

# Make-ing Life Easy: A General Makefile Framework

Daniel Bosk

School of Electrical Engineering and Computer Science  
KTH Royal Institute of Technology, Stockholm

Department of Information Systems and Technology  
Mid Sweden University, Sundsvall

16th December 2025

Copyright (c) 2011–2022 Daniel Bosk

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

# Contents

<b>1</b>	<b>Introduction and usage</b>	<b>5</b>
1.1	Makefile for this framework . . . . .	5
1.1.1	This <b>Makefile</b> 's target . . . . .	6
1.1.2	Tangling, weaving and compiling LaTeX . . . . .	7
1.1.3	The Docker image . . . . .	8
1.1.4	Packaging and publication . . . . .	9
<b>I</b>	<b>General building blocks</b>	<b>10</b>
<b>2</b>	<b>portability.mk</b>	<b>11</b>
2.1	Introduction . . . . .	11
2.2	Standard Unix commands . . . . .	12
2.2.1	File system commands . . . . .	12
2.2.2	Viewing file contents . . . . .	12
2.2.3	Filtering and transformations . . . . .	13
2.2.4	Statistics . . . . .	13
2.3	Networking commands . . . . .	13
2.4	Compressed files and archives . . . . .	14
2.4.1	Compressing and uncompressing files . . . . .	14
2.4.2	Packing and extracting from archives . . . . .	15
<b>3</b>	<b>subdir.mk</b>	<b>18</b>
3.1	Introduction and usage . . . . .	18
3.2	Implementation . . . . .	18
<b>II</b>	<b>Packaging and publishing</b>	<b>20</b>
<b>4</b>	<b>pkg.mk</b>	<b>21</b>
4.1	Introduction and usage . . . . .	21
4.1.1	Portability . . . . .	22
4.2	Implementation . . . . .	22
4.2.1	Packaging . . . . .	23

4.2.2	Cleaning . . . . .	23
4.2.3	Installation . . . . .	24
<b>5</b>	<b>pub.mk</b>	<b>26</b>
5.1	Introduction and usage . . . . .	26
5.1.1	Specifying files . . . . .	26
5.1.2	Automatically tag on publication . . . . .	26
5.2	Configuration for publishing files on a server, <code>upload</code> . . . . .	27
5.2.1	Publication methods . . . . .	28
5.2.2	Publishing to multiple sites . . . . .	29
5.3	Implementation . . . . .	29
5.3.1	The upload publication mechanism, <code>upload</code> . . . . .	30
5.3.2	Publication methods . . . . .	31
5.3.3	GitHub releases, <code>gh-release</code> . . . . .	34
5.3.4	Automatically committing and tagging, <code>autotag</code> and <code>autocommit</code> . . . . .	34
<b>6</b>	<b>transform.mk</b>	<b>37</b>
6.1	Introduction and usage . . . . .	37
6.2	Implementation overview . . . . .	37
6.3	A transformation mechanism . . . . .	38
6.3.1	Removing solutions . . . . .	39
6.3.2	Removing excessive build instructions . . . . .	39
6.3.3	Handouts and solutions . . . . .	39
6.4	Preparing camera-ready source . . . . .	40
6.5	Using encrypted files . . . . .	41
<b>III</b>	<b>Papers and documents</b>	<b>43</b>
<b>7</b>	<b>tex.mk</b>	<b>44</b>
7.1	Introduction and usage . . . . .	44
7.2	Implementation overview . . . . .	46
7.3	Targets for documents . . . . .	47
7.3.1	Auxillary files . . . . .	48
7.3.2	Bibliographies . . . . .	48
7.3.3	Indices . . . . .	49
7.3.4	PythonTeX . . . . .	50
7.3.5	Document files . . . . .	52
7.4	Targets for merging bibliographies . . . . .	52
7.5	Targets for class and package files . . . . .	53
7.6	External classes and packages . . . . .	53
7.6.1	Springer LNCS . . . . .	55
7.6.2	LNCS style for <code>biblatex</code> . . . . .	56
7.6.3	The RFC bibliography . . . . .	56
7.6.4	Proceedings of the Privacy Enhancing Technologies Symposium . . . . .	57

<b>8</b>	<b>doc.mk</b>	<b>58</b>
8.1	Introduction and usage . . . . .	58
8.2	Implementation . . . . .	58
8.2.1	Printing . . . . .	59
8.2.2	Counting words . . . . .	59
8.2.3	To-do lists . . . . .	60
8.2.4	Format conversion . . . . .	60
<b>IV</b>	<b>Literate programming</b>	<b>66</b>
<b>9</b>	<b>noweb.mk</b>	<b>67</b>
9.1	Introduction and usage . . . . .	67
9.2	Implementation . . . . .	67
9.2.1	Weaving documentation . . . . .	68
9.2.2	Tangling code . . . . .	69
<b>10</b>	<b>haskell.mk</b>	<b>74</b>
10.1	Introduction, usage and implementation . . . . .	74
<b>V</b>	<b>Assessment</b>	<b>75</b>
<b>11</b>	<b>exam.mk</b>	<b>76</b>
11.1	Introduction and usage . . . . .	76
11.2	Implementation . . . . .	77
11.2.1	Generating targets for exam TeX files . . . . .	78
11.2.2	Substituting fields from template . . . . .	78
11.2.3	Generating targets for exam PDFs . . . . .	78
11.2.4	Generating targets for exam questions . . . . .	79
<b>12</b>	<b>results.mk</b>	<b>81</b>
12.1	Introduction and usage . . . . .	81
12.1.1	Portability . . . . .	83
12.2	Processing Moodle's output . . . . .	83
12.2.1	Transforming Moodle's output . . . . .	83
12.2.2	Extracting the changes . . . . .	84
12.2.3	Extracting identifiers for reporting . . . . .	85
12.2.4	Generating the report . . . . .	86
12.3	Sending and storing the results . . . . .	87
<b>VI</b>	<b>Appendices</b>	<b>88</b>
<b>A</b>	<b>Dockerfile: an execution environment</b>	<b>89</b>
	<b>Bibliography</b>	<b>92</b>

# Chapter 1

## Introduction and usage

This chapter will introduce the framework and introduce it by examples. In Section 1.1, we examine the `Makefile` for this framework, i.e., to build (weave) this documentation and (tangle) all the include files from their literate (NOWEB) source files. This `Makefile` uses a few parts of the framework, but not everything. In subsequent sections we will cover the remaining parts.

### 1.1 Makefile for this framework

The `Makefile` for this repository should provide instructions to make all the files needed by this framework along with its documentation (`makefiles.pdf`).

```
5  <Makefile 5>≡ 6a>
    MKFILES+=    portability.mk subdir.mk
    MKFILES+=    pkg.mk pub.mk transform.mk
    MKFILES+=    tex.mk doc.mk
    MKFILES+=    noweb.mk haskell.mk
    MKFILES+=    exam.mk results.mk

    OTHERS+=     latexmkrc
    OTHERS+=     gitattributes
    OTHERS+=     Dockerfile

    .PHONY: all
    all: makefiles.pdf
    all: ${MKFILES}
    all: ${OTHERS}

<Makefile target 6e>
<Dockerfile target 8d>
<makefiles.pdf target 7a>
<MKFILES targets 7e>
```

*<OTHERS targets 8b>*

It also provides targets for creating a Docker image from the `Dockerfile` and pushing it to Docker Hub.

6a *<Makefile 5>+≡* <5 6b>  
    DOCKER\_ID\_USER?=dbosk

```
.PHONY: docker-makefiles push
docker-makefiles: Dockerfile
    <Docker image recipe 8e>
```

```
push: docker-makefiles
    <Docker push recipe 8f>
```

We also use the package framework (`pkg.mk`) to provide a package which can install the makefiles on the system.

6b *<Makefile 5>+≡* <6a 6c>  
    *<package setup 9b>*

We also need the standard targets for cleaning and some include files.

6c *<Makefile 5>+≡* <6b 6d>  
    .PHONY: clean distclean  
    clean:  
        *<clean recipe 7b>*

```
distclean:
    <distclean recipe 8g>
```

Finally, we need some include files from this very framework.

6d *<Makefile 5>+≡* <6c  
    INCLUDE\_MAKEFILES=.  
    MAKEFILES\_INCLUDE=\${INCLUDE\_MAKEFILES}  
    *<include files 7d>*

### 1.1.1 This Makefile's target

We also must add a target for this `Makefile` itself.

6e *<Makefile target 6e>≡* (5)  
    Makefile: Makefile.nw  
        \${NOTANGLE.mk}

### 1.1.2 Tangling, weaving and compiling LaTeX

The documentation depends on the main TeX file, preamble and bibliographies.

```
7a <makefiles.pdf target 7a>≡ (5)
    <flags 7c>
    makefiles.pdf: makefiles.tex preamble.tex makefiles.bib
    makefiles.pdf: intro.tex Makefile.tex
    makefiles.pdf: exam.bib
    makefiles.pdf: transform.bib
    makefiles.pdf: tex.bib
    makefiles.pdf: Dockerfile.tex
```

We must add the generated files to the clean recipe.

```
7b <clean recipe 7b>≡ (6c) 8a>
    ${RM} makefiles.pdf
    ${RM} Dockerfile.tex
```

We want to use the PythonTeX and Minted packages. This requires some flags to LaTeX.

```
7c <flags 7c>≡ (7a)
    LATEXFLAGS+= -shell-escape
    TEX_PYTHONTEX= yes
```

To automatically compile the documentation, we use the `tex.mk` include. Once this include is present, make will be able to make a `.pdf` from a `.tex` file.

```
7d <include files 7d>≡ (6d) 7f>
    include ${MAKEFILES_INCLUDE}/tex.mk
```

Additionally, it depends on `.tex` versions of all the `.mk` files. We will generate the targets for both `.tex` and `.mk` versions, both of these are generated from the `.nw` file. The functionality of `noweb.mk` allows us to reduce this to the line equivalent to `%.mk %.tex: %.nw` below. Then the pattern rules will do the work.

```
7e <MKFILES targets 7e>≡ (5)
    define makefiles_depends
    makefiles.pdf: $(1:.mk=.tex)
    $(1) $(1:.mk=.tex): $(1).nw
    endef

    $(foreach mkfile,${MKFILES},${(eval $(call makefiles_depends,${mkfile}))})
```

To automatically tangle an `.nw` file into an `.mk`, we use the `noweb.mk` include. Once this is present, make knows how to make an `.mk` file from an `.nw`.

```
7f <include files 7d>+≡ (6d) <7d 9a>
    include ${MAKEFILES_INCLUDE}/noweb.mk
```



We should also add these generated file to the cleaning recipe. However, we only remove the `.tex` files, we keep the tangled `.mk` files to use the repo as an independent Git submodule in project repos.

8a `<clean recipe 7b>+≡` (6c) <7b 8c>  
`${RM} ${MKFILES:.mk=.tex}`

We also have files for which this doesn't work (the target doesn't have the name in common with its source). Then we can provide a one-liner recipe, also thanks to `noweb.mk`.

8b `<OTHERS targets 8b>≡` (5)  
`latexmkrc: tex.mk.nw`  
`${NOTANGLE}`

`gitattributes: transform.mk.nw`  
`${NOTANGLE}`

Of these files, we only want to keep `latexmkrc` after cleaning.

8c `<clean recipe 7b>+≡` (6c) <8a 9c>  
`${RM} gitattributes`

### 1.1.3 The Docker image

The `Dockerfile` is used to produce a Docker image.

8d `<Dockerfile target 8d>≡` (5)  
`Dockerfile: Dockerfile.nw`  
`${NOTANGLE}`

We provide a phony target `docker-makefiles` to produce the `makefiles` Docker image (see above).

8e `<Docker image recipe 8e>≡` (6a)  
`docker build -t makefiles .`  
`docker tag makefiles ${DOCKER_ID_USER}/makefiles`

We also provide a phony target `push` to push the `makefiles` image to Docker Hub.

8f `<Docker push recipe 8f>≡` (6a)  
`docker push ${DOCKER_ID_USER}/makefiles`

Lastly, we need to cleaning. We provide a `distclean` target to remove the (quite sizeable) Docker image.

8g `<distclean recipe 8g>≡` (6c)  
`docker image rm makefiles`  
`docker image rm dbosk/makefiles`

### 1.1.4 Packaging and publication

We provide a package that installs the makefiles in the system where make can find them. This requires the `pkg.mk` include.

9a     $\langle include\ files\ 7d \rangle + \equiv$  (6d)  $\triangleleft 7f$   
      `include ${MAKEFILES_INCLUDE}/pkg.mk`

We only need to set some variables that `pkg.mk` expects.

9b     $\langle package\ setup\ 9b \rangle \equiv$  (6b)  
      `PKG_PACKAGES?=          main`  
      `PKG_NAME-main=         makefiles`  
  
      `PKG_PREFIX=           /usr/local`  
      `PKG_INSTALL_DIR=       /include`  
  
      `PKG_INSTALL_FILES-main= ${MKFILES}`  
      `PKG_TARBALL_FILES-main= ${PKG_INSTALL_FILES-main} ${OTHERS} Makefile README.md`  
  
      `.PHONY: all`  
      `all: makefiles.tar.gz`

Then we want to remove the package when we clean.

9c     $\langle clean\ recipe\ 7b \rangle + \equiv$  (6c)  $\triangleleft 8c$   
      `${RM} makefiles.tar.gz`

## Part I

# General building blocks

## Chapter 2

# portability.mk

### 2.1 Introduction

The purpose of this include file is to improve portability of the include files. The `make(1)` utility itself already provides certain portability between platforms, here we want to extend this portability. I.e. we provide variables which substitute to system-specific commands which corresponds to the expected action. For instance, MacOS uses an ancient version of `unzip`, a version which does not support the option `-DD` which is desirable. So, on MacOS the variable `UNZIP` will substitute to `unzip` which on other systems it will substitute to `unzip -DD`. Another examples is BSD-systems, which does not use the GNU versions of `sed` and `grep` (and `make`). On these systems `SED` will substitute to `gsed`, which is the GNU version of the command. Probably the reader can skip this chapter on a first reading.

The include file is structures similarly to a header file in C. We use the same technique to prevent multiple inclusions. The outline is as follows.

```
11a  <portability.mk 11a>≡
      ifndef PORTABILITY_MK
      PORTABILITY_MK=true

      <system-specific configuration 11b>

      <standard unix commands 12a>
      <networking commands 13d>
      <compressed files and archives 14b>

      endif
```

Since this file provides system-dependent configuration, we allow the user to provide a system-wide configuration file.

```
11b  <system-specific configuration 11b>≡ (11a)
      PORTABILITY_CONF?=  ${HOME}/.mk.conf /etc/mk.conf
```

```
-include ${PORTABILITY_CONF}
```

The file in `/etc/mk.conf` is commonly available in BSDs. However, since these files might not exist, `make(1)` should not yield a fatal error if the include directive fails.

## 2.2 Standard Unix commands

In this section we provide default commands with options for the standard Unix command line. More specifically, we cover the following areas.

