf"
```

36e ⟨*Prepare for weave* 25c⟩+≡ (36d 50e) ◁33e 39c▷

```
if [ -z "$uselst" -a -n "$HLPROG_TYPE" ]; then
  filt=./latexhl
  postproc='⟨Insert code header into frame 27b⟩'
else
  filt=./addlistings
  postproc='⟨Change nwcode to listings 32c⟩'
  sed -e '⟨Change highlight style to listings style 34a⟩' \
     < latexhl.sty > latexhl.sty.$$
  mv latexhl.sty.$$ latexhl.sty
fi
```

The LaTeX preamble is fairly straightforward. The `fontenc`, `inputenc`, `fontspec` and `babel` packages are pretty much required anywhere. The input encoding is forced to latin1 for `pdflatex`; it is expected that `xelatex` be used for Unicode. The `noweb` package is obviously needed for the NoWeb output. The `rcs` package is included to easily incorporate those nasty $-heavy RCS/CVS/SVN keywords. The `ifpdf` and `ifxetex` packages are needed for using this LaTeX source with non-PDF output and non-Unicode input. The standard PDF font packages are used to reduce the PDF size and hopefully look smoother on a greater number of operating systems. The `graphicx` package is the easiest way to include figures; the PDF output styles should use scalable PDF for figure sources (most other types can be converted to this). The `listings`, `color`, and `xcolor` packages and the `latexhl` highlight theme file are required for color syntax highlighting. The `headings` package gives more informative headers and footers by default. The `hyperref` package gives hyperlinks in the PDF file. Links are made less distracting by replacing the red outline box with dark blue-colored text, and URL links are colored the same way. The `multitoc` package provides multi-column tables of contents, which once again is to reduce the number of pages a bit.

37 ⟨*preamble.tex* 37⟩≡ (36d 52a) 38a▷

```
\usepackage[T1]{fontenc}
\usepackage{ifxetex}
\ifxetex
% Use Latin Modern; selecting standard PDF fonts does not work
% Latin Modern doesn't look that great, though (it's apparently not an
% exact duplicate of Computer Modern), but it's easy to set fonts after
% the standard preamble.
\usepackage{fontspec}
% \documentclass[twoside,english]{..} doesn't seem to work right with
% xetex.  Using polyglossia seems to help, with a language setting
% after the standard preamble instead (using \setdefaultlanguage)
\usepackage{polyglossia}
\else
\usepackage[latin1]{inputenc}
\usepackage{babel}
\fi
\usepackage{rcs}
\usepackage{ifpdf}
\ifpdf
% Use standard fonts for PDF output
\usepackage{courier}
\usepackage[scaled]{helvet}
\usepackage{mathptmx}
\fi
\usepackage{noweb}
\usepackage{graphicx}
\ifpdf
% Use pdf figs by default (hack)
\makeatletter
\def\Gin@extensions{.pdf}
\makeatother
\fi
\ifxetex
% Use pdf figs by default (hack)
\makeatletter
\def\Gin@extensions{.pdf}
\makeatother
\fi
\usepackage{listings}
\usepackage{color}
\usepackage[usenames,dvipsnames,svgnames,x11names,table]{xcolor}
\usepackage{latexhl}
%
\pagestyle{headings}
% new since 2009: rerunfilecheck (automatic include by hyperref)
\usepackage[aux]{rerunfilecheck}
```

37 ⟨*preamble.tex* 37⟩≡ (36d 52a) 38a▷

```
%
% Hyperlink it!
\definecolor{darkblue}{rgb}{0,0,0.3}
\usepackage[colorlinks=true,linkcolor=darkblue,urlcolor=darkblue]{hyperref}
% And set the title/author properly (taken from hyperref slides.pdf)
\makeatletter
\newcommand{\org@maketitle}{}% ensure not defined
\let\org@maketitle\maketitle
\def\maketitle{\hypersetup{pdftitle={\@title}, pdfauthor={\@author}}%
\org@maketitle}
%
% Multi-column table of contents
\usepackage[toc,lof]{multitoc}
```

Then, some adjustments need to be made for NoWeb. NoWeb's `\setupcode` does some adjustments to LaTeX layout that override the code highlighting, so that is disabled. NoWeb tries too hard to keep code chunks on one page, so some adjustments are made to those parameters. Finally, the NoWeb cross reference listings are adjusted a bit and the code size is made smaller. Also, the "never defined" message should probably be changed for imported symbols, and in fact all imported symbols should be shown identically. It is not currently possible to remove the "never defined" message completely, so it is changed to "(imported)" even if it is not imported.

38a ⟨*preamble.tex* 37⟩+≡ (36d 52a) ◁37 38b▷

```
% Code formatting is done by highlight/lstlisting, not noweb
\def\setupcode{\Tt}

% This is from the noweb FAQ
% Allow code chunks to span pages
\def\nwendcode{\endtrivlist \endgroup}
%\def\nwendcode{\endtrivlist \endgroup \vfil\penalty10\vfilneg}
%
% Allow docs to be split from code chunks
\let\nwdocspar=\smallbreak
%\let\nwdocspar=\par
%
% try a little harder to keep code chunks on one page
\nwcodepenalty=1000
%
% Code can be huge, so make it smaller (smallcode, scriptsizecode, footnotesizecode)
% Make chunk index useful (longchunks)
% Remove xrefs from chunks themselves (noidentxref)
\noweboptions{footnotesizecode,longchunks,noidentxref}

% change "never defined" to "imported"
\makeatletter
\def\@nwlangdepnvd{imported}
\makeatother
```

One other thing I usually do for code is to allow line breaks after underscores.

38b ⟨*preamble.tex* 37⟩+≡ (36d 52a) ◁38a 38c▷

```
% break lines after _
\let\oldunderscore\_
\def\_{\oldunderscore\discretionary{}{}{}}
```

Finally, the syntax highlighter adjustments need to be made.

38c ⟨*preamble.tex* 37⟩+≡ (36d 52a) ◁38b 39a▷

```
⟨Preamble Adjustments for listings 33a⟩
% for highlight
\ifpdf
⟨Framed Code Preamble 27a⟩
\fi
\ifxetex
⟨Framed Code Preamble 27a⟩
\fi
```

3.5 Attaching Files

In addition to the adjustments for appearance, the source files need to be attached. Attachments are done using the LaTeX `navigator` package. The `embedfile` package used by previous versions of this build sytem does not support `xelatex`. The old command is retained, though; any files in addition to the source files must be attached using the `\embedfile` command in order for other portions of the build system to pick up the files properly.

39a ⟨*preamble.tex* 37⟩+≡ (36d 52a) ◁38c 45b▷

```
\ifpdf
\usepackage{embedfile}
\fi
\ifxetex
\usepackage{navigator}
\def\embedfile#1{\embeddedfile{#1}{#1}}
\fi
```

39b ⟨*Pre-process before weave* 39b⟩≡ (36d 52a) 40a▷

```
(att_nwo="`echo $noweb_order | \
                 sed -e 's/\\([-_a-zA-Z.]*\\) */\\\\\\\\embedfile{\\1}%\\\\n/g'`"
 sed -e 's/^\\begin{document}/'"$att_nwo&/") | \
```

3.6 User Documentation

One challenge with literate programming is to provide separate user-level documentation that keeps track of the source code as well as the design documentation. For this, a few code reinsertion tricks are used. First, excerpts from the documentation can be reinserted at a chosen location. These excerpts are surrounded with begin and end comments, which must be at the beginning of the line:

```
% Begin-doc chunk-name
...
% End-doc chunk-name
```

The chunk name should consist only of alphanumeric characters, minus signs, and underscores. Insertion is done using a special input command:

```
\input{chunk-name.tex} %%% doc
```

This is done before weaving so that NoWeb formatting can be done on this, as well. It requires two passes in case the chunks are out of order or in different files. The first pass extracts the chunks, and the second pass replaces any remaining unresolved internal references.

39c ⟨*Prepare for weave* 25c⟩+≡ (36d 50e) ◁36e 41d▷

```
eval sed -n -e \''s/^\\input{\(.*\)\.tex} %%% doc$/\1/p'\' $noweb_order | \
  sort -u | while read c; do
    printf %s\\n "/^\\\\input{${c}\\.tex} %%% doc\$/{s/.*/%/;a\\%\\"
    eval sed -n "'/^% Begin-doc $c\$/,/^% End-doc $c\$/{
                  /^% Begin-doc $c\$/!{/^% End-doc $c\$/!"'{
                    s/\\/\\\\/g;s/$/\\/;p;};};}'\' $noweb_order | sed '$s/.$//'
    echo '}'
  done > /tmp/latexhl.doc.$$
while grep '^\\\\input{.*\.tex} %%% doc\\$' /tmp/latexhl.doc.$$ >/dev/null; do
  sed -e '
     s/\^/^\\\\/;s/doc\$/doc\\\\\$/;s^/%/^/%\\\\/^;h;s/$/\\\\/
     :a x;s/\\\\//g;/\\$/!{x;s/$/\\/;n;b};x;n;h;s/\\/\\\\/g;s/$/\\/;ba' \
              < /tmp/latexhl.doc.$$ >/tmp/latexhl.doc2.$$
  sed -f /tmp/latexhl.doc2.$$ /tmp/latexhl.doc.$$ >/tmp/latexhl.doc3.$$
  mv /tmp/latexhl.doc3.$$ /tmp/latexhl.doc.$$
done
```

40a ⟨*Pre-process before weave* 39b⟩+≡ (36d 52a) ◁39b 40b▷

```
sed -f /tmp/latexhl.doc.$$ | \
```

One problem with reinsertion is that labels are duplicated. While leaving the labels out manually works, it is cumbersome and may not be what is wanted. For example, one way to split the user documentation into a separate file is to reinsert it into an otherwise empty NoWeb source file. Stripping the labels out manually means that the labels will be undefined in the fresh document. A better way is to have them automatically filtered out.

One problem with filtering them out is deciding which one to filter out. Ideally, both would be kept, and the reinserted one would be given a new, unique name, and all references to that label within reinserted text would be updated as well. This is a task for a future revision, though. In fact, even deciding that the main text or the reinserted text should be preferred is too complex of a question for now. Instead, the first label after all reinsertions are complete is kept, and all others are removed. As an additional level of stupidity, label commands may not span lines or contain curly braces. They may also not appear after a percent-sign character on the same line, even if that percent sign is escaped for LaTeX.

40b ⟨*Pre-process before weave* 39b⟩+≡ (36d 52a) ◁40a 41b▷

```
awk '⟨Remove duplicate labels from LaTeX source 40c⟩
     { print }' | \
```

40c ⟨*Remove duplicate labels from LaTeX source* 40c⟩≡ (40b)

```
/(^|[^\\])\\label{/ {
  s = $0
  os = ""
  while (match(s, "\\\\label\\{[^}]*\\}")) {
    os = os substr(s, 1, RSTART - 1)
    l = substr(s, RSTART, RLENGTH)
    s = substr(s, RSTART + RLENGTH)
    if(l in gotit)
      continue;
    os = os l
    gotit[l] = 1
  }
  print(os s)
  next
}
```

Another problem is that there is more than one way to create a label. For example, the sectioning commands may be redefined to automatically create a label based on the section name. This document defines

no such things[9], but as implemented above, more awk code can be added to ⟨*Remove duplicate labels from LaTeX source* 40c⟩.

One other problem peculiar to NoWeb text is that chunk references are reformatted in a way that assumes the chunk is defined in the same document. When documenting chunk names themselves, like this document does, this will clutter up the text with useless undefined chunk reference labels. Setting the undefined reference text to empty helps a little, but it leaves a blank space between the chunk name and the terminating bracket. Since this is a feature of `noweave`, it can't really be filtered out for all files. As a kludge, if no chunks are defined in a document at all, the reference markers are filtered out. Nothing is done for HTML output, since this only results in links that go nowhere.

41a ⟨*Post-process latex after weave* 41a⟩≡ (36d) 42c▷

```
(if grep '^<<.*>>=$' "$noweb" >/dev/null; then cat; else \
 sed 's/~{\\nwtagstyle{}[^}]*}//g'; fi) | \
```

Second, plain text chunks can be extracted, highlighted, and reinserted using a special input command as well. This has only one percent (vs. three) to distinguish it from the directive above:

```
\input{file.tex} % language
```

Rather than actually generate an additional file, the input directive is replaced with what would have been the contents of that file. This also allows the file to only contain contents given the source and any dependencies, rather than also including contents from sources which extend the contents. This is done after weaving, since the text is always literal, but a preprocessing step is required to avoid NoWeb substitutions during the weave. The sed script to generate the sed script cannot be done in a backtick expression, because the argument list may overflow. Generating the script to a file also reduces the number of backslash escapes required.

It seems pointless to highlight plain text files, but it may still be desirable to give it a colored background. For cases where this is not so, a special language `verbatim` can be used to instead insert the code as verbatim quoted plain text.

41b ⟨*Pre-process before weave* 39b⟩+≡ (36d 52a) ◁40b 42a▷

```
sed '/^\\input{.*\.tex} % [^ ]*$/{s/\[\[/@&/g;s/<</@&/g;}' | \
```

41c ⟨*For each reinserted code chunk* 41c⟩≡ (41d 56b)

```
sed -n -e 's/^\\input{\(.*\)\.tex} % \([^ ]*\)$/\2 \1/p' $noweb_order | sort -u | \
    while read -r lang f; do
      fesc=$(printf %s "$f" | sed 's/[][\\.*?^$]/\\&/g')
```

41d ⟨*Prepare for weave* 25c⟩+≡ (36d 50e) ◁39c 43d▷

```
⟨For each reinserted code chunk 41c⟩
  test verbatim = $lang || continue
  printf %s\\n "/^\\\\input{${fesc}\\.tex} % $lang\$/{s/.*/%/;a\\%\\"
  eval notangle '-R"$f"' $noweb_order 2>/dev/null | (
    echo '\begin{quote}\footnotesize\begin{verbatim}'
    sed 's/<</@<</g;s/>>/@>>/g;s/^@/@@/g'
    echo '\end{verbatim}\end{quote}'
    ) | sed 's/\\/\\\\/g;s/$/\\/' | sed '$s/.$//'
  echo '}'
done > /tmp/latexhl.verb.$$
⟨For each reinserted code chunk 41c⟩
```

[9] Actually, the hyperref package, the NoWeb HTML filter, and NoWeb code chunks all create automatic labels, but they are based on file location rather than the text at the label location, so they will never conflict.

41d ⟨*Prepare for weave* 25c⟩+≡ (36d 50e) ◁39c 43d▷

```
  test verbatim = $lang && continue
  printf %s\\n "/^\\\\input{${fesc}\\.tex} % $lang\$/{s/.*/%/;a\\%\\"
  eval notangle '-R"$f"' $noweb_order 2>/dev/null | \
    if [ -z "$uselst" ]; then
      ./latexhl $lang | ⟨Insert latexhl output directly 42b⟩
    else
      printf %s\\n '\begin{lstlisting}'
      ./addlistings $lang
      printf %s\\n '\end{lstlisting}'
    fi | sed 's/\\/\\\\/g;s/$/\\/' | sed '$s/.$//'
  echo '}'
done > /tmp/latexhl.fmt.$$
```

42a ⟨*Pre-process before weave* 39b⟩+≡ (36d 52a) ◁41b 44a▷

```
sed -f /tmp/latexhl.verb.$$ | \
```

42b ⟨*Insert latexhl output directly* 42b⟩≡ (41d 43d)

```
sed 's/^\\begin{framed}[^%]*/\\begin{quote}\\codemargin=0pt&\\begin{webcode}/;
     s/^\\end{framed}/\\end{webcode}&\\end{quote}/'
```

42c ⟨*Post-process latex after weave* 41a⟩+≡ (36d) ◁41a

```
sed -f /tmp/latexhl.fmt.$$ | \
```

C API documentation requires reinsertion of the function prototype, formatted as C code. Integration into a full API documentation package is beyond the scope of the current document (previous revisions had their own implementation of an API documentation program, but that was for Ada). Instead, only the prototype reinsertion is supported. To insert a prototype, reference it using a specially formatted comment:

```
% function prototype
```

This is done by gently extracting the prototype from `cproto.h` using sed. For prototypes which do not appear in `cproto.h`, the chunk ⟨*C Prototypes* 43e⟩ may be expanded as well, but only out of the same source file. Since the C preprocessor might transform functions before they become prototypes, the ⟨*For each reinserted C prototype* 43b⟩ chunk may be extended, transforming `$mf` as needed. The sed script separates arguments by comma and terminates by close parenthesis (which means that callbacks require typedefs). The code is reinserted with each argument on a separate line, aligned by the first argument, i.e. after the left parenthesis. This is done by converting the text before the parenthesis into spaces, and stripping off the text not to be printed every pass until there is nothing left to print.

42d ⟨*Format a prototype from* `cproto.h` 42d⟩≡ (43c)

```
# strip trailing semicolon to make pattern simpler (never gets readded)
s/;//;
# create spaces out of all text up to first paren and append full proto
h;s/[^(]/ /g;s/(.*/ /;G;
# remove up to first paren plus up to end of first parameter
s/[^ ][^(]*([^),]*[),] *//;
# strip off 2nd+ parameters & print first line
x;s/\(([^,)]*[,)]\).*/\1/;p;x;
# loop until no more saved text to print
:a
/^ *$/b;
# save, strip off 2nd+ paramters & print
h;s/\(^ *[^,)]*[,)]\).*/\1/;p;
# retrieve, strip off 1st parameter, and loop
g;s/[^ ][^),]*[),] *//;
ba;
```

43a ⟨*makefile.rules* 5b⟩+≡ ◁36c 57a▷

```
ifneq ($(CFILES),)
nw2latex: cproto.h
endif
```

43b ⟨*For each reinserted C prototype* 43b⟩≡ (43d 56b)

```
sed -f /tmp/latexhl.doc.$$ "$noweb" | sed -n -e 's/^% \(..*\) prototype$/\1/p' | \
  while read f; do
    mf="$f"
```

43c ⟨*Extract and format a C prototype from* `cproto.h` 43c⟩≡ (43d 56b)

```
sed -f /tmp/latexhl.doc.$$ "$noweb" | notangle -R"C Prototypes" 2>/dev/null | \
    cat - cproto.h | egrep "[^_[:alnum:]]$mf\\(" | head -n 1 | \
        sed -n -e '⟨Format a prototype from cproto.h 42d⟩'
```

43d ⟨*Prepare for weave* 25c⟩+≡ (36d 50e) ◁41d