12a *⟨standard unix commands 12a⟩*≡ (11a)  
*⟨file system commands 12b⟩*  
*⟨printing file contents 12d⟩*  
*⟨opening files depending on file type 12c⟩*  
*⟨filtering and transforming file contents 13a⟩*  
*⟨statistics on file contents 13c⟩*

### 2.2.1 File system commands

We commonly use the commands to interact with the file system. The following basic commands cover most uses.

12b *⟨file system commands 12b⟩*≡ (12a)  
 MV?= mv  
 CP?= cp -R  
 LN?= ln -sf  
 MKDIR?= mkdir -p  
 MKTMPDIR?=mktemp -d  
 CHOWN?= chown -R  
 CHMOD?= chmod -R

The `make(1)` utility already sets `RM = rm -f` by default [GNU16, Sect. 10.3], so we need not repeat it.

### 2.2.2 Viewing file contents

Quite commonly we want to open files with the user’s desired application, e.g., to open PDFs in the user’s PDF reader. For this we use the `xdg-open(1)` utility.

12c *⟨opening files depending on file type 12c⟩*≡ (12a)  
 OPEN?= xdg-open

However, for text files, we prefer to just print the contents to standard output.

12d *⟨printing file contents 12d⟩*≡ (12a)  
 CAT?= cat

### 2.2.3 Filtering and transformations

Two of the most frequently used utilities are `sed(1)` and `grep(1)`. The version of these that we want to use is the GNU version. On Linux systems, this is the default. On BSDs, however, they are available prefixed with the letter ‘g’. The same goes to the `make(1)` utility, which means that we can use that fact to check for this.

13a     $\langle \textit{filtering and transforming file contents 13a} \rangle \equiv$  (12a) 13b  $\triangleright$

```
ifeq (${MAKE},gmake)
SED?=      gsed
SEDEX?=    gsed -E
else
SED?=      sed
SEDEX?=    sed -E
endif
```

Similarly, we let

13b     $\langle \textit{filtering and transforming file contents 13a} \rangle + \equiv$  (12a)  $\triangleleft$  13a

```
ifeq (${MAKE},gmake)
GREP=      ggrep
GREPex=    ggrep -E
else
GREP=      grep
GREPex=    grep -E
endif
```

### 2.2.4 Statistics

We also need to count words in a few places. We use `wc(1)` for this.

13c     $\langle \textit{statistics on file contents 13c} \rangle \equiv$  (12a)

```
WC?=      wc
WCw?=     wc -w
```

## 2.3 Networking commands

We also need some network related commands.

13d     $\langle \textit{networking commands 13d} \rangle \equiv$  (11a)

```
 $\langle \textit{fetching files 13e} \rangle$ 
 $\langle \textit{remote execution 14a} \rangle$ 
```

We have some common commands for fetching and copying files between remote hosts.

13e     $\langle \textit{fetching files 13e} \rangle \equiv$  (13d)

```
CURL?=    curl
SFTP?=    sftp
SCP?=     scp -r
```

We also need commands for remote execution.

14a     $\langle \text{remote execution 14a} \rangle \equiv$  (13d)  
       SSH?=        ssh

## 2.4 Compressed files and archives

We want to provide functionality to make it easy to uncompress files or extract files from archives of different kinds. We will construct two functionalities.

14b     $\langle \text{compressed files and archives 14b} \rangle \equiv$  (11a)  
        $\langle \text{variables for compression programs 15b} \rangle$   
        $\langle \text{variables for archive programs 15e} \rangle$   
        $\langle \text{general pattern rule for archiving 15d} \rangle$   
        $\langle \text{function to generate extraction targets 16c} \rangle$

Both will use the type of construction outlined in [GNU16, Sect. 10.2]: the variable `EXTRACT.suf` (`UNCOMPRESS.suf`) will contain the command to extract a file from an archive (decompress a file) with suffix `.suf`; the variable `ARCHIVE.suf` (`COMPRESS.suf`) will contain the command to update an archive of suffix `.suf` with a file (compress a file).

### 2.4.1 Compressing and uncompressing files

A compressed file is a file whose data is compressed — this is not necessarily an archive. A compressed file can be uncompressed, i.e., the compression is removed. Compressed files usually get the added suffix of the compression algorithm, e.g., a `.tar` file usually get the suffix `.tar.gz` when it is also compressed using `gzip(1)`. Another common file to compress is PostScript, i.e., turning `.ps` to `.ps.gz`. We want to form pattern rules for the compression and uncompression operations.

There are, of course, a myriad different compression formats. We will let the variable `COMPRESS_SUFFIXES` and `UNCOMPRESS_SUFFIXES` contain space-separated lists of suffixes supported for the two operations.

To compress a file, we simply passes its contents through a compression program, e.g., `gzip(1)` (gets the `.gz` suffix). We can use the following general pattern rule for compression and then use `COMPRESS_SUFFIXES` to automatically generate all the pattern rules<sup>1</sup>.

14c     $\langle \text{general pattern rule for compression 14c} \rangle \equiv$   
       define compress  
       %(1): %  
       \${COMPRESS%(1)}

---

<sup>1</sup>Note that the pattern rules in the code blocks  $\langle \text{general pattern rule for compression 14c} \rangle$  and  $\langle \text{general pattern rule for uncompression 15a} \rangle$  are not included in  $\langle \text{compressed files and archives 14b} \rangle$  above, and thus not enabled by default. This is due to causing circular dependencies.

```

    endif
    $(foreach suf,${COMPRESS_SUFFIXES},$(eval $(call compress,${suf})))

```

In a similar fashion, we can use the following general pattern rule for uncompression.

```

15a  <general pattern rule for uncompression 15a>≡
      define uncompress
      %: %$(1)
      $$${UNCOMPRESS}$(1)}
      endif
      $(foreach suf,${UNCOMPRESS_SUFFIXES},$(eval $(call uncompress,${suf})))

```

We note that due to the `call` and `eval` above, we must escape the target and prerequisite variables, `$$@` and `$$<`, respectively.

Now, let us write what we need to automatically handle the `gzip(1)` format. To uncompress a gzipped file we can use `gunzip(1)`.

```

15b  <variables for compression programs 15b>≡                                     (14b) 15c>
      UNCOMPRESS_SUFFIXES+= .gz .z
      GUNZIP?=               gzip
      UNCOMPRESS.gz?=       ${GUNZIP} $<
      UNCOMPRESS.z?=       ${UNCOMPRESS.gz}

```

To compress a file using `gzip(1)` we can use the following.

```

15c  <variables for compression programs 15b>+≡                               (14b) <15b
      COMPRESS_SUFFIXES+= .gz
      GZIP?=               gzip
      COMPRESS.gz?=       ${GZIP} $<

```

## 2.4.2 Packing and extracting from archives

We can (ab)use the archive syntax [GNU16, Chap. 11] of `make(1)` to create a pattern rule for creating archives. This rule will work for all archive types that support adding files to an existing archive. However, `make(1)` cannot check the modification times of these archive members, so they will be updated every time instead of only when necessary.

The pattern rule matches all archive member targets. Then it determines which variable to use as recipe by looking at the suffix of the archive.

```

15d  <general pattern rule for archiving 15d>≡                               (14b)
      (%):
      ${ARCHIVE}$(suffix $@)}

```

Unlike for the compression targets above (Section 2.4.1), we do not need to escape `$@` and `$<` — since we have only one (lazy) evaluation.

We do not want to break the native archive functionality of `make(1)`, so we provide the following to retain that.

```

15e  <variables for archive programs 15e>≡                                   (14b) 16a>
      ARCHIVE.a?=   ar r $@ $%

```



Now to a more interesting format, let us create tarballs using this syntax. We provide settings for both `tar(1)` and `pax(1)` using the tar format. We are interested in the `pax(1)` command because it has an interface for regular expressions, i.e., for filtering and transforming file names. The BSD `tar(1)` has this too, but the GNU `tar(1)` does not. We can use the `-u` option to both `tar(1)` and `pax(1)` to update an existing archive with a file.

```
16a  <variables for archive programs 15e>+≡ (14b) <15e 16b>
      TAR?=          tar -u
      PAX?=          pax -wzLx ustar
      ARCHIVE.tar?=  ${TAR} -f $@ $%
```

We can also create `zip(1)` archives.

```
16b  <variables for archive programs 15e>+≡ (14b) <16a 16d>
      ZIP?=          zip
      ARCHIVE.zip?=  ${ZIP} -u $@ $%
```

Unfortunately, we cannot create any pattern rules for file extraction from archives. However, we can provide a function which create such targets automatically.

```
16c  <function to generate extraction targets 16c>≡ (14b)
      define extract
      $(1): $(2)
          $$${EXTRACT$(suffix $(2))}
      endef
```

Now we only need to provide the `EXTRACT.XXX` for every type of archive we might want to use. Then we can use the function `extract` in our makefiles. Note that we are now in the same situation as for the compression targets (Section 2.4.1), so we must escape the variables.

We start with tarballs. For extraction, we do not want to restore the modification times from inside the archive. If we restore the modification times, then the archive will always be newer than the files extracted from it and thus `make(1)` will re-extract the file every time. To prevent this we add the `-m` option to `tar(1)`.

```
16d  <variables for archive programs 15e>+≡ (14b) <16b 16e>
      UNTAR?=        tar -xm
      UNPAX?=        pax -rzp m
      EXTRACT.tar?=  ${UNTAR} -f $< $@
```

It will be similar for zip archives. The option to prevent resetting the modification time for `unzip(1)` is `-DD`. Unfortunately, MacOS ships with an ancient version of `unzip(1)`, one which does not support the desired `-DD` option. Hence we check if the system is Darwin, if so, we skip the `-DD` option.

```
16e  <variables for archive programs 15e>+≡ (14b) <16d
      ifeq ($(shell uname),Darwin)
      UNZIP?=        unzip
      else
```

```
UNZIP?=          unzip -DD
endif
EXTRACT.zip?= ${UNZIP} $< $@
```

## Chapter 3

# subdir.mk

### 3.1 Introduction and usage

Sometimes we want to recursively descend into subdirectories making a specific target in each subdirectory. The subdirectories must be listed in the variable `SUBDIR`, which holds a space-separated list of directory names. Then each subdirectory may in turn hold a new set of subdirectories to descend into. We note that the subdirectories will be built in depth-first search order (unless we use parallel execution).

By default, we will recurse into the subdirectories in `SUBDIR` for any goals passed on the command line. This behaviour can be overridden by changing the value of `SUBDIR_GOALS` from its default:

```
18a  <variables 18a>≡ (18b) 19b>
      SUBDIR_GOALS?=${MAKECMDGOALS}
```

Like this we can ensure that we only recurse into the subdirectories only for a subset of desired targets.

Any variables that should be passed down to sub-makes should be exported using the `export` directive of make.

### 3.2 Implementation

The structure of the file is that of most include files. We want to ensure that it is not included more than once. Furthermore, we do not want to do anything unless the `SUBDIR` variable, containing the space-separated list of subdirectories, exists.

```
18b  <subdir.mk 18b>≡
      ifndef SUBDIR_MK
      SUBDIR_MK=true

      <variables 18a>
```

```

ifdef SUBDIR
  <let the recipe for each subdirectory recurse into it 19a>
endif

ifneq (${SUBDIR_GOALS},)
  <add subdirectories as prerequisites for the goals 19c>
endif

endif

```

The thing we want to do is to build the given goals (`SUBDIR_GOALS`), i.e., the targets specified on the command-line by default, in all subdirectories listed in `SUBDIR`. For each directory, we specify a recipe which runs `make` in the subdirectory with the appropriate goals.

```

19a  <let the recipe for each subdirectory recurse into it 19a>≡ (18b)
      ${SUBDIR}::
      ${MAKE} -C $@ ${actual_goals}

```

Those `actual_goals` is the intersection of the goals specified on the command-line and those specified in `SUBDIR_GOALS`.

```

19b  <variables 18a>+≡ (18b) <18a
      actual_goals=$(sort $(filter ${SUBDIR_GOALS},${MAKECMDGOALS}))

```

To ensure these recipes are run we need to ensure that they are prerequisites to those goals.

```

19c  <add subdirectories as prerequisites for the goals 19c>≡ (18b) 19d>
      ifneq (${actual_goals},)
      .PHONY: ${actual_goals}
      ${actual_goals}: ${SUBDIR}
      endif

```

Finally, we must consider the case of the default goal, i.e., no goal is specified on the command line. In this case we simply check if the default goal (`.DEFAULT_GOAL`) is among the desired goals (`SUBDIR_GOALS`).

```

19d  <add subdirectories as prerequisites for the goals 19c>+≡ (18b) <19c
      ifneq ($(filter ${.DEFAULT_GOAL},${SUBDIR_GOALS}),)
      .PHONY: ${.DEFAULT_GOAL}
      ${.DEFAULT_GOAL}: ${SUBDIR}
      endif

```

We note that if we specify no goals on the command line, then `actual_goals` will be the empty string. This means that we execute the default goal in each subdirectory recursively: e.g., if the current default goal is `all`, but its `clean` in a subdirectory, then we will execute `all` here but `clean` there — not `all` in both places.

## Part II

# Packaging and publishing

# Chapter 4

## pkg.mk

### 4.1 Introduction and usage

The idea of this include file is to provide an easy way to package files together for publication. It can be for packaging the source code of a document or package a script with automatic installation instructions.

The first thing we need for a package is a name. This is controlled by the `PKG_NAME` variable.

21a    `<variables 21a>≡` (23a) 21b>  
          `PKG_NAME?=`            `${PACKAGE}`

Its default value is set for backwards compatibility, so that makefiles using the old variable names will still work. The package name will, by default, determine the name of the tarball that is generated.

21b    `<variables 21a>+≡` (23a) <21a 21c>  
          `PKG_TARBALL?=`        `${PKG_NAME}.tar.gz`

The next thing we need is to control which files are included. There are two types of files: files that should be installed and files that should not.

21c    `<variables 21a>+≡` (23a) <21b 21d>  
          `PKG_INSTALL_FILES?=` `${INSTALL_FILES}`  
          `PKG_TARBALL_FILES?=` `${PACKAGE_FILES} ${PKG_INSTALL_FILES}`

The tarball files will only be included in the tarball, but the install files will be installed if the `install` target is made. For example, a `Makefile` should be included (since it contains the installation target), but it should not be installed. When the file lists include directories it might be interesting to ignore certain files, e.g., version management. This can be done with the following.