```
⟨For each reinserted C prototype 43b⟩
    printf %s\\n "/^% $f prototype\$/{a\\%\\"
    ⟨Extract and format a C prototype from cproto.h 43c⟩ | \
      if [ -z "$uselst" ]; then
        ./latexhl C | ⟨Insert latexhl output directly 42b⟩
      else
        printf %s\\n '\begin{lstlisting}'
        ./addlistings C
        printf %s\\n '\end{lstlisting}'
      fi | sed 's/\\/\\\\/g;s/$/\\/' | sed '$s/.$//'
    echo '}'
done >> /tmp/latexhl.fmt.$$
```

43e ⟨*C Prototypes* 43e⟩≡

3.7 Include Directive Processing

Finally, a method to include an arbitrary NoWeb file is provided. The only way to weave and tangle such files correctly is to insert the contents of the file in place of the include directive. Since `\include` has other side effects (it forces a page break), only `\input` is supported. The syntax is very specifically defined in order to avoid conflicts with other `\input` directives. Namely, it must not have any following text, and the file name must end in `.nw`.

43f ⟨*Insert included NoWeb files* 43f⟩≡ (44a)

```
awk '⟨Insert included NoWeb files with awk 43g⟩'
```

43g ⟨*Insert included NoWeb files with* `awk` 43g⟩≡ (43f)

```
function readfile(f) {
  while (getline < f) {
    if ($0 ~ /^\\input{[^}]*\.nw}$/) {
      sub(".*{", ""); sub("}", "");
      fname = $0;
      if (index(fname, "/") != 0 && index(f, "/") >= 0) {
        fdir = f;
        sub("[^/]*$", "", fdir);
        fname = fdir fname;
      }
      readfile(fname);
      close(fname);
    } else
      print;
  }
}
BEGIN { readfile("-") }
```

44a ⟨*Pre-process before weave* 39b⟩+≡ (36d 52a) ◁42a

```
⟨Insert included NoWeb files 43f⟩ | \
```

4 HTML Code Documentation

The NoWeb file is intended to be written in LaTeX, and converted to HTML after and/or during the weave process.

4.1 HTML from TeX4ht

One way to generate HTML would be to use TeX4ht[10]. It is supposed to produce output very similar to what `pdflatex` produces, because it uses LaTeX as its underlying engine. Once again a wrapper script (`nwtex2html`) is needed.

44b ⟨*Source Code Documentation Files* 20d⟩+≡ (5d) ◁20d

```
$(patsubst %.nw,%.html,$(NOWEB_NINCL)) \
```

44c ⟨*makefile.rules tex4ht* 44c⟩≡ (57g)

```
%.html: %.tex $(FIGS_EPS) nwtex2html
        ./nwtex2html $< $@ "$(call tree,$<)" "$(HL_THEME)" "$(HLPROG_TYPE)"
```

44d ⟨*Clean built files* 9b⟩+≡ (5b) ◁22b 50d▷

```
rm -f *.html
```

A configuration file is needed to set extended options. In particular, the `\color` command behaves oddly, so an attempt is made to treat it mostly like `\textcolor` instead. Even then, the background color doesn't show through, so it's forced by placing all code the class `lstlisting` and setting the color in the style sheet. Overriding the code environments and doing some other fine tuning for TeX4ht is done in the preamble.

[10] `http://www.cse.ohio-state.edu/~gurari/TeX4ht/`

44e ⟨*myhtml.cfg* 44e⟩≡ (46b)

```
\Preamble{xhtml,uni-html4,css-in,$splitstyle,graphics-,Gin-dim+}
% attempted bug fix for tex4ht color
\HAssign\textcolornest=0
\makeatletter
\def\reset@color{
  \ifnum\textcolornest>0\gHAdvance\textcolornest by -1\HCode{</span>}\fi
  \special{color pop}
}
\makeatother
\Configure{color}{
   \gHAdvance\textcolorN by 1
   \HCode{<span id="textcolor\textcolorN">}
   \Configure{SetHColor}{%
   \gHAdvance\textcolornest by 1%
   \Css{span\#textcolor\textcolorN{color:\HColor}}}
}
% end bug fix
\embedfile{build.nw}%
\begin{document}
  \Css{.lstlisting {$(⟨Extract HTML CSS for current theme's background 45a⟩)
                    margin-left: 10pt; margin-right: 10pt;}}
\EndPreamble
```

45a ⟨*Extract HTML CSS for current theme's background* 45a⟩≡ (44e)

```
case "$HLPROG_TYPE" in
  highlight-[23])
    highlight -S c ${theme:+-s $theme} -I </dev/null | \
      sed -n -e '/pre.hl/{s/.*{//;s/}.*//;
                 s/\(background-color:#\)ffffff/\1fafafa/;
                 s/#/\\#/g;
                 s/font-size.*/font-size: smaller;/
                 p;}'
    ;;
  source-highlight)
    echo 'nodoctemplate
          "background-color:$docbgcolor; font-size: smaller;"
          end' > mybg.outlang
    source-highlight --lang-def=/dev/null --outlang-def=mybg.outlang \
                     ${theme:+--style-css-file=$theme.css} </dev/null | \
      sed 's/#ffffff/#fafafa/i;s/white/#f6f6f6/i'
    ;;
  *) echo "background-color:#fafafa; font-size: smaller;" ;;
esac
```

45b ⟨*preamble.tex* 37⟩+≡ (36d 52a) ◁39a 52c▷

```
\ifpdf
\else
\ifxetex
\else
% Following is for tex4ht only
\def\embedfile#1{}
\newenvironment{framed}{\HCode{<div class="lstlisting">}}{\HCode{</div>}}
%\renewenvironment{lstlisting}{\HCode{<div class="lstlisting">}}{\HCode{</div>}}
\noweboptions{webnumbering,nomargintag}

% Minimize mangling by tex4ht:
\lstset{columns=flexible}
% make spaces in chunk names into nbsp
\let\oldsetupmodname\setupmodname
\def\setupmodname{\oldsetupmodname\catcode`\ =13}
\fi
```

45b ⟨*preamble.tex* 37⟩+≡ (36d 52a) ◁39a 52c▷

```
\fi
```

The script just calls `htlatex` in its own directory (because it generates a lot of the same aux files as `pdflatex`), cleans up the HTML a bit, and tacks on the NoWeb source in compressed, uuencoded form. It depends on a working TeX4ht configuration, which is unfortunately mostly undocumented and easily broken. Configuring TeX4ht is beyond the scope of this document.

46a ⟨*Build Script Executables* 13c⟩+≡ (12h) ◁36b 49e▷

```
nwtex2html \
```

46b ⟨*nwtex2html* 46b⟩≡

```
#!/bin/sh
⟨Common NoWeb Warning 4c⟩

tex="$1"
base="${1%.tex}"
noweb="$base.nw"
outf="$2"
noweb_order="$3"
theme="${4#*:}"
export HLPROG_TYPE="$5"

⟨Check highlighter 25d⟩

figs=`sed -n 's/^\\\\includegraphics.*{\\(.*\\)}\$/\\1.eps/p' $tex`
texi=`sed -n -e 's/^\\\\input{\\(.*\\.tex\\)} % .*/\\1/p' "$tex"`

# Exit on error
set -e

# Done in its own dir to avoid stomping on .pdf aux files
dir=/tmp/html.$$
rm -rf $dir
trap "rm -rf $dir" 0
mkdir -p $dir
for x in $noweb $tex $figs $texi *.sty; do ln -s "$tdir/$x" $dir; done
tdir="`pwd`"
cd $dir

if [ y = "$HTML_SPLIT" ]; then
  splitstyle=frames,3
else
  splitstyle=fn-in
fi
cat <<EOF > myhtml.cfg
⟨myhtml.cfg 44e⟩
EOF

# runs latex 3x, regardless
htlatex $1 myhtml >&2

mv *.png "$tdir" 2>/dev/null || :
for x in *.html; do
  case "$x" in
    "$outf") is_outf=y ;;
    *) is_outf=n ;;
  esac
  perl -e '⟨Post-process TeX4ht output using perl 47b⟩' "$base" $is_outf <"$x" >"$tdir/$x"
done
```

46b ⟨*nwtex2html* 46b⟩≡

```
(
  ⟨Print attachments as HTML comment 47a⟩
) >>"$tdir/$outf"
```

47a ⟨*Print attachments as HTML comment* 47a⟩≡ (46b 52a)

```
# Attach original file as comment
# adds 64 to <>- to avoid HTML comment escapes
echo "<!--"
at="`sed -n -e 's/^\\\\embedfile.*{\([^}]*\)}%*$/\1/p' $noweb | sort -u`"
echo begin 644 ${noweb%.*}.tar.gz
eval tar chf - $noweb_order $at | gzip | \
  uuencode "${noweb%.*}".tar.gz | tail -n +2 | tr '<>-' '|~m'
echo "-->"
```

The HTML cleanup is done using a perl script, once again because awk and sed are too slow for at least one of the tasks.

47b ⟨*Post-process TeX4ht output using perl* 47b⟩≡ (46b)

```
use strict;

⟨Initialize TeX4ht post-processor 47c⟩

while (<STDIN>) {
  ⟨Post-process TeX4ht line 47d⟩
  print;
}
```

47c ⟨*Initialize TeX4ht post-processor* 47c⟩≡ (47b) 48a▷