21d    `<variables 21a>+≡` (23a) <21c 22a>  
          `IGNORE_FILES?=`        `\(\\.svn\\|\\.git\\|CVS\\)`  
          `PKG_IGNORE?=`          `${IGNORE_FILES}`

The installation path is controlled by the following variables.

```
22a  <variables 21a>+≡ (23a) <21d 22b>
      PKG_PREFIX?=      ${PREFIX}
      PKG_INSTALL_DIR?=  ${INSTALLDIR}
```

Sometimes different parts of a package must be installed to different places, e.g., a script to `/usr/local/bin` and a manual page to `/usr/local/share/man`. For this purpose, a package can be divided into several sub-packages. By default we have one package called `main`.

```
22b  <variables 21a>+≡ (23a) <22a 22c>
      PKG_PACKAGES?=      main
```

For each such package we can set a specialized version of the variables we discussed above. By default, they will inherit the global values set above.

```
22c  <variables 21a>+≡ (23a) <22b 22d>
      define variables
      PKG_NAME-$(1)?=      ${PKG_NAME}
      PKG_INSTALL_FILES-$(1)?=  ${PKG_INSTALL_FILES}
      PKG_PREFIX-$(1)?=      ${PKG_PREFIX}
      PKG_INSTALL_DIR-$(1)?=    ${PKG_INSTALL_DIR}

      PKG_TARBALL-$(1)?=      ${PKG_NAME-$(1)}.tar.gz
      PKG_TARBALL_FILES-$(1)?=  ${PKG_TARBALL_FILES}
      PKG_IGNORE-$(1)?=       ${PKG_IGNORE}
      endef
```

Then we use this as a function to set the variables for each sub-package.

```
22d  <variables 21a>+≡ (23a) <22c 22e>
      $(foreach pkg,${PKG_PACKAGES},$(eval $(call variables,${pkg})))
```

### 4.1.1 Portability

For portability, this include file requires the following programs to be available.

```
22e  <variables 21a>+≡ (23a) <22d
      ifneq (${MAKE},gmake)
      INSTALL?=      ${SUDO} install -Dp
      else
      INSTALL?=      ${SUDO} install -CSp
      endif
```

## 4.2 Implementation

This is an include file, so we will use a C-style header construction to prevent it from being included more than once. Then the overview of the structure is

as follows.

```
23a  <pkg.mk 23a>≡
      ifndef PACKAGE_MK
      PACKAGE_MK=true

      INCLUDE_MAKEFILES?=
      include ${INCLUDE_MAKEFILES}/portability.mk

      <variables 21a>
      <an all-like target 23b>
      <targets for packaging 23c>
      <targets for cleaning 24a>
      <targets for installation 24d>

      endif
```

We want to have an all-like target, we call it **package**. The **package** target should, of course, have all tarballs as prerequisites. The reason for not using **all** is that we leave the **all** target for the user, with its prerequisites defined in the main makefile.

```
23b  <an all-like target 23b>≡ (23a)
      .PHONY: package
      package: $(foreach pkg,${PKG_PACKAGES},${PKG_TARBALL-${pkg}})
```

### 4.2.1 Packaging

The packaging step shall take the files specified and create a tarball containing them. What we will do is to create a target for the tarball. We will do this by using the archive functionality of `make(1)` and the compression functionality we added in Section 2.4.2.

```
23c  <targets for packaging 23c>≡ (23a)
      define tarball
      $(foreach f,${PKG_TARBALL_FILES-${1}},\
        $(eval ${PKG_TARBALL-${1}}(${f}): ${f}))
      ${PKG_TARBALL-${1}}: ${PKG_TARBALL-${1}}(${PKG_TARBALL_FILES-${1}})
      endef
      $(foreach pkg,${PKG_PACKAGES},${eval $(call tarball,${pkg}))})
```

### 4.2.2 Cleaning

The kind of cleaning we are interested in is to remove the tarballs that we generate. The other files, install and tarball files, should be cleaned using other cleaning targets — if they need cleaning at all.



The technique we use is to provide a `clean-package` target which we add as a prerequisite to the general target `clean`. This way the user can have a recipe for `clean` in the main makefile without us interfering.

24a     $\langle$ *targets for cleaning* 24a $\rangle \equiv$  (23a) 24b $\triangleright$   
       `.PHONY: clean clean-package`  
       `clean: clean-package`

We now create a cleaning target for each sub-package and add those as prerequisites for the `clean-package` target. The recipe is to remove the tarball of that particular sub-package.

24b     $\langle$ *targets for cleaning* 24a $\rangle + \equiv$  (23a)  $\triangleleft$ 24a  
       `define clean-package`  
       `.PHONY: clean-package-$(1)`  
       `clean-package: clean-package-$(1)`  
       `clean-package-$(1):`  
       `${RM} ${PKG_TARBALL-$(1)}`  
       `endef`  
       `$(foreach pkg,${PKG_PACKAGES},${(eval $(call clean-package,${pkg}))})`

### 4.2.3 Installation

The `install` target will install the files that are configured to be installed where they are configured to be installed. The installation process proceeds in the following steps.

24c     $\langle$ *installation process* 24c $\rangle \equiv$  (24d)  
       `.PHONY: pre-install do-install post-install`  
       `post-install: do-install`  
       `do-install: pre-install`  
       `pre-install: ${PKG_INSTALL_FILES}`

This will ensure that the targets' recipes are run in the desired order (since the prerequisites' recipes are run first, if needed). This means that the files to be installed are made before `pre-install` is run. To start the process with the `install` target, we add the following.

24d     $\langle$ *targets for installation* 24d $\rangle \equiv$  (23a) 24e $\triangleright$   
       `.PHONY: install`  
       `install: post-install`  
        $\langle$ *installation process* 24c $\rangle$

Now we need to provide package dependent versions of these targets. We achieve this by simply adding the package dependent versions as prerequisites for the general targets, and in the same order as we did for the general targets.

24e     $\langle$ *targets for installation* 24d $\rangle + \equiv$  (23a)  $\triangleleft$ 24d  
       `define post-install`  
       `.PHONY: post-install-$(1)`  
       `post-install: post-install-$(1)`

```

post-install-$(1): do-install-$(1)
endif
$(foreach pkg,${PKG_PACKAGES},${eval $(call post-install,${pkg})))

define do-install
.PHONY: do-install-$(1)
do-install: do-install-$(1)
do-install-$(1): pre-install-$(1)
    <default do-install recipe 25>
endif
$(foreach pkg,${PKG_PACKAGES},${eval $(call do-install,${pkg})))

define pre-install
.PHONY: pre-install-$(1)
pre-install: pre-install-$(1)
pre-install-$(1): ${PKG_INSTALL_FILES-$(1)}
endif
$(foreach pkg,${PKG_PACKAGES},${eval $(call pre-install,${pkg})))

```

Finally, we need a default recipe for the `do-install` target, otherwise the user would have to write one every time — and that would counter the purpose of this include file. The procedure is straight-forward. We first create the target directory, with a possible prefix. Then, for every non-directory, we install using the `install` command.

```

25  <default do-install recipe 25>≡ (24e)
    for f in ${PKG_INSTALL_FILES-$(1)}; do \
        [ -d "$$$$f" ] || ${INSTALL} -t ${PKG_PREFIX-$(1)}${PKG_INSTALL_DIR-$(1)}/ "$$$$f"; \
    done

```

# Chapter 5

## pub.mk

### 5.1 Introduction and usage

Sometimes we wish to easily publish a release of the material we work with. Here we provide the functionality of publishing files, we have two ways:

1. the **upload** target which uploads files to a server, and
2. the **gh-release** target which creates a release on the GitHub repo.

To use the **publish** target, we must add either **publish: upload** or **publish: gh-release** to our Makefile.

#### 5.1.1 Specifying files

The idea is to publish files, and this is common between all publication methods. This is controlled with the **PUB\_FILES** variable, which is set to a space separated list of file names.

26a `<variables 26a>≡` (29c) 26b>  
`PUB_FILES?=`

The **upload** target will take the files in **PUB\_FILES** and upload them to the target server (more below). The **gh-release** on the other hand will use the **PUB\_FILES** as attachment to the release.

For convenience, we can also control files to ignore.

26b `<variables 26a>+≡` (29c) <26a 27a>  
`IGNORE_FILES?=` `\\(.svn\\|.git\\|CVS\\)`  
`PUB_IGNORE?=` `${IGNORE_FILES}`

However, this only applies to the **upload** target.

#### 5.1.2 Automatically tag on publication

Since the published files usually are stripped of their versioning information, it can be a good idea to keep track of the corresponding version in the version

management system. One way is to create a tag every time a publication is made.

To enable this feature we set the variable `PUB_AUTOTAG` to true. By default we let it be false, i.e., this feature is disabled.

27a `<variables 26a>+≡` (29c) <26b 27b>  
`PUB_AUTOTAG?= false`

Note that for `gh-release` this doesn't matter, a tag will be created on the GitHub side if it doesn't exist already.

The first thing we need is to know which version control system (VCS) is used. We control this with `PUB_VCS`.

27b `<variables 26a>+≡` (29c) <27a 27c>  
`PUB_VCS?= git`

The only thing needed more than this is any options that the user want to use.

27c `<variables 26a>+≡` (29c) <27b 27d>  
`PUB_TAG_OPTS?=`

The tag name is controlled with the following variable. The default value is today's date and the current time.

27d `<variables 26a>+≡` (29c) <27c 27e>  
`PUB_TAG_NAME?= $(shell date +%Y%m%d-%H%M)`

The tagging will be wrong if we have forgotten to commit the files we were working on. For this reason we also provide a similar feature which automatically makes a commit. This feature is also disabled by default.

27e `<variables 26a>+≡` (29c) <27d 27f>  
`PUB_AUTOCOMMIT?= false`

The command and options are similarly set with the following.

27f `<variables 26a>+≡` (29c) <27e 27g>  
`PUB_COMMIT_OPTS?= -av`

## 5.2 Configuration for publishing files on a server, upload

Publication means that we upload the files somewhere. This is controlled by the following variable.

27g `<variables 26a>+≡` (29c) <27f 27h>  
`PUB_SERVER?= localhost`

We are also interested in where on the server the files are written.

27h `<variables 26a>+≡` (29c) <27g 28a>  
`PUB_DIR?= ${PUBDIR}/${CATEGORY}`

Once written to the location, we must consider the owner, group and access rights.

28a     $\langle \text{variables } 26a \rangle + \equiv$  (29c)  $\langle 27h \ 28b \rangle$   
       PUB\_USER?=            \${USER}  
       PUB\_GROUP?=         \${GROUP}  
       PUB\_CHMOD?=        o+r

### 5.2.1 Publication methods

There are currently three methods for publication: 1. **ssh**, 2. **git**, and 3. **at**. The default method is **ssh**.

28b     $\langle \text{variables } 26a \rangle + \equiv$  (29c)  $\langle 28a \ 28c \rangle$   
       PUB\_METHOD?=        ssh

The remaining parts of the configuration depends on which publication method is used.

**ssh** The **ssh** method will use the Secure SHell (SSH) protocol to transfer the files. It will compress the files, pipe the output to the **ssh** process which runs the decompression on the server — in the specified directory. After successful transfer it will try to change the access rights to what is given by the settings above.

**at** The **at** method works similarly to **ssh**, the difference is that it postpones the publication until a certain time. The time is given by the **PKG\_AT** variable, or **at** as a shortcut for the command-line (**make at=tomorrow**).

28c     $\langle \text{variables } 26a \rangle + \equiv$  (29c)  $\langle 28b \ 28d \rangle$   
       at?=                    tomorrow  
       PKG\_AT?=              \${at}

The way this works is that instead of writing the files to **PUB\_DIR** on the server, we write the files to **PUB\_TMP** and then add an **at** job that will move the files from the temporary to the final directory.

28d     $\langle \text{variables } 26a \rangle + \equiv$  (29c)  $\langle 28c \ 28e \rangle$   
       PUB\_TMPDIR?=         /var/tmp

**git** The **git** method uses Git's archive functionality. This means that Git will export an archive made from a branch in the repository, which branch is used is controlled by the following variable.

28e     $\langle \text{variables } 26a \rangle + \equiv$  (29c)  $\langle 28d \ 29a \rangle$   
       PUB\_BRANCH?=         master

## 5.2.2 Publishing to multiple sites

We might also be interested in publishing files to several places, e.g., to a set of mirrors. The variable `PUB_SITES` contains a list of sites.

29a `<variables 26a>+≡` (29c) `<28e 29b>`  
`PUB_SITES?= main`

We supply one by default, this allows us to simply use the general variables above. This way, site-specific overrides can be specified by appending the variable with the site name, e.g., `-main`. All other values are copied from the defaults, i.e., the general variables.

29b `<variables 26a>+≡` (29c) `<29a 32f>`  
`define variables`  
`PUB_METHOD-$(1)?= ${PUB_METHOD}`  
  
`PUB_SERVER-$(1)?= ${PUB_SERVER}`  
`PUB_DIR-$(1)?= ${PUB_DIR}`  
`PUB_FILES-$(1)?= ${PUB_FILES}`  
`PUB_IGNORE-$(1)?= ${PUB_IGNORE}`  
  
`PUB_USER-$(1)?= ${PUB_USER}`  
`PUB_GROUP-$(1)?= ${PUB_GROUP}`  
`PUB_CHMOD-$(1)?= ${PUB_CHMOD}`  
  
`PUB_AT-$(1)?= ${PUB_AT}`  
`PUB_TMPDIR-$(1)?= ${PUB_TMPDIR}`  
  
`PUB_BRANCH-$(1)?= ${PUB_BRANCH}`  
`endif`  
  
`$(foreach site,${PUB_SITES},$(eval $(call variables,${site})))`

**Example 1.** To publish the same material to three different mirrors, we can do the following.

```
1 PUB_SITES= main mirror1 mirror2
2 PUB_SERVER = foo.bar
3 PUB_SERVER-mirror1 = foo.bar.mirror1
4 PUB_SERVER-mirror2 = foo.bar.mirror2
```

## 5.3 Implementation

This is an include file, so we will first use the C-style technique to prevent inclusion more than once. Thus the structure is as follows.

29c `<pub.mk 29c>≡`  
`ifndef PUB_MK`

```

PUB_MK=true

INCLUDE_MAKEFILES?=.
include ${INCLUDE_MAKEFILES}/portability.mk

<variables 26a>

.PHONY: publish upload gh-release

ifeq (${PUB_AUTOTAG},true)
upload: autotag
gh-release: autotag
else ifeq (${PUB_AUTOCOMMIT},true)
upload: autocommit
gh-release: autocommit
endif

<upload target 30a>
<gh-release target 34c>
<autotag and autocommit targets 35a>

endif

```

We will now cover the different parts below. The *<variables 26a>* block has been covered in the usage section, but the remaining are discussed below.

### 5.3.1 The upload publication mechanism, upload

The upload target consists of two parts.

30a    *<upload target 30a>* ≡ (29c)  
       *<target for uploading 30b>*  
       *<publication methods 31b>*

We have a general publication mechanism that drives the publication process and uses the methods described below. We have a general target **upload** to be invoked by the user. Then we have a specific **upload-site** target for each site, which does the actual publication. We add all those as prerequisites to the main target.

30b    *<target for uploading 30b>* ≡ (30a) 30c>  
       .PHONY: upload  
       upload: \$(foreach site,\${PUB\_SITES},upload-\${site})

Depending on the settings for automatic commits and tags, we also add targets for those functionalities as prerequisites.