```
my $is_outf = $ARGV[1] eq "y";
```

47d ⟨*Post-process TeX4ht line* 47d⟩≡ (47b) 48c▷

```
if (?^<html ?) {
  print "<!--\n";
  print "Generated using noweb+tex4ht;" .
        " original .nw source attached in comment at end\n";
  if ($is_outf) {
    print "\n";
    print "Extract with uudecode, or, if uudecode chokes, use:\n";
    print "   tr '\''|~m\''\'' '\''<>- '\'' | uudecode -o '\''$base.nw.gz'\''\n";
    print "-->\n";
  } else {
    print "of top-level HTML (see that for details on how to extract). -->\n";
  }
}
```

That task, in particular, is to compress the number of color styles. TeX4ht produces a new color style for every use of color, even if it is the same color as a previous use. To fix this, the color styles are extracted from the CSS file, and a hash array is built to associate each with the first style of the same color. Then, all duplicates are filtered from the CSS file and converted to the first version in the HTML.

47e ⟨*Copy TeX4ht CSS to stdout* 47e⟩≡ (48d)

```
# way too many color styles are generated: merge them
open CSS,"<$base.css";

while (<CSS>) {
  if (/^span#textcolor(\d+)(\{.*\})/) {
    my ($colorno, $colordef) = ($1, $2);

    $textcolor[$colorno] = $rootcolor{$colordef} || $colorno;

    if ($textcolor[$colorno] == $colorno) {
      $rootcolor{$colordef} = $colorno;
      print;
    }
  } else {
    print;
  }
}

close CSS;
```

48a ⟨*Initialize TeX4ht post-processor* 47c⟩+≡ (47b) ◁47c 48b▷

```
my $base = $ARGV[0];

my @textcolor;
my %rootcolor;
```

48b ⟨*Initialize TeX4ht post-processor* 47c⟩+≡ (47b) ◁48a

```
my $spn = "<span id=\"textcolor";
```

48c ⟨*Post-process TeX4ht line* 47d⟩+≡ (47b) ◁47d 48d▷

```
foreach my $x (/$spn(\d+)/g) {
  s/$spn$x"/$spn$textcolor[$x]"/g;
}
```

The next post-processing task is to reduce the number of files to one, if possible. This is done by in-lining the CSS. TeX4ht will do this with the right option, but only if run twice, which effectively runs LaTeX six times. Instead, it is just generated directly into the output file during hash table generation.

48d ⟨*Post-process TeX4ht line* 47d⟩+≡ (47b) ◁48c 48e▷

```
if(?<meta name=\"date?) {
  print;
  print "<style type=\"text/css\">\n<!--\n";
  ⟨Copy TeX4ht CSS to stdout 47e⟩
  print "//-->\n</style>\n";
  next;
}
```

The remainder of the adjustments are minor tweaks to the HTML output.

48e ⟨*Post-process TeX4ht line* 47d⟩+≡ (47b) ◁48d 48f▷

```
# tex4ht inserts a newline before each chunk
# no easy way to deal right now, so leave it
```

48f ⟨*Post-process TeX4ht line* 47d⟩+≡ (47b) ◁48e 49a▷

```
# tex4ht splits lines mid-tag; rejoin at least the span tags
while (/<span $/) {
  s/\n//;
  $_ .= <STDIN>;
}
```

49a ⟨*Post-process TeX4ht line* 47d⟩+≡ (47b) ◁48f 49b▷

```
# Merge consecutive spans on same line w/ same class into one span
while(s%(<span class="[^"]*">)([^<]*)</span>\1%\1\2%) {};
```

49b ⟨*Post-process TeX4ht line* 47d⟩+≡ (47b) ◁49a 49c▷

```
# Remove plain text spans
s%<span class="ecrm-0900">([^<]*)</span>%\1%g;
```

49c ⟨*Post-process TeX4ht line* 47d⟩+≡ (47b) ◁49b 49d▷

```
# I prefer mostly plain text for the web, so:
# remove ligatures
s/&#xFB01;/fi/g; s/&#xFB02;/fl/g;
# remove hex nbsp
s/ /\ /g;
# remove unicode quotes
s/&#8217;/'\"/g; s/&#8216;/`/g;
```

49d ⟨*Post-process TeX4ht line* 47d⟩+≡ (47b) ◁49c

```
# remove trailing space
s/ *$//;
```

4.2 HTML from l2h

Another way is to use NoWeb's included l2h filter. This once again requires some filtering, which is complicated enough that it's done in a separate script (nw2html) instead of inline in the makefile. In addition, highlighting of chunks must produce HTML, so a small conversion script (htmlhl) is used. This uses the ordered list output, as the ordered list CSS can be adjusted more easily to look like the LaTeX output.

49e ⟨*Build Script Executables* 13c⟩+≡ (12h) ◁46a

```
nw2html \
htmlhl \
```

49f ⟨*htmlhl* 49f⟩≡

```
#!/bin/sh
⟨Common NoWeb Warning 4c⟩

case $HLPROG_TYPE in
  highlight-[23]) echo '<ol class="code">' ;;
esac
⟨Do generic per-chunk highlighting preparation 31b⟩
case $HLPROG_TYPE in
  highlight-2)
    highlight -l --ordered-list -f -S $1 -E $HLDIR | \
      ⟨Filter highlight HTML output 50a⟩ ;;
  highlight-3)
    highlight -l --ordered-list -f $larg | \
```

49f ⟨*htmlhl* 49f⟩≡

```
        ⟨Filter highlight HTML output 50a⟩ ;;
    source-highlight)
      # remove last line and move /pre up
      # should probably retain one line if empty
      source-highlight -f xhtml-css $larg | \
        sed -n -e 's/<pre><tt>/<pre class="code">/;x;$s%%</pre>%;1!p'
esac
case $HLPROG_TYPE in
  highlight-[23]) echo '</ol>'
esac
```

50a ⟨*Filter* highlight *HTML output* 50a⟩≡ (49f)

```
sed 's%^</li>$%<li></li>%;s/<li[^>]*>/<li><pre>/;s%</li>%</pre></li>%'
```

50b ⟨*makefile.vars* 5d⟩+≡ ◁21a

```
FIGS_PNG:=$(patsubst %, %.png, $(FIGS))
```

50c ⟨*makefile.rules l2h* 50c⟩≡ (57g)

```
nw2html:  htmlhl nwweavefilt latexhl

%.html: %.nw $(FIGS_PNG) nw2html
        ./nw2html $< $@ "$(call tree,$<)" "$(HL_THEME)" "$(HLPROG_TYPE)"
```

50d ⟨*Clean built files* 9b⟩+≡ (5b) ◁44d

```
rm -f *.png
```

The wrapper script tacks on a similar header to what the TeX4ht script produces, but using the direct HTML output of `highlight` for the CSS, and making adjustments to make the HTML appear at least vaguely similar to the PDF.

50e ⟨*nw2html* 50e⟩≡ 52a▷

```
#!/bin/sh
⟨Common NoWeb Warning 4c⟩

noweb="$1"
outf="$2"
noweb_order="$3"
theme="$4"
export HLPROG_TYPE="$5"

# tex version allows list: prefix - ignore if there
theme="${theme#list:}"

⟨Prepare for weave 25c⟩

(
# This replaces l2h's header
echo "<html><!--
Generated using noweb; original .nw source attached as tar.gz in comment at end
Extract with uudecode, or, if uudecode chokes, use:
   tr '|~m\`' '<>- ' | uudecode -o '${noweb%.*}.tar.gz'
-->
 <head>
<meta name=\"generator\" content=\"noweave -html -filter l2h, $HLPROG_TYPE\">
  <style type=\"text/css\">
```

50e ⟨*nw2html* 50e⟩≡ 52a▷

```
  <!--"
case "$HLPROG_TYPE" in
  highlight-[23])
    highlight ${theme:+-s $theme} -S c -I </dev/null | sed -n -e '
      # replace white bg with very light gray to distinguish from text
      s/\(background-color:#\)ffffff/\1fafafa/g
      # remove extra junk from block style
      /pre.hl/{
        s/pre.hl/ .code/
        s/font-size:.*;/font-size: smaller;/
        p
        # make links look like normal text
        s/.code/& a/;s/margin.*;//
        p
      }
      # fix class names
      /^\./{s/^/.code /;s/\.hl\././;p;}'
    # option: line #s or not (leave both in HTML for user to see)
    echo ".code li {display: block;}"
    echo "/* either above line to remove line #s, or make space for line #s with: */"
    echo "/* .code {margin-left: 2em;} */"
    ;;
  source-highlight)
    ⟨Get all source-highlight color names 28a⟩
    source-highlight ${theme:+--style-css-file=$theme.css} --doc -f xhtml \
                     --lang-def=source-hl-elts.lang < source-hl-elts | \
      sed -n -e '
        # replace white bg with very light gray to distinguish from text
        s/\(background-color: *#\)ffffff/\1fafafa/ig
        s/\(background-color: *\)white/\1#fafafa/ig
        s/<body style="\(.*\)">/.code { \1; font-size: smaller }/p
        ta; :a s%</span>%%g;T
        s/\(.*>\)\([^>]*\)$/.code .\2 { \1 }/
        s/<span style="//g; s/">/; /g;p'
    ;;
esac
# option: border or not (leave both in HTML for user to see)
echo "/* optional: add border around code */"
echo "/* .code {border: thin solid black;} */"
# display code within chunk names correctly
echo "code {font-style: normal;}"
# display code chunk immediately below chunk name
echo "pre.defn {margin-bottom: 0;}"
# use same margins as LaTeX
echo ".code {margin-left: 10pt; margin-right: 10pt; margin-top: 0; padding: 1px}"
# squeeze lines together
echo ".code li,pre {margin-top: 0; margin-bottom: 0;}"
# make chunk refs more legible by removing underline
echo ".code a {text-decoration: none;}"
echo "pre.dfn a {text-decoration: none;}"
echo "//-->"
echo "</style>"
) >"$outf"
```

Next, it runs `noweave` with the `l2h` filter. This filter runs after the highlighter. The index generator tries to make switching to a code chunk also show the documentation by passing `-docanchor 10` to `noidx`. This places anchors 10 lines above the code chunk, regardless of what is actually there. This can be very bad, so instead, `@xref` is removed using a filter after `-index`, and then `nodix` is called directly using another filter, but without that `-docanchor` option. If readers want to read the documentation for a code chunk, they will need to just scroll up manually. Doing this eliminates the chunk reference without keeping a placeholder where the index should go, causing the reference to always appear at the end. To fix this, l2h must emit a different tag, which can then be replaced in the post-filter. In any case, post processing of

the HTML is required. After the document is generated, the source is once again tacked on as an HTML comment.

52a ⟨*nw2html* 50e⟩+≡ ◁50e 56b▷