30c    *<target for uploading 30b>* + ≡ (30a) <30b 31a>  
       ifeq (\${PUB\_AUTOTAG},true)  
       upload: autotag

```

else ifeq (${PUB_AUTOCOMMIT},true)
upload: autocommit
endif

```

Next up is the actual site-specific targets. The prerequisites are the files that should be uploaded. Then the recipe is simply a call to the relevant publication method.

```

31a  <target for uploading 30b>+≡ (30a) <30c
      define upload_target
      .PHONY: upload-$(1)
upload-$(1): $(foreach file,${PUB_FILES-$(1)},${file})
      $$ (call upload-${PUB_METHOD-$(1)},$(1))
      endef

      $(foreach site,${PUB_SITES},$(eval $(call upload_target,${site})))

```

### 5.3.2 Publication methods

We will now cover the different publication methods. The outline is as follows.

```

31b  <publication methods 31b>≡ (30a)
      <helper functions 31c>
      <ssh method 32a>
      <at method 33b>
      <git method 34b>

```

We will first discuss two helper functions, **chown** and **chmod**. Then we will process with the different methods discussed in the introduction.

Both **chown** and **chmod** takes one argument, the name of the site. Then each function can use the site name to find the relevant configuration. The **chown** function simply runs **chown(1)** on the **PUB\_DIR** directory on the server.

```

31c  <helper functions 31c>≡ (31b) 31d>
      define chown
      $(if ${PUB_GROUP-$(1)},\
        $(if ${PUB_SERVER-$(1)},${SSH} ${PUB_SERVER-$(1)}\
          ${CHOWN} ${PUB_USER-$(1)}:$(strip ${PUB_GROUP-$(1)})\
          $(foreach f,${PUB_FILES-$(1)},${PUB_DIR-$(1)}/${f} );\
        ,)
      endef

```

Conversely, the **chmod** function does the same but with the **chmod(1)** command. Note, however, that we do not run these commands if **PUB\_GROUP** or **PUB\_CHMOD**, respectively, are empty.

```

31d  <helper functions 31c>+≡ (31b) <31c
      define chmod
      $(if ${PUB_CHMOD-$(1)},\
        $(if ${PUB_SERVER-$(1)},${SSH} ${PUB_SERVER-$(1)}\

```



```

    ${CHMOD} ${PUB_CHMOD}-${(1)}\
    $(foreach f,${PUB_FILES}-${(1)},${PUB_DIR}-${(1)}/${f });\
    ,)
endif

```

**ssh** Now to the first publication method, the one using copying over SSH. We define the method as a make function which takes one argument, the name of the site.

```

32a  <ssh method 32a>≡ (31b)
      define upload-ssh
      <create directory on server 32b>; \
      <pack the files and pipe them to the server 32c>; \
      $(call chown,$(1)) \
      $(call chmod,$(1))
      endif

```

To create the directory on the server is straight-forward, we simply run the command over SSH.

```

32b  <create directory on server 32b>≡ (32a 33b)
      $(if ${PUB_SERVER}-${(1)},${SSH} ${PUB_SERVER}-${(1)}) ${MKDIR} ${PUB_DIR}-${(1)}

```

Next is the packing of the files.

```

32c  <pack the files and pipe them to the server 32c>≡ (32a)
      <generate file list 32d> | \
      <pack the files 32e> | \
      <extract the files on the server 33a>

```

Before we do anything with the files, we must ensure that the list of files is not empty — if it was empty, that would break all of the following commands. If not, we will use `find(1)` to generate a list of files to include. We do this in case there is a directory in the list `PUB_FILES`. If there is a directory in there, we cannot filter it using `PKG_IGNORE`, so we must generate a list of the entire hierarchy included.

```

32d  <generate file list 32d>≡ (32c 33b)
      [ -n "${PUB_FILES}-${(1)}" ] && find ${PUB_FILES}-${(1)} -type f -or -type l

```

Once we have the list we can use `pax(1)` to put them into an archive, an archive which is written to standard out.

```

32e  <pack the files 32e>≡ (32c 33b)
      xargs ${PAX} \
      $(foreach regex,${PUB_REGEX}-${(1)},-s ${regex}) \
      -s "|^.*/${strip ${PUB_IGNORE}-${(1)}}/.*$$|p"

```

We also filter the file list through a series of regular expressions. The user may add regular expressions as a space-separated list in the following variable.

```

32f  <variables 26a>+≡ (29c) <29b 33d>
      PUB_REGEX?= "|^(.*)$$$$|\1|p"
      $(foreach site,${PKG_SITES},$(eval PUB_REGEX-${site}?=${PUB_REGEX}))

```

Finally, we extract the files on the server by running the corresponding `pax(1)` instance over SSH.

```
33a <extract the files on the server 33a>≡ (32c)
    $(if ${PUB_SERVER}-${(1)},${SSH} ${PUB_SERVER}-${(1)}) ${UNPAX} \
    -s "\"|^|${strip ${PUB_DIR}-${(1)}}/|p\""
```

**at** The next method is very similar to the first. The difference here is a middle step where we copy the files to a temporary place on the server and an additional final step where we upload them in the destination at some predefined time.

```
33b <at method 33b>≡ (31b)
    define upload-at
    <create directory on server 32b>; \
    <create temporary directory 33c>; \
    <generate file list 32d> | \
    <pack the files 32e> | \
    <extract the files in the temporary directory 33e>; \
    <add at-job on the server 33f>
    endef
```

We have already seen some of these code blocks above, we will now cover the new ones.

The first thing we want to do is to create a temporary directory on the server. We do this in the proper way.

```
33c <create temporary directory 33c>≡ (33b)
    TMPPUB=$((if ${PUB_SERVER}-${(1)},${SSH} ${PUB_SERVER}-${(1)}) \
    "export TMPDIR=${PUB_TMPDIR}-${(1)} && ${MKTMPDIR}-${(1)}")
```

We allow the user to override the `mktemp` command per server, since this command might differ on different servers.

```
33d <variables 26a>+≡ (29c) <32f 34a>
    $(foreach site,${PUB_SITES},$(eval MKTMPDIR=${site}?=${MKTMPDIR}))
```

Next we upload the files to the temporary directory on the server. The difference between this and previous upload is the extraction. We will now use a different regular expression, one which prepends the temporary directory to all files.

```
33e <extract the files in the temporary directory 33e>≡ (33b)
    $(if ${PUB_SERVER}-${(1)},${SSH} ${PUB_SERVER}-${(1)}) ${UNPAX} \
    -s "\"|^|${TMPPUB}/|p\""
```

Finally, we must add the `at(1)` job on the server. This is done by changing the directory to the temporary directory, then we echo the commands we want to execute later and pipe those to the `at(1)` command.

```
33f <add at-job on the server 33f>≡ (33b)
    $(if ${PUB_SERVER}-${(1)},${SSH} ${PUB_SERVER}-${(1)}) "cd ${TMPPUB} && (\
    echo 'mv ${PUB_FILES}-${(1)} ${PUB_DIR}-${(2)}';' \
```

```

$(if ${PUB_CHMOD}-${(1)},\
    echo '${CHMOD}-${(1)} ${PUB_CHMOD}-${(1)} ${PUB_DIR}-${(1)}; ',) \
$(if ${PUB_GROUP}-${(1)},\
    echo '${CHOWN}-${(1)} ${PUB_USER}-${(1)}:${(strip ${PUB_GROUP}-${(1)})} ${PUB_DIR}-${(1)}; ',) \
) | at ${PKG_AT}"

```

We note that we allow the user to specify different `CHOWN` and `CHMOD` variables for different servers, since these commands might differ per server.

```

34a  <variables 26a>+≡ (29c) <33d
      define chown_and_chmod
      CHOWN-${(1)}?=  ${CHOWN}
      CHMOD-${(1)}?=  ${CHMOD}
      endef
      $(foreach site,${PUB_SITES},$(eval $(call chown_and_chmod,${site})))

```

**git** The last method uses Git's functionality to pack the files. We simply use `git archive` and specify which branch to use. Then we pipe the archive to the server, unpack as before and finally run `chown` and `chmod`.

```

34b  <git method 34b>≡ (31b)
      define upload-git
      git archive ${PUB_BRANCH}-${(1)} ${PUB_FILES}-${(1)} \
      | $(if ${PUB_SERVER}-${(1)},${SSH} ${PUB_SERVER}-${(1)}) \
      ${UNPAX} -s ",^,${(strip ${PUB_DIR}-${(1)})},"; \
      $(call chown,${(1)}) \
      $(call chmod,${(1)})
      endef

```

### 5.3.3 GitHub releases, `gh-release`

Now let's turn our attention to the `gh-release` target. It's important that we push any changes, since the tag and release are created on the server.

```

34c  <gh-release target 34c>≡ (29c)
      .PHONY: gh-release
      gh-release: ${PUB_FILES}
      git push -all
      git push -tags
      gh release create -t ${PUB_TAG_NAME} ${PUB_TAG_NAME} ${PUB_FILES}

```

### 5.3.4 Automatically committing and tagging, `autotag` and `autocommit`

The last feature allows us to automatically commit and make a tag when we publish. We accomplish this by two targets that we have already seen above.

These targets use functions specific to the selected VCS.

35a  $\langle autotag \text{ and } autocommit \text{ targets } 35a \rangle \equiv$  (29c)  
 $\langle commit \text{ and } tag \text{ functions } 35b \rangle$

```
.PHONY: autocommit
autocommit:
    $(call autocommit- $\{PUB\_VCS\}$ )
```

```
.PHONY: autotag
autotag:
    $(call autotag- $\{PUB\_VCS\}$ )
```

Below we will cover the different VCSs.

For now there are two functions, one for committing and one for tagging. The commit functions are quite straight-forward for all three VCSs. The tagging is similarly straight-forward for two, but not the third.

35b  $\langle commit \text{ and } tag \text{ functions } 35b \rangle \equiv$  (35a)  
 $\langle autocommit \text{ for } git, \text{ svn and cvs } 35c \rangle$   
 $\langle autotag \text{ for } git \text{ and cvs } 35d \rangle$   
 $\langle autotag \text{ for svn } 35e \rangle$

The commit functions are as expected for all three VCSs.

35c  $\langle autocommit \text{ for } git, \text{ svn and cvs } 35c \rangle \equiv$  (35b)  
`autocommit-git = git diff -quiet || git commit  $\{PUB\_COMMIT\_OPTS\}$`   
`autocommit-svn = svn commit  $\{PUB\_COMMIT\_OPTS\}$`   
`autocommit-cvs = cvs commit  $\{PUB\_COMMIT\_OPTS\}$`

The tagging is similarly straight-forward for Git and Concurrent Versions System (CVS).

35d  $\langle autotag \text{ for } git \text{ and cvs } 35d \rangle \equiv$  (35b)  
`autotag-git = git tag  $\{PUB\_TAG\_OPTS\}$   $\{PUB\_TAG\_NAME\}$`   
`autotag-cvs = cvs tag  $\{PUB\_TAG\_OPTS\}$   $\{PUB\_TAG\_NAME\}$`

The tagging function for Subversion (SVN) is not as easy though. The outline is as follows.

35e  $\langle autotag \text{ for svn } 35e \rangle \equiv$  (35b)  
 $\langle helper \text{ functions for svn tagging } 36a \rangle$

```
define autotag-svn
 $\langle find \text{ the root of repo } 35f \rangle$ 
 $\langle go \text{ to root and create tag } 36b \rangle$ 
endef
```

To find the root of the repository, or more exactly where the directories `trunk` and `tags` are located, we must search through the parent directories. We start in the current working directory and add one level per iteration.

35f  $\langle find \text{ the root of repo } 35f \rangle \equiv$  (35e)  
`ROOT=.`

```

while ! [ -d ${ROOT}/trunk ]; do \
    $(call exit_if_fs_root,${ROOT})
    ROOT=${ROOT}/.. \
done \

```

We must check if we reach the root of the file system. We use the function `exit_if_fs_root` for this. This function exits with value 1 if the current directory examined is the root of the file system. If this happens, `make(1)` will abort the recipe and the code after will not be executed. The way we check for equality is to check that the device identifiers and the inode numbers are equal, we can do that using `stat(1)`.

36a  $\langle$ *helper functions for svn tagging* 36a $\rangle \equiv$  (35e)

```

define exit_if_fs_root
if [ $(stat -c %i $(1)) = $(stat -c %i /) \
    -a $(stat -c %d $(1)) = $(stat -c %d /) ]; then \
    exit 1; \
fi
endif

```

Finally, if the recipe is still executing, this means that we have found the root and we can copy the trunk to tags.

36b  $\langle$ *go to root and create tag* 36b $\rangle \equiv$  (35e)

```

cd ${ROOT} \
    && svn copy trunk tags/${PUB_TAG_NAME} \
    && svn commit ${PUB_COMMIT_OPTS};

```

## Chapter 6

# transform.mk

### 6.1 Introduction and usage

It is difficult to work openly with assessment material. We do not want to publish the solutions to the assignment so that the students can find them and pass the assessment without actually learning the material. On the other hand, we want to be able to publicly collaborate with other teachers, to improve the assignments and their solutions. This include file provides some tools to achieve this.

### 6.2 Implementation overview

The structure is similar to other include files. We want to prevent repeated inclusion, so we use a C-style technique to avoid that.

```
37 <transform.mk 37>≡
    ifndef TRANSFORM_MK
    TRANSFORM_MK=true

    INCLUDE_MAKEFILES?=.
    include ${INCLUDE_MAKEFILES}/portability.mk

    <variables 38a>
    <suffix rule for transformations 38b>
    <target generation for transformations 39a>
    <suffix rules for camera-ready source 40b>
    <suffix rules for encrypted files 41d>

    endif
```

We will now explore now these are implemented.

## 6.3 A transformation mechanism

We want to provide suffix rules for transforming files in different ways. We will transform any file with a suffix in `TRANSFORM_SRC` for which there is a corresponding target in `TRANSFORM_DST`.

38a  $\langle \text{variables 38a} \rangle \equiv$  (37) 38e $\triangleright$

```

TRANSFORM_SRC?=      .tex
TRANSFORM_DST?=      .transformed.tex

```

Then we can form the following suffix rule, which covers all combinations of sources and destinations.

38b  $\langle \text{suffix rule for transformations 38b} \rangle \equiv$  (37)

$\langle \text{transformations 38c} \rangle$

```

.SUFFIXES: ${TRANSFORM_SRC} ${TRANSFORM_DST}
$(foreach src,${TRANSFORM_SRC},$(foreach dst,${TRANSFORM_DST},${src}${dst})):
   $\langle \text{transformation recipe 38d} \rangle$ 

```

The  $\langle \text{transformations 38c} \rangle$  will be covered below, we start with how they are applied.

We will now describe a function which makes it easier to apply a list of transformations. The first argument is the input file, the second is a space separated list of transformations and the third is the output file.

38c  $\langle \text{transformations 38c} \rangle \equiv$  (38b) 39b $\triangleright$

```

define transform
cat $(1) $(foreach t,$(2),| $(call ${t})) > $(3)
endef

```

What we do here is to expand each transformation in the list to a pipe expression, so the result is a pipeline through which the file contents is piped. Thus every transformation must read from standard input and write to standard output.

Now back to our  $\langle \text{transformation recipe 38d} \rangle$ . This is a suffix rule, but we want the transformation to be target-dependent. To solve this, we will have a variable `TRANSFORM_LIST-target` containing the space separated list of transforms to apply to the target. We will also use `TRANSFORM_LIST.suf`, where `.suf` is the suffix of the target file. Thus we can just apply this list using the function above.

38d  $\langle \text{transformation recipe 38d} \rangle \equiv$  (37) 38b 39a $\triangleright$

```

$(call transform,\
  $$^,\
  ${TRANSFORM_LIST}$(suffix $$@) ${TRANSFORM_LIST-$$@},\
  $$@)

```

We will let `TRANSFORM_LIST` be the default list of transformations applied.

38e  $\langle \text{variables 38a} \rangle + \equiv$  (37) <38a 39e $\triangleright$

```

$(foreach suf,${TRANSFORM_DST},$(eval TRANSFORM_LIST${suf}?=${TRANSFORM_LIST}))

```

This will work well for a lot of cases, however, there are cases where suffix rules simply will not work. For these we must generate specific targets. Let `TRANSFORM_TARGETS` contain a space separated list of target files.

```
39a <target generation for transformations 39a>≡ (37)
    define target_recipe
    $(1):
        <transformation recipe 38d>
    endef
    $(foreach target,$(TRANSFORM_TARGETS),$(eval $(call target_recipe,$(target))))
```

Note that we use the same recipe as above.

### 6.3.1 Removing solutions

To remove solutions we will supply a filtering transformation. The filter uses `sed` to remove every solution environment from the content.

```
39b <transformations 38c>+≡ (38b) <38c 39c>
    NoSolutions?= ${SED} "/\\\\\\begin{solution}/,\\\\\\end{solution}/d"
```

### 6.3.2 Removing excessive build instructions

Sometimes we have extra build instructions in the internal repo, which are not necessary for the exported source code.

```
39c <transformations 38c>+≡ (38b) <39b 39d>
    ExportFilter?= ${SED} "/#export \\(false\\\\no\\\\)/,#export \\(true\\\\yes\\\\)/d"
    OldExportFilter?= ${SED} "/#export no/,#endexport/d"
```

### 6.3.3 Handouts and solutions

It is common that we want to produce handouts from slides and solutions for assignments or exams. We do not want to do this by hand, so we add two transformations that can be used to do this for us.

```
39d <transformations 38c>+≡ (38b) <39c
    PrintAnswers?= ${SED} "${MATCH_PRINTANSWERS}"
    Handout?= ${SED} "${MATCH_HANDOUT}"
```

We will need quite a few layers of escaping for these two regular expressions.

First we will handle the printing of solutions. We want to add the `\printanswers` command [Hir15] to the preamble. What we do is to match on the exam document class, then we insert the `\printanswers` command directly after it.

```
39e <variables 38a>+≡ (37) <38e 40a>
    exam_class= "(\\\\\\\\\\\\\\\\\\\\documentclass\\\\[?.*\\\\]?{.*exam.*})"
    with_print= "\\1\\\\\\\\\\\\\\\\\\\\printanswers"
    SED_PRINTANSWERS= "s/${exam_class}/${with_print}/"
```



Now we will solve the handouts. What we want to do is to add the `handout` option to the Beamer document class [TWM15].

40a  $\langle \text{variables 38a} \rangle + \equiv$  (37)  $\langle 39e \ 41c \rangle$

```
without_handout= "\\documentclass\\[?(.*)\\]{beamer}"
with_handout=    "\\documentclass\\[\\1,handout\\]{beamer}"
SED_HANDOUT=     "s/${without_handout}/${with_handout}/"
```

## 6.4 Preparing camera-ready source

Sometimes we must prepare ‘camera-ready source’, which essentially means that everything must be contained in a single TeX file. Unfortunately, this is difficult to accomplish with the transformations outlined above<sup>1</sup>. For now, we will use some functions which requires parameters — the transformations above must not require any parameter — so the outline looks like this:

40b  $\langle \text{suffix rules for camera-ready source 40b} \rangle \equiv$  (37)

$\langle \text{function to substitute bibliography for bbl 40c} \rangle$   
 $\langle \text{function to fill filecontents environments 40d} \rangle$   
 $\langle \text{function to insert biblatex bbl code 41a} \rangle$

```
.SUFFIXES: .tex .cameraready.tex
.tex.cameraready.tex:
     $\langle \text{camera-ready recipe 41b} \rangle$ 
```

The first thing we want to do is to replace the `\bibliography` command with the `bbl`-code generated by `bibtex`. This function also takes one argument, the file name of the `bbl`-file (which is the main document name with the `.tex` suffix replaced by `.bbl`).

40c  $\langle \text{function to substitute bibliography for bbl 40c} \rangle \equiv$  (40b)

```
define bibliography
${SED} \
    -e "/\\\\bibliography{[^}]*}/{s/\\\\bibliography.*//;r $(1)}" \
    -e "]"
endef
```

The `bibtex` alternative `biblatex` is becoming more popular. So we want to provide similar functionality for `biblatex`. To do this for `biblatex` we can use the `filecontents` package to include the bibliographies inline.

First we want to use is to fill `filecontents` environments with the actual content from the file. We provide a function which takes the filename as an argument and then uncomments the environment and reads the file contents into the environment.

40d  $\langle \text{function to fill filecontents environments 40d} \rangle \equiv$  (40b)

```
define filecontent
```

---

<sup>1</sup>It is possible and this solution will eventually be converted to such a solution.

```

${SED} "/^%\\\\begin{filecontents}\\*\\?}\\{$(1)}\\/,/^%\\\\end{filecontents}\\*\\?}\\{/s/^%/" \
| ${SED} "/^%\\\\begin{filecontents}\\*\\?}\\{$(1)}\\{/r $(1)"
endif

```

Next, for this to work with `biblatex` we need to insert some extra code.

```

41a  <function to insert biblatex bbl code 41a>≡ (40b)
      define _the_bblcode
      \\\makeatletter\\\def\\\blx@bblfile@biber{\\\blx@secinit\\\begingroup\\\blx@bblstart
      endif

      define bblcode
      ${SED} "s/^%biblatex-bbl-code/${_the_bblcode}/"
      endif

```

Finally, with these functions we can write the following suffix rule, which calls the above functions one by one.

```

41b  <camera-ready recipe 41b>≡ (40b)
      cat $< \
      | $(call filecontent,\
      | $(shell ${SED} -n "s/^%\\\\begin{filecontents}\\*\\?}\\{\\([~]*\\)}\\/\1/p" \
      | $<)) \
      | $(call bibliography,$<:.tex=.bbl) \
      | $(call bblcode) \
      > $@

```

## 6.5 Using encrypted files

The idea of this approach is to encrypt the confidential data in the repository. Thus the repository can be available to everyone, but only those with the decryption keys can read and make sensible changes in the confidential contents.

We will achieve this using the GNU Privacy Guard (GPG) version of Pretty Good Privacy (PGP). We need a command to encrypt, the recipients and a command to decrypt. We will use the following by default.

```

41c  <variables 38a>+≡ (37) <40a
      GPG?=          gpg
      TRANSFORM_ENC?=  ${GPG} -aes
      TRANSFORM_RECIPIENTS?= me
      TRANSFORM_DEC?=  ${GPG} -d

```

This will yield the following suffix rules.

```

41d  <suffix rules for encrypted files 41d>≡ (37)
      .SUFFIXES: .tex .tex.asc
      .tex.tex.asc:
          ${TRANSFORM_ENC} $(foreach r,${TRANSFORM_RECIPIENTS}, -r $r) < $< > $@

```

```
.tex.asc.tex:
    ${TRANSFORM_DEC} < $< > $@
```

An alternative approach, probably less prone to errors, is to use Git. We can use Git's attributes and filter functionality. This means that we apply a filter to all files with the `.asc` suffix. We have two alternatives: do this ourselves or use the `git-crypt` package<sup>2</sup> [Aye]. The set up we must do for this to work is to set a Git attribute.

```
42 <gitattributes 42>≡
    *.asc      filter=git-crypt
```

This will yield similar behaviour as with the makefile approach, except that many things are automated further.

---

<sup>2</sup>Install on Ubuntu by running `sudo apt install git-crypt`.

## Part III

# Papers and documents

# Chapter 7

## tex.mk

### 7.1 Introduction and usage

The aim of this include file is to make building LaTeX documents easier. First we want to add suffix rules for LaTeX similar to those already in `make(1)` [see GNU16, Sect. 10.2] for e.g., (plain) TeX and C.

The main goal is to improve the default rules of GNU make; to change from plain TeX to LaTeX, to change from DVI-format to PDF-format. We provide several suffix rules: first, for ordinary documents, i.e., to compile a `.tex` file to `.pdf`, but we also provide suffix rules for `.dvi`; second, for classes and packages, i.e., to compile 1. a DocTeX `.dtx` file to `.pdf` or `.dvi` and 2. an `.ins` file to `.cls` or `.sty`. The suffix rules we provide here follows the conventions set out in [GNU16, Sect. 10.2].

To do the main LaTeX compilation, we use the following set up. We note that `xelatex` requires the `-8bit` flag to be able to handle tabs in `minted` output. Although, the option `-halt-on-error` seems interesting to stop the compilation if an error occurs, so that we don't miss it, it doesn't work. The reason is that `latexmk` doesn't stop on error, it just keeps on going. TeX on the other hand stops that round of execution, but doesn't ask for input—which it does otherwise. So the `-halt-on-error` option actually makes it more likely that we miss that an error occurred.

```
44 <variables 44>≡ (46) 45a>
    LATEX?=          latexmk -dvi -use-make -xelatex -8bit
    PDFLATEX?=       latexmk -pdf -use-make -xelatex -8bit
    LATEXFLAGS?=
    PREPROCESS.tex?= ${PDFLATEX} ${LATEXFLAGS} $<
    PREPROCESS.dtx?= ${PREPROCESS.tex}
    TEX_OUTDIR?=     ltxobj
    COMPILE.tex?=    \
        ${PDFLATEX} ${LATEXFLAGS} -output-directory=${TEX_OUTDIR} $<; \
        while (grep "Rerun to get cross" ${TEX_OUTDIR}/${<:.tex=.log}); \
```

```

do ${PDFLATEX} ${LATEXFLAGS} -output-directory=${TEX_OUTDIR} $<; \
done
COMPILE.dtx?=      ${COMPILE.tex}

```

Thus one would have to change the `COMPILE.tex` variable to change format from PDF to DVI unless a suffix rule is used.

Normally, the above is all that is needed. However, if you need to manually build the bibliography, you can either add the `.bbl` file as a prerequisite or set the following variable to a non-empty string.

45a `<variables 44>+≡` (46) `<44 45b>`  
`TEX_BBL?=`

Similarly as for the main LaTeX commands, the `bibtex(1)` command is controlled by

45b `<variables 44>+≡` (46) `<45a 45c>`  
`BIBTEX?= bibtexu  
BIBTEXFLAGS?=  
BIBLIOGRAPHY.aux?= ${BIBTEX} ${BIBTEXFLAGS} $<`

And, in case we use `biblatex`, the `biber(1)` command is controlled by

45c `<variables 44>+≡` (46) `<45b 45d>`  
`BIBER?= biber  
BIBERFLAGS?=  
BIBLIOGRAPHY.bcf?= ${BIBER} -O $@ ${BIBERFLAGS} $<`

Similarly as for the bibliography, to enable indexing you can either manually add the `.ind` file as a prerequisite, or you can set the following variable to a non-empty string.

45d `<variables 44>+≡` (46) `<45c 45e>`  
`TEX_IND?=`

The indexing-related programs are the following.

45e `<variables 44>+≡` (46) `<45d 45f>`  
`XINDY?= texindy  
XINDYFLAGS?=  
COMPILE.idx?= ${XINDY} ${OUTPUT_OPTION} ${XINDYFLAGS} $<  
  
MAKEINDEX?= makeindex  
MAKEIDXFLAGS?=  
COMPILE.nlo?= ${MAKEINDEX} ${OUTPUT_OPTION} ${MAKEIDXFLAGS} -s nomencl.ist $<`

We also provide support for PythonTeX. This is enabled by the following variable.

45f `<variables 44>+≡` (46) `<45e 45g>`  
`TEX_PYTHONTEX?=`

Then the required command and flags are controlled with the following variables.

45g `<variables 44>+≡` (46) `<45f 52c>`  
`PYTHONTEX?= python3 $$ (which pythontex)  
PYTHONTEXFLAGS?= -interpreter python:python3`

Finally, we provide targets to easily add external classes as dependencies. We add the phony targets

- `lncs` for Springer Lecture Notes in Computer Science (LNCS),
- `biblatex-lncs` for the LNCS bibliography style for the `biblatex` package,
- `rfc` or `rfc.bib` for an up-to-date bibliography containing all Internet Engineering Task Force (IETF) Request For Commentss (RFCs),
- `popets` for the style files of the Proceedings of the Privacy Enhancing Technologies (PoPETS, De Gruyter Open/Sciendo).

Finally, we add a method to quickly merge bibliographies without redundancy. This is based on the archive syntax of `make(1)`. This allows us to have the resulting full bibliography `full.bib` consist of the subbibliographies `sub0.bib` and `sub1.bib`:

```
1 full.bib: full.bib(sub0.bib sub1.bib)
```

Or, directly as a dependency for the PDF:

```
1 paper.pdf: full.bib(sub0.bib sub1.bib)
```

## 7.2 Implementation overview

The structure of the include file is similar to a header file in C or C++. The include file uses the old C-style technique to prevent multiple inclusions.

```
46 <tex.mk 46>≡
    ifndef TEX_MK
    TEX_MK=true

    .NOTPARALLEL:

    INCLUDE_MAKEFILES?=.
    include ${INCLUDE_MAKEFILES}/portability.mk

    <variables 44>
    <targets for documents 47c>
    <targets for class and package files 53a>
    <targets for external classes 53e>
    <targets for cleaning 47a>

    endif
```

We include `portability.mk` (Chapter 2) to get portable settings for several common utilities. It is currently not possible to run the TeX-builds in parallel, hence the need for the `.NOTPARALLEL` target — this runs any jobs that includes `tex.mk` in serial. However, this is not inherited, any recursive make invocations will run in parallel unless `.NOTPARALLEL` is specified there as well.