```
( (
  ln='grep -n '^%%% latex preamble' "$noweb" | cut -d: -f1 | head -n 1`
  head -n $ln "$noweb"
  cat <<"EOF"
⟨preamble.l2h 53a⟩
% l2h substitution nowebchunks </nowebchunks>
% l2h substitution nowebindex </nowebindex>
⟨preamble.tex 37⟩
EOF
  ⟨Further l2h preamble 53b⟩
  tail -n +$((ln+1)) "$noweb"
) | ⟨Pre-process before weave 39b⟩
  noweave -html -filter "./nwweavefilt ./htmlhl $noweb_order" -filter l2h -index \
          -filter 'sed -n "⟨Filter out noidx results 52b⟩"' -filter noidx - | \
        ⟨Post-process HTML after weave 53c⟩
        cat

⟨Print attachments as HTML comment 47a⟩
) >>"$outf"
```

52b ⟨*Filter out noidx results* 52b⟩≡ (52a)

```
/^@nl/{
  h;
:a
  n;
  /^@nl/{H;ba};
  /^@index begin/{
    :b
      n;
      /^@index end/b;
      bb;
  };
  x;p;g;
};
/^@index begin/{
  :b
    n;
    /^@index end/b;
    bb;
};
s%</nowebindex>%<nowebindex>%;
s%</nowebchunks>%<nowebchunks>%;
/^@xref/!p
```

The LaTeX preamble is supplemented with directives for `l2h`, which are comments in the form `% l2h directive token arguments`. A new command, `\hypertxt`, is introduced as well, because doing hyperrefs with custom text requires an optional argument, and optional arguments are not supported by `l2h` directives. The RCS keyword commands must be inserted by extracting them from the original file, or they will end up coming from the build document every time.

52c ⟨*preamble.tex* 37⟩+≡ (36d 52a) ◁45b

```
\usepackage{comment}
\excludecomment{rawhtml}
% for l2h, since it doesn't understand opt args
\def\hypertxt#1#2{\hyperref[#1]{#2}}
```

53a ⟨preamble.l2h 53a⟩≡ (52a)

```
% Overrides for noweb's l2h
%
% Things it doesn't understand
% template is {, A=arg, [=optarg, C=optarg/save, ==assign, +=white, (=..)
% l2h macro hypertxt 2 <a href="###$1">#2</a>
% l2h macro url 1 <a href="#$1">#$1</a>
% l2h substitution ref * A{
% l2h envblock centering center
% l2h ignore discretionary {{{
% l2h ignore nwcodepenalty =
% l2h ignore edef A{
% l2h ignore excludecomment {
% l2h ignore the
% l2h ignore toks A={
% l2h ignore lstset {
% l2h ignore lstloadlanguages {
% l2h ignore lstdefinelanguage [[[[{
% l2h ignore definecolor {{{
% l2h ignore ifpdf
% l2h ignore ifxetex
% l2h ignore bkframefalse
% l2h ignore bkcounttrue
% l2h ignore ,
% l2h ignore break
% l2h substitution textbar |
% l2h substitution textasciitilde ~
% l2h substitution textasciicircum ^
% l2h substitution sim ~
% l2h substitution textbackslash \
% l2h substitution backslash \
% l2h ignoreenv webcode
% l2h macro input 1 <!-- input #$1 -->
% l2h ignore embedfile [{
% \RCS still needs to go into header to get stripped out properly
% l2h ignore RCS
% l2h ignore catcode
%
% doesn't understand extensionless image names, or that I want png
% l2h argblock includegraphics <img#src=" .png"><br/> [
%
% I prefer my footnotes shrunken and italicized
% l2h argblock footnote <font#size=2><b>[</b><em> </em><b>]</b></font>
%
% l2h ignore select@language {
```

53b ⟨Further l2h preamble 53b⟩≡ (52a)

```
sed -f /tmp/latexhl.doc.$$ "$noweb" | grep '^\\RCS \$' | while IFS= read -r l; do
  l="${l#*\$}"
  l="${l%\$*}"
  echo "% l2h macro RCS${l%%:*} 0 ${l#*:}"
done
echo "% l2h substitution jobname ${noweb%.*}"
```

The first required filter is one supplied with l2h: generating the hyperlinked table of contents.

53c ⟨Post-process HTML after weave 53c⟩≡ (52a) 54a▷

```
htmltoc -1234 | \
```

Next, the hidden sections must be removed. The top-level table of contents entries may as well be removed at the same time. These are always the document title and the abstract, which are not in the LaTeX table of contents, either.

54a ⟨*Post-process HTML after weave* 53c⟩+≡ (52a) ◁53c 54c▷

```
sed -n -e '⟨Filter noweave output with removal 54b⟩' | \
```

54b ⟨*Filter noweave output with removal* 54b⟩≡ (54a)

```
# strip out all before first header (title)
/<h1>/,${
  #strip out comment blocks delimited by <!--> & <--> on their own line
  # this is sort of unsafe, but it works for this document
  /^<!-->$/{
   :a
    n
    /^<-->$/bb
    ba
   :b
    n
  }
  #strip out the first top-level TOC - always title + abstract
  /^<tableofcontents>/{
   a\<h1>Table of Contents</h1>
   :c p;n;/^ *<li>/!bc
   :d n
     # allow one level of ul .. /ul
     /^ *<ul[ >]/{
       :e n;/^ *<\/ul>/!be
     }
     /^ *<li>/!bd
  }
  p
}
```

Then the title needs to actually be reinserted as the HTML document title.

54c ⟨*Post-process HTML after weave* 53c⟩+≡ (52a) ◁54a 55g▷

```
sed -e '⟨Filter noweave output without removal 54d⟩' | \
```

54d ⟨*Filter noweave output without removal* 54d⟩≡ (54c) 54e▷

```
# finish up head, copying 1st header as title
1{
  h
  s/h1>/title>/g
  s/<a name=[^>]*>//
  s%</a>%%
  s%/title>%&\n </head>\n<body>%
  p
  x
}
```

Some adjustments need to be made to code chunk headers. This allows them to be inset and gives them similar appearance to the LaTeX output.

54e ⟨Filter noweave output without removal 54d⟩+≡ (54c) ◁54d 55a▷

```
# Give style to pre directives around defn start, and limit pre to 1 line
/<dfn>/{
  s/<pre>/<pre class="dfn">/;s%$%</pre>%
 :a
  # newer versions of highlight added "hl " to the class names
  s/class="hl /class="/g
  # Use same bracket style as LaTeX (<sigh> w3m/links/IE hate this)
  # mathematical seems lowered, so use discouraged non-mathematical
  #s/<i>&lt;/<i>\&#x2329;/g; s%&gt;</i>%\&#x232A;</i>%g
  #s/<dfn>&lt;/<dfn>\&#x3008;/; s%&gt;\(.*</dfn>\)%\&#x3009;\1%
  n;/^<\/pre>/!ba
  s%^</pre>%%
}
```

Finally, there are some miscellaneous minor tweaks.

55a ⟨Filter noweave output without removal 54d⟩+≡ (54c) ◁54e 55b▷

```
# shrink & italicize "defined here" bits (from %def directives or autodefs)
s%\(<blockquote>\)\(Defines\)%\1<font size=1><em>\2%g
# it is hard to determine end of above group, so make HTML a litte bad
s%\(</blockquote>\)%</em></font>\1%g
```

55b ⟨Filter noweave output without removal 54d⟩+≡ (54c) ◁55a 55c▷

```
# strip out diagrams from index
# actually, it would better to strip out all hidden section chunks
s%^<li>.*\.dia&gt;</i></a>:.*%%
```

55c ⟨Filter noweave output without removal 54d⟩+≡ (54c) ◁55b 55d▷

```
# strip out *s on their own line - not sure what that is all about
s%<a name=[^>]*>\*</a>$%%
s/<br>\*$/<br>/
```

55d ⟨Filter noweave output without removal 54d⟩+≡ (54c) ◁55c 55e▷