We will start with the targets for cleaning. We provide two phony targets, `clean-tex` and `distclean-tex`, and we add them as prerequisites to `clean` and `distclean`, respectively.

```
47a  <targets for cleaning 47a>≡ (46)
      .PHONY: clean clean-tex
      clean: clean-tex

      clean-tex:
        <clean recipe 47b>

      .PHONY: distclean distclean-tex
      distclean: distclean-tex

      distclean-tex:
        <distclean recipe 48b>
```

We will add to the recipes in the remainder of the chapter. However, as `latexmk(1)` is set as the default in Section 7.1, we can already add the following line to the cleaning recipe.

```
47b  <clean recipe 47b>≡ (47a)
      -latexmk -C -output-directory=${TEX_OUTDIR}
      [ "${TEX_OUTDIR}" -ef "$(pwd)" ] || \
        ${RM} -R ${TEX_OUTDIR}
      ${RM} *.pytxcode
      ${RM} -R pythontex-files-*
```

## 7.3 Targets for documents

Now we will treat how to compile documents.

```
47c  <targets for documents 47c>≡ (46)
      <auxillary files 48c>
      <bibliography files 48e>
      <indices files 49d>
      <PythonTeX files 50d>
      <document files 52a>
      <target for latexmkrc 48a>
```

These will be discussed in the following sections. However, since we use `latexmk(1)` by default (Section 7.1), we will discuss the relevant `<latexmkrc (never defined)>`



entries in parallel. We supply a target to easily use our *latexmkrc* (never defined) with `latexmk(1)`.

48a  $\langle$ *target for latexmkrc* 48a $\rangle \equiv$  (47c)  
`latexmkrc:`  
`[ -e $@ -o "${INCLUDE_MAKEFILES}" = "." ] || \`  
`${LN} -s ${INCLUDE_MAKEFILES}/latexmkrc $@`

We also add the corresponding line for cleaning.

48b  $\langle$ *distclean recipe* 48b $\rangle \equiv$  (47a)  
`[ ! -L latexmkrc ] || ${RM} latexmkrc`

We will now discuss the different files we need `latex(1)` to generate. Note that in many cases we want `latex(1)` to generate more files, e.g., `.toc` files, but we do not have to care about these here. The reason we can ignore those files is that they do not require any external tool, e.g., `bibtex(1)`, to be run, these files just requires another run of `latex(1)`.

### 7.3.1 Auxillary files

Many steps in compiling a LaTeX document needs the `.aux` file. Thus we will first introduce a rule for creating the `.aux` file. We will create it in the specified output directory.

48c  $\langle$ *auxillary files* 48c $\rangle \equiv$  (47c)  
`${TEX_OUTDIR}/%.aux: %.tex`  
 $\langle$ *create output directory and preprocess TeX* 48d $\rangle$

Then we simply do

48d  $\langle$ *create output directory and preprocess TeX* 48d $\rangle \equiv$  (48–50)  
`${MKDIR} ${TEX_OUTDIR}`  
`${PREPROCESS}.tex}`

### 7.3.2 Bibliographies

One file that is commonly needed is the one used to create the bibliography, the `.bbl` file. There are two ways to create this file, either using classical `bibtex(1)` or using the `biblatex` package and `biber(1)`.

48e  $\langle$ *bibliography files* 48e $\rangle \equiv$  (47c) 49c $\triangleright$   
 $\langle$ *bbl target for bibtex* 48f $\rangle$   
 $\langle$ *bbl target for biber* 49a $\rangle$

The first approach, using `bibtex(1)`, depends on the `.aux` file. This means that we can have a target creating the desired `.bbl` file from the `.aux` file.

48f  $\langle$ *bbl target for bibtex* 48f $\rangle \equiv$  (48e)  
`${TEX_OUTDIR}/%.bbl: ${TEX_OUTDIR}/%.aux`  
`${BIBLIOGRAPHY}.aux}`  
`${MV} $@ ${@:.bbl=.b1g} ${TEX_OUTDIR}`

The second approach uses the `biblatex` package [Leh+16] and `biber(1)` [KC16]. They do not rely on the `.aux` file, instead `biblatex` creates a `.bcf` file. Thus its target is exactly the same as that of the `.aux` file.

49a `\bbl target for biber 49a`≡ (48e) 49b>  
`\${TEX_OUTDIR}/%.bcf: %.tex`  
`\create output directory and preprocess TeX 48d`

This `.bcf` file is in turn used by `biber(1)` to create the `.bbl` file. To compile the `.bbl` with `biber(1)` and put the output files in the desired directory, we do the following.

49b `\bbl target for biber 49a`+≡ (48e) <49a  
`\${TEX_OUTDIR}/%.bbl: \${TEX_OUTDIR}/%.bcf`  
`\${BIBLIOGRAPHY}.bcf`

As mentioned in Section 7.1, we can automatically add the `.bbl` file as a prerequisite if the variable `TEX_BBL` is set.

49c `\bibliography files 48e`+≡ (47c) <48e 50e>  
`ifneq (\${TEX_BBL},)`  
`%.pdf \${TEX_OUTDIR}/%.pdf: \${TEX_OUTDIR}/%.bbl`  
`endif`

### 7.3.3 Indices

There are several time we need to work with indices, e.g., when working with standard indices but also glossaries. Here we provide some suffix rules to make it easier to build such indices.

Before we start, however, we will note that many of these rules are not needed if the `imakeidx` package [Gre16] is used. We do recommend to use this package. Furthermore, we provide rules for the `nomenc1` package, however, we recommend to use the `glossaries` package [Tal16] instead. The `glossaries` package also has native support for `xindy(1)`. Although the `glossaries` package supports abbreviations and acronyms, we recommend the `acro` package [Nie16] for this instead.

The standard LaTeX index uses an `.idx` file, which is generated similarly as the `.aux` file. Thus we can use the same type of target.

49d `\indices files 49d`≡ (47c) 49e>  
`\${TEX_OUTDIR}/%.idx: %.tex`  
`\create output directory and preprocess TeX 48d`

The actual index, which resides in a `.ind` file, can then be generated as follows.

49e `\indices files 49d`+≡ (47c) <49d 50a>  
`\${TEX_OUTDIR}/%.ind: \${TEX_OUTDIR}/%.idx`  
`\${COMPILE}.idx`

As mentioned in Section 7.1, we can automatically add the `.ind` file as a prerequisite if the variable `TEX_IND` is set.

```
50a <indices files 49d>+≡ (47c) <49e 50b>
    ifneq (${TEX_IND},)
    %.pdf ${TEX_OUTDIR}/%.pdf: ${TEX_OUTDIR}/%.ind
    endif
```

For backwards compatibility, we provide the following code for the `nomenclature` package.

```
50b <indices files 49d>+≡ (47c) <50a
    ${TEX_OUTDIR}/%.nlo: %.tex
    <create output directory and preprocess TeX 48d>
```

```
    ${TEX_OUTDIR}/%.nls: ${TEX_OUTDIR}/%.nlo
    ${COMPILE.nlo}
```

And now we add the corresponding code for `latexmk(1)`. The code is fetched from the `latexmk` example-files on Comprehensive TeX Archive Network (CTAN)<sup>1</sup>.

```
50c <latexmkrc 50c>≡ 51>
    add_cus_dep( 'nlo', 'nls', 0, 'makenlo2nls' );
    sub makenlo2nls {
        system( "makeindex -s nomencl.ist -o \"$_[0].nls\" \"$_[0].nlo\"" );
    }
```

### 7.3.4 PythonTeX

Occasionally we use PythonTeX. We also provide a target for the required files. However, this construction requires that we load PythonTeX as follows.

```
\usepackage{pythontex}
\setpythontexoutputdir{.}
\setpythontexworkingdir{..}
```

The code is as follows.

```
50d <PythonTeX files 50d>≡ (47c)
    ${TEX_OUTDIR}/%.pytxcode: ${TEX_OUTDIR}/%.aux
    cd $(dir $@) && ${PYTHONTEX} ${PYTHONTEXFLAGS} $(basename $(notdir $@))
    %.pytxmcr ${TEX_OUTDIR}/%.pytxmcr:: ${TEX_OUTDIR}/%.pytxcode
    cd ${TEX_OUTDIR} && ${PYTHONTEX} ${PYTHONTEXFLAGS} $(basename $(notdir $@))
```

As mentioned in Section 7.1, we can automatically add the `.pytxmcr` file as a prerequisite if the variable `TEX_PYTHONTEX` is set.

```
50e <bibliography files 48e>+≡ (47c) <49c
    ifneq (${TEX_PYTHONTEX},)
    ${TEX_OUTDIR}/%.pdf: ${TEX_OUTDIR}/%.pytxmcr
    endif
```

---

<sup>1</sup>URL: [http://mirrors.ctan.org/support/latexmk/example\\_rcfiles/nomenclature\\_latexmkrc](http://mirrors.ctan.org/support/latexmk/example_rcfiles/nomenclature_latexmkrc)

If we use `latexmk(1)`, then we must also add instructions for this in `<latexmkrc` (never defined). The following code is fetched from the `latexmk` example-files on CTAN<sup>2</sup>.

```

51 <latexmkrc 50c>+≡<50c
# This version has a fudge on the latex and pdflatex commands that
# allows the pythontex custom dependency to work even when $out_dir
# is used to set the output directory. Without the fudge (done by
# tricky symbolic links) the custom dependency for using pythontex
# will not be detected.

add_cus_dep('pytxcode', 'pytxmcr', 0, 'pythontex');
sub pythontex {
    # This subroutine is a fudge, because it from latexmk's point of
    # view, it makes the main .tex file depend on the .pytxcode file.
    # But it doesn't actually make the .tex file, but is used for its
    # side effects in creating other files. The dependence is a way
    # of triggering the rule to be run whenever the .pytxcode file
    # changes, and to do this before running latex/pdflatex again.
    return system("pythontex3 -verbose \"$_[0]\"");
}

$pdflatex = 'internal mylatex %R %Z pdflatex %O %S';
$latex = 'internal mylatex %R %Z latex %O %S';
sub mylatex {
    my $root = shift;
    my $dir_string = shift;
    my $code = "$root.pytxcode";
    my $result = "pythontex-files-$root";
    if ($dir_string) {
        warn "mylatex: Making symlinks to fool cus_dep creation\n";
        unlink $code;
        if (-l $result) {
            unlink $result;
        }
        elsif (-d $result) {
            unlink glob "$result/*";
            rmdir $result;
        }
        symlink $dir_string.$code, $code;
        if ( ! -e $dir_string.$result ) { mkdir $dir_string.$result; }
        symlink $dir_string.$result, $result;
    }
    else {

```

---

<sup>2</sup>URL: [http://mirrors.ctan.org/support/latexmk/example\\_rcfiles/pythontex-latexmkrc](http://mirrors.ctan.org/support/latexmk/example_rcfiles/pythontex-latexmkrc)

```

        foreach ($code, $result) { if (-1) { unlink; } }
    }
    return system @_;
}

```

### 7.3.5 Document files

Now that we have all prerequisite files, we can actually compile the document. For simplicity we add file in both the current working directory and the `TEX_OUTDIR` as targets. The reason for this is that it makes the makefile easier to write, usually we prefer writing just the `.pdf` file — and not the path to the `.pdf` in the `TEX_OUTDIR` directory. And for the same reason, we create a symbolic link between them after compilation — this allows `make(1)` to track modification times correctly. (We need a symbolic link, rather than a hard link, because upon recompilation, the file in `TEX_OUTDIR` is unlinked, whereas the other is not.)

52a `<document files 52a>≡` (47c)

```

%.pdf ${TEX_OUTDIR}/%.pdf: %.tex
    ${COMPILE.tex}
    -${LN} ${TEX_OUTDIR}/$@ $@

```

```

%.dvi ${TEX_OUTDIR}/%.dvi: %.tex
    <compile DVI 52b>
    -${LN} ${TEX_OUTDIR}/$@ $@

```

The compilation step for DVI is the usual. We compile once, then we recompile as long as the log file tells us.

52b `<compile DVI 52b>≡` (52a 53c)

```

${LATEX} -output-directory=${TEX_OUTDIR} ${LATEXFLAGS} $<
while ( grep "Rerun to get cross" ${TEX_OUTDIR}/${<:.tex=.log} ); do \
    ${LATEX} -output-directory=${TEX_OUTDIR} ${LATEXFLAGS} $<; \
done

```

## 7.4 Targets for merging bibliographies

To merge bibliographies we will use `bibttool` and `make`'s archive rules. In particular, we extend the general rules we defined in `portability.mk`.

52c `<variables 44>+≡` (46) <45g

```

BIBTOOL?=      bibttool
BIBTOOLFLAGS?=-preserve.key.case=on -print.deleted.entries=off -s -d -r biblatex
ARCHIVE.bib?=  ${CAT} $(if $(wildcard $@),$@) $% | \
    ${BIBTOOL} ${BIBTOOLFLAGS} -o $@

```

## 7.5 Targets for class and package files

There are two parts concerning class and package files.

53a  $\langle$ *targets for class and package files 53a* $\rangle \equiv$  (46)  
 $\langle$ *compile sty and cls files 53b* $\rangle$   
 $\langle$ *compile class and package documentation 53c* $\rangle$

These are very similar to what we have done above, especially the documentation.

Compiling a class or package from DocTeX source is easier than compiling a document. We can normally create the .sty and .cls files by running latex(1) on the .ins file.

53b  $\langle$ *compile sty and cls files 53b* $\rangle \equiv$  (53a)  
 $\%.\text{cls} \ \%.\text{sty}: \ \%.\text{ins}$   
 $\$\{\text{LATEX}\} \ \$<$

We can then compile the documentation similarly to how we compile normal documents.

53c  $\langle$ *compile class and package documentation 53c* $\rangle \equiv$  (53a) 53d $\triangleright$   
 $\%.\text{pdf} \ \${\text{TEX\_OUTDIR}}/\%.\text{pdf}: \ \%.\text{dtx}$   
 $\$\{\text{COMPILE.dtx}\}$   
 $-\$\{\text{LN}\} \ \${\text{TEX\_OUTDIR}}/\$@ \ \$@$   
  
 $\%.\text{dvi} \ \${\text{TEX\_OUTDIR}}/\%.\text{dvi}: \ \%.\text{dtx}$   
 $\langle$ *compile DVI 52b* $\rangle$   
 $-\$\{\text{LN}\} \ \${\text{TEX\_OUTDIR}}/\$@ \ \$@$

However, we must tell make(1) how to make a .bbl etc. from .dtx.

53d  $\langle$ *compile class and package documentation 53c* $\rangle + \equiv$  (53a)  $\triangleleft$ 53c  
 $\$\{\text{TEX\_OUTDIR}}/\%.\text{aux} \ \${\text{TEX\_OUTDIR}}/\%.\text{bcf} \ \${\text{TEX\_OUTDIR}}/\%.\text{idx}: \ \%.\text{dtx}$   
 $\$\{\text{PREPROCESS.dtx}\}$

## 7.6 External classes and packages

Occasionally, we are required to use document classes that are not in CTAN. Here we provide targets for some such classes and packages.

53e  $\langle$ *targets for external classes 53e* $\rangle \equiv$  (46)  
 $\langle$ *a general downloader 54a* $\rangle$   
 $\langle$ *Springer LNCS 56a* $\rangle$   
 $\langle$ *biblatex LNCS style 56c* $\rangle$   
 $\langle$ *the RFC bibliography 56d* $\rangle$   
 $\langle$ *PoPETS 57e* $\rangle$

We will now construct a general downloader, then we will use this downloader to write targets for the external classes we are interested in. In general, what we want this function to do is to download an archive or repository (`TEX_EXT_SRC`), extract the files we are interested in (`TEX_EXT_FILES`) to the destination directory (`TEX_EXT_DIR`), and finally, create symbolic links to those files from the current working directory.

```
54a  <a general downloader 54a>≡ (53e)
      define download_archive
      <targets for symlinks 54b>
      <targets for desired files 54d>
      <targets for archive 55a>
      <target for cleaning 55e>
      endif
      define download_repo
      <targets for symlinks 54b>
      <targets for desired files 54d>
      <targets for repo 55b>
      <target for cleaning 55e>
      endif
```

The first thing we want to do is to generate targets and recipes for how to create the symbolic links, assuming that the target files already exists. First we set up a dependency between the files we are interested in and where that file is actually located. Then we create a recipe which will create a symbolic link between them. We use the `notdir` function [GNU16, Sect. 8.3] to remove any directory-part since we want to create the symbolic link in the current working directory.

```
54b  <targets for symlinks 54b>≡ (54a) 54c>
      $(foreach file,${TEX_EXT_FILES-$(1)},\
        $(eval $(notdir ${file}): ${TEX_EXT_DIR-$(1)}/${file}))
      $(notdir ${TEX_EXT_FILES-$(1)}):
      ${LN} $$~ $$$@
```

Note that we need the `eval` command above to evaluate the rule for each file, otherwise we would get *one line* with many colons — which is not valid syntax for `make(1)`. To make it easier to add these files as prerequisites to a target, we also provide the following phony target.

```
54c  <targets for symlinks 54b>+≡ (54a) <54b>
      .PHONY: $(1)
      $(1): $(notdir ${TEX_EXT_FILES-$(1)})
```

Now we need something to trigger the download of the archive or the repository. One way to do this is to add a prerequisite for the files from the archive or repository.

```
54d  <targets for desired files 54d>≡ (54a)
      $(addprefix ${TEX_EXT_DIR-$(1)}/,${TEX_EXT_FILES-$(1)}): \
        ${TEX_EXT_DIR-$(1)}/${TEX_EXT_SRC-$(1)}
```

Now we turn to the recipe. In the case of an archive, we must extract the desired files. We let the variable `TEX_EXT_EXTRACT` contain the extraction command.

```
55a <targets for archive 55a>≡ (54a) 55c>
    $(addprefix ${TEX_EXT_DIR-$(1)}/,${TEX_EXT_FILES-$(1)}):
        ${TEX_EXT_EXTRACT-$(1)}
```

For a repository we can simply copy the file or create a link. We prefer the latter.

```
55b <targets for repo 55b>≡ (54a) 55d>
    $(addprefix ${TEX_EXT_DIR-$(1)}/,${TEX_EXT_FILES-$(1)}):
        ${LN} ${TEX_EXT_SRC-$(1)}/${@:${TEX_EXT_DIR-$(1)}/%=%} $$@
```

The file `TEX_EXT_SRC` can be either an archive or a repository. We let `TEX_EXT_URL` be the uniform resource location (URL) to fetch it from in both cases. In the case of an archive we do the following.

```
55c <targets for archive 55a>+≡ (54a) <55a
    ${TEX_EXT_DIR-$(1)}/${TEX_EXT_SRC-$(1)}:
        ${MKDIR} ${TEX_EXT_DIR-$(1)}
        ${CURL} -o $$@ ${TEX_EXT_URL-$(1)}
```

In the case of a repository, we simply clone it.

```
55d <targets for repo 55b>+≡ (54a) <55b
    ${TEX_EXT_DIR-$(1)}/${TEX_EXT_SRC-$(1)}:
        git clone ${TEX_EXT_URL-$(1)} $$@
```

We note that the directory of the repo should be an order-only prerequisite [see GNU16, Sect. 4.3] for the files inside. Unfortunately this is not the case at the moment.

Finally, we must also do some cleaning.

```
55e <target for cleaning 55e>≡ (54a)
.PHONY: distclean clean-$(1)
distclean: clean-$(1)
clean-$(1):
    ${RM} ${TEX_EXT_FILES-$(1)}
    [ "${TEX_EXT_DIR-$(1)}" = "." ] && ${RM} ${TEX_EXT_SRC-$(1)} \
    || ${RM} -R ${TEX_EXT_DIR-$(1)}
```

### 7.6.1 Springer LNCS

Springer's LNCS series is used for the proceedings of many conferences. The style files are available on Springer's web site, but unfortunately not under any permissive license<sup>3</sup>. So, we must, each and every one of us, connect to Springer's server and download our own copy. This is what we automate here.

---

<sup>3</sup>It would be most desirable that they were made available in CTAN under an open license.



We use the downloader described above.

```
56a  <Springer LNCS 56a>≡ (53e) 56b>
      TEX_EXT_FILES-lncs?=  llncs.cls sprmindx.sty splncs03.bst aliascnt.sty remreset.sty
      TEX_EXT_DIR-lncs?=    lncs
      TEX_EXT_SRC-lncs?=    llncs2e.zip
      TEX_EXT_URL-lncs?=    https://resource-cms.springernature.com/springer-cms/rest/v1/content
      TEX_EXT_EXTRACT-lncs?=${UNZIP} -u $$< -d ${TEX_EXT_DIR-lncs}

      $(eval $(call download_archive,lncs))
```

We also want to add backwards compatibility for when we used `llncs` instead of just `lncs`.

```
56b  <Springer LNCS 56a>+≡ (53e) <56a
      .PHONY: llncs
      llncs: lncs
```

### 7.6.2 LNCS style for biblatex

There is also an LNCS style for the `biblatex` package available on GitHub. Since it is available on GitHub, we recommend adding it as a Git submodule. I.e. run the following command.

```
1  git submodule add https://github.com/neapel/biblatex-lncs.git
```

This will add a directory `biblatex-lncs` to the current directory.

If we do not add it as a submodule we can use the downloader above.

```
56c  <biblatex LNCS style 56c>≡ (53e)
      TEX_EXT_FILES-biblatex-lncs?= lncs.bbx lncs.cbx lncs.dbx
      TEX_EXT_DIR-biblatex-lncs?=    lncs
      TEX_EXT_SRC-biblatex-lncs?=    biblatex-lncs
      TEX_EXT_URL-biblatex-lncs?=    https://github.com/NorwegianRockCat/biblatex-lncs.git

      $(eval $(call download_repo,biblatex-lncs))
```

### 7.6.3 The RFC bibliography

Occasionally we want to cite IETF RFCs. Fortunately, Roland Bless of Karlsruher Institute of Technology provides an up-to-date bibliography file for all RFCs, so we will use that one. This is a single file, so we do not need to use the downloader.

```
56d  <the RFC bibliography 56d>≡ (53e) 57c>
      rfc.bib:
        <download rfc.bib 57a>
        <change misc to techreport 57b>

      ${TEXMF}/tex/latex/rfc.bib:
```

```
mkdir -p ${TEXMF}/tex/latex/
⟨download rfc.bib 57a⟩
⟨change misc to techreport 57b⟩
```

We will use curl(1) to download a compressed version from Bless' site. We let curl(1) output the contents to standard out and pipe it to the uncompress(1) utility and, finally, redirect the result to the target file.

```
57a  ⟨download rfc.bib 57a⟩≡ (56d)
      ${CURL} -o - http://tm.uka.de/~bless/rfc.bib.gz 2>/dev/null \
      | ${UNCOMPRESS}.gz - > $@ ; \
```

According to IETF [CP11, Sect. 5.2] the RFCs should be cited as the techreport BibTeX type.

```
57b  ⟨change misc to techreport 57b⟩≡ (56d)
      ${SED} -i "s/@misc/@techreport/" $@
```

We also provide a phony target for these two files.

```
57c  ⟨the RFC bibliography 56d⟩+≡ (53e) <56d 57d>
      .PHONY: rfc
      rfc: rfc.bib ${TEXMF}/tex/latex/rfc.bib
```

Finally, we provide a phony cleaning for cleaning. The target is named clean-rfc and is added as a prerequisite for distclean — this way its recipe will not interfere with any cleaning recipe written by the user.

```
57d  ⟨the RFC bibliography 56d⟩+≡ (53e) <57c
      .PHONY: distclean clean-rfc
      distclean: clean-rfc
      clean-rfc:
          ${RM} rfc.bib
```

## 7.6.4 Proceedings of the Privacy Enhancing Technologies Symposium

We would also like to be able to use the PoPETS format.

```
57e  ⟨PoPETS 57e⟩≡ (53e)
      TEX_EXT_FILES-popets?=by-nc-nd.pdf sciendo-logo.pdf dgruyter_NEW.sty
      TEX_EXT_URL-popets?=https://petsymposium.org/files/popets.zip
      TEX_EXT_DIR-popets?=popets
      TEX_EXT_SRC-popets?=popets.zip
      TEX_EXT_EXTRACT-popets?=${UNZIP} -p $$$ popets/$(notdir $$$) > $$$

      $(eval $(call download_archive,popets))
```

# Chapter 8

## doc.mk

### 8.1 Introduction and usage

When working with large sets of documents we sometimes want to do some operations on them, e.g., print them or convert them between formats. This include file provides exactly that.

We provide a target **print** which prints its prerequisites, see Section 8.2.1 for details. It prints them using `${LPR} ${LPRFLAGS} <file>`. Each of these variables has specialized `LPR-<file>` and `LPRFLAGS-<file>` versions, which means that different documents can be printed with different commands and flags.

We also provide a target **wc** which counts the words of its prerequisites (Section 8.2.2). The counting is done with `${WC} ${WCFLAGS} <file>`. There is also a **todo** target (Section 8.2.3) which grep's its prerequisites for common todo comments (e.g., TODO, XXX, FIXME).

Each of **wc** and **todo** has the specialized `-<file>` versions of the variables, same as for **print**. However, **wc** and **todo** additionally have a **PREWC** and **PRETODO** commands which pipe their output to the **WC** and **TODO** commands, respectively. These variables have `-<file>` versions, but no flags variable.

Finally, we also provide a set of suffix rules for automatic conversion between different formats, see Section 8.2.4 for details about the formats. In general, we provide the variable `CONVERT.a.b` which can be used in recipes to convert a file of format `.a` to a file of format `.b`. The command `run` is designed to fit a rule of the form `%.b: %.a`.

### 8.2 Implementation

Since the makefile is designed for inclusion, we want to ensure that it is not included more than once — like we do in C and C++. Then first comes our variables described above followed by the targets.

```
58 <doc.mk 58>≡  
    ifndef DOC_MK
```

```

DOC_MK=true

INCLUDE_MAKEFILES?=
include ${INCLUDE_MAKEFILES}/portability.mk

<variables 59a>
<target for printing 59b>
<target for word counting 60b>
<target for to-do lists 60c>
<suffix rules for format conversion 61a>

endif

```

### 8.2.1 Printing

We provide a target **print** to print all documents in a set. The usage is simply that documents are added as prerequisites, then the target prints all documents in its dependency list. The printing is done using the **lpr** command by default. However, this can be changed with the **LPR** variable.

```

59a <variables 59a>≡ (58) 60a>
    LPR?=          lpr
    LPRFLAGS?=

```

If **lpr** is used, we note that the files added as prerequisites for the **print** target must be printable by **lpr**, e.g., we must supply PostScript-files instead of PDF-files. Fortunately, the automatic file format conversion (Section 8.2.4) solves most of those problems. For example, if you want to print a PDF-file **something.pdf**, then just add **something.ps** as a prerequisite to print and the suffix rules below will do the rest.

The implementation is quite simple. We will iterate through the list of prerequisites and process them one by one. For each document we will check if there is an overriding setting for the printing command and its arguments, if there is not we use the default set above.

```

59b <target for printing 59b>≡ (58)
    .PHONY: print
    print:
        $(foreach doc,$^,\
            $(if ${LPR-${doc}},${LPR-${doc}},${LPR}) \
            $(if ${LPRFLAGS-${doc}},${LPRFLAGS-${doc}},${LPRFLAGS}) \
            ${doc};)

```

### 8.2.2 Counting words

We provide a **wc** target which counts the words in its prerequisites. The files added as prerequisites must thus be text files. Similarly as for print, there are suffix rules to convert e.g., TeX-files to plain text files using **detex**.

The implementation is similar to that for `print`. The counting is done using the `wc` command by default, but we allow overrides using the following variable.

```
60a <variables 59a>+≡ (58) <59a 60d>
      WC?=          wc
      WCFLAGS?=     -w
```

We will simply iterate through the list of prerequisites and process them one by one using `wc`. We first print the name followed by a colon, then we print the word count. Similarly as above, we check for each document whether there is an overriding setting for the word counting command. We also check if there is a preprocessing command, e.g., it might be useful to run `detex(1)` on TeX files before counting the words.

```
60b <target for word counting 60b>≡ (58)
      .PHONY: wc
      wc:
      $(foreach doc,$^,echo -n "${doc}: "; ${CAT} ${doc} | \
        $(if ${PREWC}-${doc}},${PREWC}-${doc} |, $(if ${PREWC},${PREWC} |,)) \
        $(if ${WC}-${doc}},${WC}-${doc}},${WC}) \
        $(if ${WCFLAGS}-${doc}},${WCFLAGS}-${doc}},${WCFLAGS});)
```

### 8.2.3 To-do lists

Similarly to the `wc` target, we would also like to add a `todo` target which generates a to-do list from the to-do comments in the source files (i.e., ‘TODO’, ‘XXX’ or ‘FIXME’).

```
60c <target for to-do lists 60c>≡ (58)
      .PHONY: todo
      todo:
      $(foreach doc,$^,echo "${doc}: "; ${CAT} ${doc} | \
        $(if ${PRETODO}-${doc}},${PRETODO}-${doc} |, $(if ${PRETODO},${PRETODO} |,)) \
        $(if ${TODO}-${doc}},${TODO}-${doc}},${TODO}) \
        $(if ${TODOFLAGS}-${doc}},${TODOFLAGS}-${doc}},${TODOFLAGS});echo;)
```

We will use the `grep(1)` utility to grep for these flags in the files.

```
60d <variables 59a>+≡ (58) <60a 61b>
      TODO?=      ${GREP} "\(XXX\|TODO\|FIXME\)"
      TODOFLAGS?=
```

### 8.2.4 Format conversion

We often want to convert a file from one format to another. We will now write a pattern along the lines of make’s conventions for compilation targets [GNU16, Sect. 10.2, second to last paragraph]. I.e., `CONVERT.a.b` will convert a file with suffix `.a` to a format with suffix `.b`. However, we will only provide pattern

rules for the most common direction, e.g., from `.a` to `.b` but not from `.b` to `.a`. Having both will cause make to drop one due to circular dependencies.