```
# Yet another useful subst, but w3m/links/IE hate it
#s/---/&mdash;/g
```

55e ⟨Filter noweave output without removal 54d⟩+≡ (54c) ◁55d 55f▷

```
# l2h puts marks where hrefs point to, but that is not really necessary
s%<b>\[\*\]</b>%%g;s/\[\*\]//g
```

55f ⟨Filter noweave output without removal 54d⟩+≡ (54c) ◁55e 57b▷

```
# l2h does not handle quotes at all, so just remove duplicate
#s/``/\&#8216;/g;s/''/\&#8217;/g
s/``/`/g;s/''/'/g
```

One of those minor tweaks is to move the footnotes into their own section at the bottom of the document. Since `sed` is not up to the task, `awk` is used.

55g ⟨Post-process HTML after weave 53c⟩+≡ (52a) ◁54c

```
awk '⟨Move noweave filtering with awk 56a⟩' | \
```

56a ⟨*Move noweave filtering with awk* 56a⟩≡ (55g)

```
# May as well just move footnotes to the bottom
BEGIN {
  # uses FS to extract footnote text
  FS="<font size=2><b>.</b><em>|</em><b>.</b></font>";
  # odd is 1 if previous footnote spanned lines
  odd=0;
  # fn is the footnote counter
  fn=0;
  # fnt is the complete text of all footnotes
  fnt="";
}
# if NF > 1, a footnote has been detected
# odd is set to 1 if a footnote spans lines
NF > 1 || odd {
  # accumulate non-footnote text into ln
  ln="";
  i=0;
  if(odd) ft=ft "\n";
  while(i < NF) {
    i++;
    if(i > 1 && i % 2 == odd) {
      fn++;
      ln=ln "<sup><a href=\"#fn" fn "\">" fn "</a></sup>";
      ft=ft "\n<hr/>\n<a name=\"fn" fn "\"><b>" fn "</b>.</a> ";
    }
    if(i % 2 == odd) ft = ft $i; else ln = ln $i;
  }
  if(!odd || NF > 1) print ln;
  if(NF % 2 == 0) odd = 1 - odd;
}
# dump all footnotes at the end
$0 ~ "</body>" {
  if(fn > 0) print "<hr/><h3>Footnotes</h3>" ft
  print
}
# print everything that is not a footnote
NF <= 1 && !odd && ($0 !~ "</body>")
```

Copying in the highlighted code must be done after `noweave` is finished, or else it would need raw HTML sentinels. On the other hand, verbatim code should be copied in before weaving, and in fact it already has above.

56b ⟨*nw2html* 50e⟩+≡ ◁52a 57d▷

```
# insert highlighted extracted chunks/files
⟨For each reinserted code chunk 41c⟩
  test $lang = verbatim && continue
  printf '%s\\\n' '/<!-- input '"$fesc"'.tex --> <!-- '$lang'-->/{
       s/<!--.*-->//;i'
  eval notangle '-R"$f"' $noweb_order 2>/dev/null | ./htmlhl $lang | \
    sed '$!s/$/\\/'
  printf '}\n'
done >/tmp/latexhl.fmt.$$
# insert highlighted C prototypes
⟨For each reinserted C prototype 43b⟩
  printf '%s\\\n' "/<!-- $f prototype-->/{
       s/<!--.*-->//;i"
  ⟨Extract and format a C prototype from cproto.h 43c⟩ | ./htmlhl C | \
    sed '$!s/$/\\/'
  echo '}'
done >> /tmp/latexhl.fmt.$$
sed -i -f /tmp/latexhl.fmt.$$ "$outf"   # -i is GNU sed only
```

57a ⟨*makefile.rules* 5b⟩+≡ ◁43a 57e▷

```
nw2html: cproto.h
```

Since the sed scripts and simplicity of `l2h` can cause the HTML to not be standards compliant any more, HTML Tidy[11] is used to clean up the output.

57b ⟨*Filter noweave output without removal* 54d⟩+≡ (54c) ◁55f 57c▷

```
# tidy gives errors on unknown tags
s%</*tableofcontents>%%
```

57c ⟨*Filter noweave output without removal* 54d⟩+≡ (54c) ◁57b

```
# rather than trying to put TOC /li in right place, just remove them all
/href="#toc/s%</li>%%
```

57d ⟨*nw2html* 50e⟩+≡ ◁56b

```
tidy -q -wrap 0 -i -asxhtml -m "$outf" 2>tidy.log
# report unexpected errors
fgrep -ivf- tidy.log <<EOF >/dev/null || :
missing <!DOCTYPE>
unexpected </em>
unexpected </font>
replacing unexpected em
missing </a> before <a>
discarding unexpected </a>
lacks "summary" attribute
lacks "alt" attribute
column 18 - Warning: <a> anchor "NW
trimming empty
EOF
```

Generation of the figures depends on ImageMagick[12] with GhostScript support to convert them to PNG format, even though using GhostScript directly would have worked as well.

57e ⟨*makefile.rules* 5b⟩+≡ ◁57a 57g▷

```
%.png: %.eps
        convert -density 144x144 -comment " $< -transparent white $@
```

4.3 Method Selection

Unfortunately, neither formatter is obviously better. TeX4ht mangles its output a bit (color in particular) and takes forever to run. NoWeb's l2h filter is quicker, but is pretty stupid and unconfigurable for things like tables and footnotes. For now, I prefer the speed and accuracy of l2h.

57f ⟨*makefile.config* 5c⟩+≡ ◁35d

```
# Program to use for generating HTML formatted source code
# Set to tex4ht or l2h
HTML_CONV:=l2h
```

[11] http://tidy.sourceforge.net
[12] http://www.imagemagick.org

57g ⟨*makefile.rules* 5b⟩+≡ ◁57e

```
ifeq ($(HTML_CONV),l2h)
⟨makefile.rules l2h 50c⟩
else
ifeq ($(HTML_CONV),tex4ht)
⟨makefile.rules tex4ht 44c⟩
else
$(error Unknown HTML_CONV conversion method; use l2h or tex4ht)
endif
endif
```

5 Usage

Users of this document fall into at least two categories: ones who wish to create documents that use this build system, and ones who wish to build packages which were created using this build system. This section is for the former. Instructions for the latter were included on the first page, but should also be included in any document using this system:

```
\input{build-doc.tex} %%% doc
```

Of course the instructions for document users are required to build and test the project for document creators as well. After creating the makefile, you can use `make` to build and test. The standard targets are `all` (the default), `bin`, `install`, `doc`, `misc`, `clean`, `distclean`, `count`, `check`, `test-bin`, and `test`. In particular, the `check` target can help ensure that changes to a source file will not result in weird errors due to lost chunks or mistyped chunk names.

The first few lines of a NoWeb file are LaTeX code. This must at least include the `\documentclass` directive. It can also include the build system's preamble in the place marked by `%%% latex preamble`. Copying the beginning of this document to start with would not be a bad idea:

```
% Build with noweb:
%  notangle -t8 build.nw > makefile
%  make
\documentclass[twoside,english]{article}
\usepackage[letterpaper,rmargin=1.5in,bmargin=1in]{geometry}
%%% latex preamble
\RCS $Id$
\RCS $Revision$

\begin{document}

\title{...}
\author{...}
\date{Revision \RCSRevision}

\maketitle
```

Dependencies on other files should be listed somewhere (usually near the top), as well. They are specified by comments of the form `%%% requires` *name* on lines by themselves. Only direct dependencies need to be listed, as a full tree will be generated. The `.nw` extension is optional. For example, somewhere in your project, you will need to depend on this file:

```
%%% requires build
```

Chunks defined in a document extend chunks defined in documents on which it depends. Chunks defined in other documents can be used, as well. In both cases, they will not be weaved correctly. Chunks which are extended will be indicated with the original source file in parentheses at the start of the chunk name; this format is used mainly to move external referneces to the top of the chunk index and sort them by source name. However, the first extension will have no "previous chunk" link, rather than pointing to the source document's definition. Uses are also indicated with the original file name, but they always claim to be undefined. This is somewhat alleviated by the fact that undefined references are labeled with the word imported rather than undefined, but the only way to remove that indicator is to define the chunk elsewhere for extension (e.g. in an appendix) as neutral as possible (blank or a comment in the source language). This can be disabled for user-level documentation by adding the commment:

```
%%% no-ext-ref
```

Within the document, several extensions can be used:

- Language definitions: the listing package's `\lstset{language=`*lang*`}` directive is used to set the language for any subsequent code chunks. The default language is C.
- Hidden sections: any figures or other files which are normally included in the document in a processed form can be stored in-line in the NoWeb document in their raw form in hidden sections. Each hidden section is delimited by <!--> on its own line at the start, and <--> on its own line at the end. This is intended to be used as follows:

```
% hidden section - notangle chunks that shouldn't appear in printed output
% mainly for diagrams, which are extracted and included in visual form.
\begin{rawhtml}
<!-->
\end{rawhtml}
\iffalse
```

... hidden material ...

```
\fi
\begin{rawhtml}
<-->
\end{rawhtml}
```

  NoWeb doesn't actually strip these out — iffalse does that. Because NoWeb generates cross-reference information and the chunk index after the last chunk, the last chunk may not be commented out in this way. If you see a message like "LaTeX Warning: The are no \nowebchunks on input line ...", there are eiher no code chunks at all, or the last code chunk was commented out.

- Syntax highlighted, fully extracted code chunks: a code chunk can be repeated in e.g. a users' guide appendix, fully extracted and syntax highlighted, by using a special `\input` directive: `\input{`*chunk-name*`.tex} % `*language*. This directive will be replaced by the contents of the named chunk. For example, the configurable variables are included here using:

```
\input{makefile.config.tex} % make
```

```
# Installation prefix to apply to all install locations
DESTDIR:=
# Installation directory for binaries
BIN_DIR:=/usr/local/bin
# The name of the file containing the makefile
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#BUILD_NOWEB:=build.nw
# The name of the source files
# Any time you change this, makefile.* should be removed and rebuilt
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#NOWEB:=$(wildcard *.nw)
#Set to override the automatically determined project name
#PROJECT_NAME=
# Set to -L for #line, but lose code indentation
USE_LINE:=-L
# Where to find the non-standard tac command (GNU coreutils)
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#TACCMD:=sed -n -e '1!G;h;$$p'
# The noroots command is used to extract file names from $(NOWEB)
# The following works well enough in most cases if noweb is missing
# Note: due to Makepp restrictions, this can only be set on the command line
# or in makefile.config.local
#NOROOTS=sh -c "sed -n '/^<<.*>>=\$$/{s/=\$$//;p;}' \$$* \
#                                                  | sort -u" /dev/null
NOROOTS=noroots
# The PDF command to use: must be pdflatex or xelatex.
LATEXCMD=pdflatex
# note: to use attachments with xetex, the following may be necessary:
#LATEXCMD=env DVIPDFMXINPUTS=.:\$$TEXMF/dvipdfmx/ xelatex
# if non-blank, specify which highlighter to use:
#    highlight-2    named highlight, acts like highlight-2.x
#    highlight-3    named highlight, acts like highlight-3.x
#    source-highlight named source-highlight (probably 3.x)
# if blank, autodetect
HLPROG_TYPE=
# Highlight theme for PDF/HTML formatted source code
# Blank uses default highlight style; prefix w/ list: to use listings package
HL_THEME:=
# Program to use for generating HTML formatted source code
# Set to tex4ht or l2h
HTML_CONV:=l2h
```

One use of this feature which should be in every document is to list the RCS/CVS/subversion ID tag of all source files from which the document was built:

> This document was generated from the following sources, all of which are attached:
> `\input{Sources.tex} % txt`

- Syntax highlighted C prototypes: the prototype of a C function can be repeated as a highlighted code chunk using a special comment: `% `*function-name* `prototype` prints a reformatted version of the prototype extracted from `cproto.h`, which is automatically generated for all C files. Any prototypes which would not be placed there can be manually added using the ⟨*C Prototypes* 43e⟩ chunk; all prototypes from the chunk must consist of exactly one line of text.

- Repeated documentation: a separate NoWeb-like macro facility which only does substitutions can be used to repeat documentation in e.g. a users' guide in an appendix. Such documentation chunks are delimited by `% Begin-doc` *chunk-name* and `% End-doc` *chunk-name* at the beginning of a line. They are inserted in place of any occurence of `\input {`*chunk-name*`.tex} %%% doc` at the beginning of a line.

- Simple input directives: a file can be included verbatim using `\input{`*file*`.nw}`. This actually substitutes the file in place rather than doing an actual include so that weaving and tangling operate correctly.
- Documentation attachments: any files which should be attached to the PDF or HTML output can be attached using the `embedfile` package's `\embedfile` directive.

Somewhere in the document, the code chunk index should be added. Right now, an identifier index is not supported.

```
\section{Code Index}
\nowebchunks

% note: no identifier indexing is done right now

\begin{rawhtml}
<!-->
\end{rawhtml}
%\vspace{1ex}
%\hrule
%\vspace{1ex}
\begin{rawhtml}
<-->
\end{rawhtml}

%\nowebindex
```

Of course the document must be ended with `\end{document}`, like any LaTeX document.

```
\end{document}
```

Once the documentation has been built, the PDF output needs to be checked for overflows. Horizontal overflows in code chunks are hard to detect, as they do not always generate `Overfull \hbox` messages. They can be corrected by reformatting the source code. Vertical overflows as the result of problems with the `framed` package can be detected by `Overfull \vbox` messages, and can be corrected by prefixing chunks which are not broken at the end of a page with a `\break` directive. Naturally, any edits to the document will require rechecking all `\break` directives to see if they are still necessary, and possibly to add more.

Since I write a lot of C, a few special tricks are available for C programs. In addition to the C-specific chunks described below, a special header is generated, `cproto.h`, which contains all exported prototypes, except for prototypes already explicitly included in headers. This is always included after ⟨*Common C Includes* 19c⟩ in the ⟨*Common C Header* 19e⟩. In order to facilitate writing static functions out of order, a static prototype file is generated for each C file as well. This is included right after `cproto.h`, unless the comment `// static_proto` is found on a line by itself, in which case it is placed after that comment. That way, the static prototypes can be placed after any private data type declarations. As mentioned above, `cproto.h` is used for API documentation as well.

Several internal variables are meant to be set or expanded in extensions to ⟨*makefile.vars* 5d⟩:

- `C_POSTPROCESS` is a pipeline to apply to all automatically extracted C files. This is a function taking the file name as an argument (which can be ignored if there are no file-specific transformations). The return value (or value of the plain variable, since there is no difference between functions which ignore their arguments and variables) should be a pipeline element, starting with the pipe symbol. For example:

  ⟨*makefile.vars* 5d⟩=
  ```
  C_POSTPROCESS+=|sed '1i\copyright blurb'
  @
  ```

- The `EXTRA_CFLAGS` variable should be extended with required options; that way, the user can replace `CFLAGS` with just optimization and debugging flags.
- The `EXTRA_LDFLAGS` variable should be extended with required options; that way, the user can replace `LDFLAGS` with just optimization and debugging flags.

Finally, extensions to the build system can be made by adding code chunks; they will be appended to existing values in the order of the tree, from the most depended on file to the least depended on file (i.e., anything depending on `build.nw` will append values *after* the values set in `build.nw`):

Basic build rules:

- ⟨*makefile.config* 5c⟩: add user-configurable variables to the makefile.
- ⟨*makefile.vars* 5d⟩: add non-configurable variables to the makefile.
- ⟨*makefile.rules* 5b⟩: add rules to the makefile.
- ⟨*Install other files* 5e⟩: add installation commands.
- ⟨*Clean temporary files* 9c⟩: add commands to clean temporary files.
- ⟨*Clean built files* 9b⟩: add commands to clean built files.
- ⟨*Plain Files* 5f⟩: add plain files to build; always add a backslash to the end of each added line. If additional processing other than simple extraction is required for any file, it should instead be added to ⟨*makefile.rules* 5b⟩ as a separate rule and a dependency for the `misc` target. If any plain files are not in the NoWeb source, but instead are distributed separately, they should instead be added to the `ATTACH_EXTRA` variable, to be attached to the documentation. That variable can also be a make function taking the name of the file being attached to as an argument.
- ⟨*Plain Build Files* 8d⟩: add plain text files to be built when required as a dependency; always add a backslash to the end of each added line. This is not added to any explicit target. It is merely a shorthand to create a rule for extracting it from NoWeb and cleaning it up afterwards. It is intended for build support files that are not meant to be distributed.
- ⟨*Script Executables* 12e⟩: add plain text files to build as executables; always add a backslash to the end of each added line. Like ⟨*Plain Files* 5f⟩, no processing is done by the default rules.
- ⟨*Build Script Executables* 13c⟩: add plain text files to build as executables when required; always add a backslash to the end of each added line. This is not added to any explicit target. It is merely a shorthand to create a rule for extracting it from NoWeb and cleaning it up afterwards. It is intended for build support files that are not meant to be distributed.
- ⟨*Test Scripts* 15c⟩: add plain text files to build as executables, but only for the `test` and `test-bin` targets. The `test` target will execute these scripts without any interpretation other than return code checks.
- ⟨*Test Support Scripts* 15f⟩: names of executable scripts to build for the `test-bin` target. These scripts are not run, but are guaranteed to be built before running any tests.
- ⟨*Additional Tests* 15h⟩: tests that are more complicated to run than simple scripts. These are makefile actions.
- ⟨*Build Source* 11e⟩: add files to the NoWeb-free tar distribution; always add a backslash to the end of each added line. In particular, any specially processed plain files with their own rules should be added to this list.

- ⟨*Files to Count* 12b⟩: files to count as part of the `count` target; always add a backslash to the end of each added line. Anything added to the ⟨*Build Source* 11e⟩ should probably be added here, as well.
- ⟨*Executables* 11a⟩: additional specially-built execuatables, other than scripts and C executables.

C-Specific rules

- ⟨*C Files* 11c⟩, ⟨*C Headers* 11d⟩: C files which are NoWeb roots are automatically found, but generated files are not. Add these with these chunks, terminated by a backslash.
- ⟨*C Executables* 12f⟩: C files are either the name of an executable with the `.c` extension, or the name of a library member, with `.o` replaced by `.c`. The executables are listed here, without the `.c` extension, but with a trailing backslash on every line. Technically, this could be considered a list of generated executables instead, since there is no rule requiring that these be C.
- ⟨*C Test Executables* 15b⟩: names of executables to build for the `test` and `test-bin` targets. See ⟨*C Executables* 12f⟩ and ⟨*Test Scripts* 15c⟩ for details.
- ⟨*C Test Support Executables* 15e⟩: names of executables to build for the `test-bin` target, as explained by ⟨*C Executables* 12f⟩ and ⟨*Test Support Scripts* 15f⟩ for details.
- ⟨*C Build Executables* 13d⟩: names of executables to build when required by other dependencies. See ⟨*C Executables* 12f⟩ and ⟨*Build Script Executables* 13c⟩ for details.
- ⟨*Library* `name` *Members* (imported)⟩: A library named `name` will be built using objects listed in these chunks. `name` can of course be anything appropriate. Unlike the chunks which expand into specific `make` variables, no backslashes are required or allowed at the end of each line.
- ⟨*For each reinserted C prototype* 43b⟩: add a transformation for the modified function name (`$mf`). Typically, this will involve invoking the C preprocessor with a few standard include files, followed by the function name. For example:

⟨*makefile.vars* 5d⟩=
```
export MY_CPP=$(CC) $(CFLAGS) $(EXTRA_CFLAGS) -E
@
```

⟨*For each reinserted C prototype* 43b⟩=
```
    mf="`printf '⟨Common C Includes 19c⟩\\n%s' \"\$mf\" |
            eval \"\$MY_CPP -\" | tail -n 1`"
@
```

Boilerplate

- ⟨*Common NoWeb Warning* 4c⟩: a boilerplate comment for the top of script files; extend with `# $Id$`.
- ⟨*Version Strings* 19f⟩: A C string containing the version ID of all contributing source files; extend with `"$Id$\n"`.
- ⟨*Sources* 5a⟩: A simple text chunk containing the version ID of all contributing source files; extend with just `$Id$`.
- ⟨*Common C Warning* 19d⟩: the same comment enclosed in a C comment; not meant to be extended
- ⟨*Common C Includes* 19c⟩: a list of `#include` directives for files safe to be included by all C source

- ⟨*Common C Header* 19e⟩: stuff safe for the top of every C file, including the warning, common includes, static version string, and automatically generated prototypes. Prototypes for static functions for the file are included as well, unless a comment of the form `// static_proto` is found on a line by itself; in that case, the static prototypes will be included there instead.
- ⟨*Common Perl Prefix* 22d⟩: stuff to insert at the top of perl executables. This includes the NoWeb warning, a portable hash-bang prefix, and `use strict`.

Highlighting Assistance

- ⟨*Known Data Types* 33d⟩: comma-separated list of data types to emphasize; end each line (including the last) with a comma and a percent-sign.
- ⟨*Translate listings source type to source-highlight source type* 31c⟩, ⟨*Translate listings source type to highlight-3 source type* 31g⟩: add new cases to a shell case switch to convert listings language names to highlighter language names.

6 Code Index

⟨*Post-process TeX4ht output using perl* 47b⟩ 46b, 47b
⟨*Pre-process before weave* 39b⟩ 36d, 39b, 40a, 40b, 41b, 42a, 44a, 52a
⟨*Preamble Adjustments for* `listings` 33a⟩ 33a, 33b, 33c, 34b, 34c, 38c
⟨*preamble.l2h* 53a⟩ 52a, 53a
⟨*preamble.tex* 37⟩ 36d, 37, 38a, 38b, 38c, 39a, 45b, 52a, 52c
⟨*Prepare for* `latexhl` 26d⟩ 26d, 27c, 27e, 36d
⟨*Prepare for weave* 25c⟩ 25c, 29, 33e, 36d, 36e, 39c, 41d, 43d, 50e
⟨*Print attachments as HTML comment* 47a⟩ 46b, 47a, 52a
⟨*Process lines from NoWeb token stream* 22g⟩ 22e, 22g, 23a
⟨*Reduce interchunk whitespace* 35c⟩ 35c, 36d
⟨*Remove duplicate labels from LaTeX source* 40c⟩ 40b, 40c
⟨*Remove makefile* 6c⟩ 5b, 6c
⟨*Script Executables* 12e⟩ 11b, 12e
⟨*Source Code Documentation Files* 20d⟩ 5d, 20d, 44b
⟨*Sources* 5a⟩ 5a
⟨*Strip down C prototype* 19b⟩ 18a, 19b
⟨*Test Executables* 15a⟩ 14e, 15a
⟨*Test Scripts* 15c⟩ 14e, 15c
⟨*Test Support Executables* 15d⟩ 14e, 15d
⟨*Test Support Scripts* 15f⟩ 14e, 15f
⟨*Translate listings source type to highlight-3 source type* 31g⟩ 31f, 31g
⟨*Translate listings source type to source-highlight source type* 31c⟩ 31b, 31c
⟨*Version Strings* 19f⟩ 19e, 19f

7 TODO

This is an unorganized TODO list for my own amusement. It in no way represents a promise to get anything done; sometimes I completely reverse my position on these items, and some have been lingering on my TODO list for years.

7.1 Not missing

- Plain TeX: not really an option, even though NoWeb supports it. Porting framed and listings from LaTeX is beyond the scope of this project. Most of the highlighters I find also expect to be used with LaTeX.
- texinfo: Way too TeX-depedent; see above item about that. It's not really as output-independent as claimed; much of its independence comes from having to use output-specific escapes everywhere. Most output independence I need is HTML vs. PDF, and I get that with latex-to-HTML converters with about as much utility as texinfo provides. And the texinfo format itself (having to repeat section titles in node definitions and manually insert TOC entries as menus) is not that appealing (although I could automate some of that).
- docbook: probably not possible, but should look at it since it's popular. It's definitely a pain in the ass raw; like all XML formats, it's not meant to be edited by a plain text editor or read/understood by humans (even though the XML movement started with the exact opposite claims).
- asciidoc: docbook in a way humans can edit it, but still can't produce PDF easily or correctly.
- reStructuredText: slightly better than asciidoc; may be worth a look some day.

7.2 Missing

- bibtex, makeindex for l2h. bibtex requires running latex and bibtex once, and makeindex requires running latex and makeindex and also providing support for reading the .ind and formatting it correctly.
- optional automatic symbol index (requires a lot of work; the symbol extractor for NoWeb is way too simplistic and in fact anything short of a compiler can't deal with name spaces and other issues).

7.3 Needs testing

- tex4ht is an undocumented piece of crap (well, I guess the original author being dead may have something to do with that, but I doubt the situation would be any different were that not the case) that is flaky and in fact the last release does not work any more and I don't have the energy to try and fix it.
- I don't have access to commercial UNIX any more, so I don't know what does or does not work. In particular, I use GNU sed's `-i` option too much now. I should probably just convert all that to perl or something like that that people regularly install on systems.

7.4 Improvements

- Go back to explicit import/export: maybe just require a prefix like @ for all globals.

  How:

  - In tangle, prepend source name to all nodes that don't have @ prefix in a filter:

    ```
    @file <fn> -> set file
    @defn <x> -> convert <x> if not @-prefixed
    @use <x> -> convert <x> if not @-prefixed
    ```

  - In weave, remove @, but add "(exported)" to node name if first defined in this file, or "(source-file.nw)" to node name if defined elsewhere. Note that r77 already implemented the latter, but could not implement the former because exports are not specially flagged. Perhaps printing the first file that defines the symbol is not so good: the last file before this would be better, as any extensions would be included, and presumably that file then links to the previous file in the chain anyway. On the other hand, documentation for the symbol is likely only in the first.
  - Would need backend work to support hyperlinking to other documents, although adding a label in front of the first definition of an exported symbol and then modifying the chunk index and the first chunk's previous reference somehow. That would be nearly impossible with split HTML, though, unless the split HTML were generated first.

- Related to above: it might be possible to come up with alternate methods of combining globals:

  ```
  Global import/export requires prefix char:
   <<+...>>= -> append to inherited global
   <<-...>>= -> prepend to inherited global
   <<=...>>= -> replace inherited global
   <<!...>>= -> same as <<+, but disallow replacement
   <<@...>>= -> reference to a global (import only)
  ```

  How:

  - Actual symbol name is renamed to <sourcename>+<node_without_prefix> in tangle
  - If an inherited global is not defined in a file, the symbol is inherited from parent by defining a single chunk with the parent's chunk as contents. In other words, as if an empty «+...»= chunk defined.

  - In one file, can only use one prefix. Same node name with two different prefixes raises error and refuses to tangle/weave.
  - If any file declares !, and later one decalres =, raises error and refuses to tangle/weave.
  - in tangle, insert parent's global after first +.
  - in tangle, insert parent's global after end of last -.
  - in tangle, ignore parent's global if =.
  - in tangle, replace all @-references with top-level source file's version

- Switch to something other than GNU make. GNU make shows bugs: sometimes dependencies are just ignored, especially in parallel builds where they are most important. Makepp is a little better in some respects, but is only stopgap. Something like Odin would be best, but Odin needs a lot of work.

- Automatically set CreationDate and ModDate for PDF to RCSDate. Need to actually find all dates of all relevant files and use most recent. svn-multi will do that automatically in LaTeX, but I need to do it manually for l2h. Even for svn-multi, I need to tack all svnid commands from depenencies. Probably not that useful, since even if ModDate and CreationDate are both static, multiple runs will still always produce different results. It is apparently impossible to stabilize the output or filter out just the time-sensitive parts.

  ```
  \hypersetup{pdfinfo={CreationDate=D:\svnpdfdate}}
  \hypersetup{pdfinfo={CreationDate=D:YYYYMMDDHHMMSS-hh'mm')}
  ```