We provide the following patterns:

61a  $\langle \textit{suffix rules for format conversion 61a} \rangle \equiv$  (58)

$\langle PS \textit{ to PDF 61c} \rangle$   
 $\langle DVI \textit{ to PS 61e} \rangle$   
 $\langle ODF \textit{ to PDF 62b} \rangle$   
  
 $\langle SVG \textit{ to PDF 62d} \rangle$   
 $\langle SVG \textit{ to PS 63a} \rangle$   
 $\langle DIA \textit{ to TeX 63c} \rangle$   
 $\langle XCF \textit{ to PNG 63e} \rangle$   
  
 $\langle PDF \textit{ to cropped PDF 63f} \rangle$   
  
 $\langle MD \textit{ to TeX 64b} \rangle$   
 $\langle MD, TeX \textit{ to HTML 64d} \rangle$   
 $\langle TeX \textit{ to text 65} \rangle$

## Document formats

To convert PDFs to PostScript format, we use the `pdf2ps` command by default.

61b  $\langle \textit{variables 59a} \rangle + \equiv$  (58)  $\langle 60d \ 61d \rangle$

`PDF2PS?= pdf2ps`  
`PDF2PSFLAGS?=`  
`CONVERT.pdf.ps?= ${PDF2PS} ${PDF2PSFLAGS} $<`  
  
`PS2PDF?= ps2pdf`  
`PS2PDFFLAGS?=`  
`CONVERT.ps.pdf?= ${PS2PDF} ${PS2PDFFLAGS} $<`

This allows us to specify the rules as follows.

61c  $\langle PS \textit{ to PDF 61c} \rangle \equiv$  (61a)

`%.pdf: %.ps`  
`${CONVERT.pdf.ps}`

We do similarly for DVI-files that we want to convert to PostScript.

61d  $\langle \textit{variables 59a} \rangle + \equiv$  (58)  $\langle 61b \ 62a \rangle$

`DVIPS?= dvips`  
`DVIPSFLAGS?=`  
`CONVERT.dvi.ps?= ${DVIPS} ${DVIPSFLAGS} $<`

With those variables we let

61e  $\langle DVI \textit{ to PS 61e} \rangle \equiv$  (61a)

`%.ps: %.dvi`  
`${CONVERT.dvi.ps}`

There is no good conversion program for the Open Document Format (ODF) files. We will use LibreOffice although that will incur some manual labour.

62a  $\langle \text{variables 59a} \rangle + \equiv$  (58)  $\langle 61d \ 62c \rangle$

```

ODF2PDF?=      soffice -convert-to pdf
ODF2PDFFLAGS?= -headless
CONVERT.odt.pdf?= ${ODF2PDF} ${ODF2PDFFLAGS} $<
CONVERT.ods.pdf?= ${ODF2PDF} ${ODF2PDFFLAGS} $<
CONVERT.odg.pdf?= ${ODF2PDF} ${ODF2PDFFLAGS} $<
CONVERT.odp.pdf?= ${ODF2PDF} ${ODF2PDFFLAGS} $<
CONVERT.doc.pdf?= ${ODF2PDF} ${ODF2PDFFLAGS} $<
CONVERT.docx.pdf?= ${ODF2PDF} ${ODF2PDFFLAGS} $<
CONVERT.xls.pdf?= ${ODF2PDF} ${ODF2PDFFLAGS} $<
CONVERT.xlsx.pdf?= ${ODF2PDF} ${ODF2PDFFLAGS} $<
CONVERT.ppt.pdf?= ${ODF2PDF} ${ODF2PDFFLAGS} $<
CONVERT.pptx.pdf?= ${ODF2PDF} ${ODF2PDFFLAGS} $<

```

Then we can generate the suffix rules as follows.

62b  $\langle \text{ODF to PDF 62b} \rangle \equiv$  (61a)

```

define def_convert_rule
%.pdf: %.$(1)
    ${CONVERT.$(1).pdf}
endef
$(foreach suf,odt ods odg odp doc docx xls xlsx ppt pptx,\
    $(eval $(call def_convert_rule,${suf})))

```

## Figure formats

Usually we want to keep figures in their source form, so that we can still edit them later. However, just as usually, we cannot use the source form directly in TeX documents, so we want to convert them to TeX or PDF.

When working with SVG-files, there are two things: the graphics and the text in the graphics. We will use Inkscape for working with SVGs, because Inkscape allows us to export the graphics part as PDF and all text in it as TeX. Unlike previously, we will only allow flags for `inkscape` to be set.

62c  $\langle \text{variables 59a} \rangle + \equiv$  (58)  $\langle 62a \ 63b \rangle$

```

INKSCAPE?=      inkscape
INKSCAPEFLAGS?= -D -z -export-latex
CONVERT.svg.pdf?= ${INKSCAPE} ${INKSCAPEFLAGS} -file=$< -export-pdf=$@
CONVERT.svg.ps?=  ${INKSCAPE} ${INKSCAPEFLAGS} -file=$< -export-ps=$@
CONVERT.svg.eps?= ${INKSCAPE} ${INKSCAPEFLAGS} -file=$< -export-eps=$@

```

62d  $\langle \text{SVG to PDF 62d} \rangle \equiv$  (61a)

```

%.pdf: %.svg
    ${CONVERT.svg.pdf}

```

We can thus create similar rules for the formats PS and EPS, instead of PDF.

63a  $\langle SVG \text{ to } PS \text{ 63a} \rangle \equiv$  (61a)  
 $\%.\text{ps} \ \%.\text{eps}: \ \%.\text{svg}$   
 $\$\{\text{CONVERT.svg}\}(\text{suffix } \$@)\}$

Dia is a useful tool for making figures over network topologies etc. Fortunately, Dia can output native TeX. Similarly to Inkscape, we will only provide flags for Dia.

63b  $\langle \text{variables 59a} \rangle + \equiv$  (58)  $\langle 62c \ 63d \rangle$   
 $\text{DIA?} = \text{dia}$   
 $\text{DIAFLAGS?} =$   
 $\text{CONVERT.dia.tex?} = \ \$\{\text{DIA}\} \ \$\{\text{DIAFLAGS}\} \ -e \ \$@ \ -t \ \text{pgf-tex} \ \$\langle$

That gives the suffix rule as follows.

63c  $\langle DIA \text{ to } TeX \text{ 63c} \rangle \equiv$  (61a)  
 $\%.\text{tex}: \ \%.\text{dia}$   
 $\$\{\text{CONVERT.dia.tex}\}$

Gimp is a useful tool for raster graphics. We can convert Gimps own format XCF to PNG using the ImageMagic library.

63d  $\langle \text{variables 59a} \rangle + \equiv$  (58)  $\langle 63b \ 63g \rangle$   
 $\text{XCF2PNGFLAGS?} = \text{-flatten}$   
 $\text{XCF2PNG?} = \text{convert } \ \$\{\text{XCF2PNGFLAGS}\} \ \$\langle \ \$@$   
 $\text{TRIM?} = \text{convert -trim } \$@ \ \$@$   
 $\text{CONVERT.xcf.png?} = \ \$\{\text{XCF2PNG}\}$

Then we can have the following rule:

63e  $\langle XCF \text{ to } PNG \text{ 63e} \rangle \equiv$  (61a)  
 $\%.\text{png}: \ \%.\text{xcf}$   
 $\$\{\text{CONVERT.xcf.png}\}$   
 $\$\{\text{TRIM}\}$

## Cropping PDFs

Sometimes we want to crop PDFs, e.g., to remove the margins of figures drawn on the reMarkable. When we do this, we also want to create a corresponding `.pdf_tex` file. For this, we assume that the only `.pdf` in the file is the filename.

63f  $\langle PDF \text{ to cropped PDF 63f} \rangle \equiv$  (61a)  
 $\%.\text{cropped.pdf}: \ \%.\text{pdf}$   
 $\$\{\text{PDFCROP}\} \ \$\{\text{PDFCROPFLAGS}\} \ \$\langle \ \$@$   
 $\%.\text{cropped.pdf\_tex}: \ \%.\text{pdf\_tex} \ \%.\text{cropped.pdf}$   
 $\text{sed "s/\.pdf/\.cropped\.pdf/g" } \$\langle \ > \ \$@$

63g  $\langle \text{variables 59a} \rangle + \equiv$  (58)  $\langle 63d \ 64a \rangle$   
 $\text{PDFCROP?} = \text{pdfcrop}$   
 $\text{PDFCROPFLAGS?} =$



## Text-based formats

The conversion of the text-based formats differ from the formats above. Most of these tools automatically write their output to stdout, which is customary when working with text in the terminal.

We use the **pandoc** program to convert between Markdown and TeX.

```
64a  <variables 59a>+≡ (58) <63g 64c>
      MD2TEX?=          pandoc
      MD2TEXFLAGS?=      -s
      CONVERT.md.tex?=${MD2TEX} ${MD2TEXFLAGS} -o $@ $<

      TEX2MD?=          pandoc
      TEX2MDFLAGS?=      -s
      CONVERT.tex.md?=${TEX2MD} ${TEX2MDFLAGS} -o $@ $<
```

This gives the following suffix rules.

```
64b  <MD to TeX 64b>≡ (61a)
      %.md: %.tex
      ${CONVERT.tex.md}
```

Sometimes we want to convert to HTML, we will use **pandoc** to convert both Markdown and LaTeX to HTML.

```
64c  <variables 59a>+≡ (58) <64a 64e>
      MD2HTML?=          pandoc
      MD2HTMLFLAGS?=      -s
      CONVERT.md.html?=${MD2HTML} ${MD2HTMLFLAGS} -o $@ $<

      TEX2HTML?=          pandoc
      TEX2HTMLFLAGS?=      -s
      CONVERT.tex.html?=${TEX2HTML} ${TEX2HTMLFLAGS} -o $@ $<
```

Then we can provide following two rules:

```
64d  <MD, TeX to HTML 64d>≡ (61a)
      define to_html_rule
      %.html: %.$(1)
      ${CONVERT.$(1).html}
      endef
      $(foreach suf,md tex,$(eval $(call to_html_rule,${suf})))
```

There are times when we want to convert out TeX-files to plain text, e.g., to count the words. To do this we simply use the **detex** program.

```
64e  <variables 59a>+≡ (58) <64c>
      TEX2TEXT?=          detex
      TEX2TEXTFLAGS?=
      CONVERT.tex.txt?=${TEX2TEXT} ${TEX2TEXTFLAGS} -o $@ $<
```

This gives us the following suffix rule.

$$\begin{array}{lcl}
 65 & \langle TeX \text{ to } text \rangle \equiv & (61a) \\
 & \quad \% .txt : \% .tex & \\
 & \quad \quad \$\{CONVERT.tex.txt\} & 
 \end{array}$$

## Part IV

# Literate programming

## Chapter 9

# noweb.mk

### 9.1 Introduction and usage

The `noweb.mk` include provides suffix rules for weaving and tangling (produce documentation and code, respectively). The framework uses the conventions of make [GNU16, Section 10.2].

The suffix rules of make works by taking a prerequisite with one suffix and applying the recipe to get a target with another suffix. This requires the stem of the filename to be identical. The framework can handle files of suffixes defined in `NOWEB_SUFFIXES`. However, we also provide `NOWEAVE.tex` to weave a TeX file (`.tex`) from a NOWEB file (`.nw`) and `NOTANGLE.suf` to tangle a `.suf` file from a NOWEB file (`.nw`). There is also a `NOWEAVE.pdf` to weave directly to PDF. This assumes that the file is independent, i.e., no special LaTeX preamble.

### 9.2 Implementation

The overall structure is the same as for other include files. We will cover the suffix rules for documentation first and then those for code.

```
67 <noweb.mk 67>≡
    ifndef NOWEB_MK
      NOWEB_MK = true

    <variables 68b>

    INCLUDE_MAKEFILES?= .
    MAKEFILES_DIR?=${INCLUDE_MAKEFILES}
    include ${MAKEFILES_DIR}/tex.mk

    <suffix rules for weaving documentation 68a>
    <suffix rules for tangling code 69b>
```

endif

### 9.2.1 Weaving documentation

We will use the `noweave` command to weave the documentation. We are interested in two cases:

1. when a source program should be converted to TeX to be included in a larger document, and
2. when a source program is independent and should be converted to PDF.

The order of the rules are important. To ensure make takes the ‘shortcut’ of the second case, we must specify that rule first.

68a     $\langle$ suffix rules for weaving documentation 68a $\rangle \equiv$  (67)

$\langle$ suffix rule for weaving to PDF 68d $\rangle$

$\langle$ suffix rule for weaving to TeX 68c $\rangle$

Now, for the first case, we let

68b     $\langle$ variables 68b $\rangle \equiv$  (67) 69a $\triangleright$

NOWEAVE.tex?=            noweave \${NOWEAVEFLAGS.tex} \$< > \$@

NOWEAVEFLAGS.tex?=    \${NOWEAVEFLAGS} -x -n -delay -t2

Now we need to specify all the suffixes to use and then construct suffix rules for all of them. Fortunately we can use the same recipe for all, so we only need to write one recipe for multiple targets. We will use a variable `NOWEB_SUFFIXES` to keep a list of supported suffixes. Since these suffixes only matter for tangling, we will set the variable in that section. For now, we only use it.

68c     $\langle$ suffix rule for weaving to TeX 68c $\rangle \equiv$  (68a)

%.tex: %.nw

      \${NOWEAVE.tex}

define def\_weave\_to\_tex

%.tex: %\$(1).nw

      \${NOWEAVE.tex}

endef

\$(foreach suf,\${NOWEB\_SUFFIXES},\$(eval \$(call def\_weave\_to\_tex,\${suf})))

To differentiate the second case from the first (in terms of suffix rules), we go from `.nw` directly to `.pdf`<sup>1</sup>.

68d     $\langle$ suffix rule for weaving to PDF 68d $\rangle \equiv$  (68a)

%.pdf: %.nw

---

<sup>1</sup>Note, however, that these pattern rules will never be used by make. The make algorithm performs a depth-first search, thus make will take the long way by first converting to TeX, then to PDF. We can determine which of these two pattern rules should be used by moving the inclusion of the `tex.mk` include file above.

```

    ${NOWEAVE.pdf}

define def_weave_to_pdf
%.pdf: %$(1).nw
    $$NOWEAVE.pdf}
endef

$(foreach suf,${NOWEB_SUFFIXES},$(eval $(call def_weave_to_pdf,${suf})))

```

What differs NOWEAVE.pdf from NOWEAVE.tex is the options to `noweave` and the compilation step (instead of having that separately).

69a    *<variables 68b>+≡* (67) <68b 70a>

```

NOWEAVE.pdf?= \
    noweave ${NOWEAVEFLAGS.pdf} $< > ${@:.pdf=.tex} && \
    latexmk -pdf ${@:.pdf=.tex}
NOWEAVEFLAGS.pdf?= \
    ${NOWEAVEFLAGS} -x -t2 \
    -option "shift,breakcode,longxref,longchunks"

```

## 9.2.2 Tangling code

We will now cover the rules for tangling the source code for different languages.

69b    *<suffix rules for tangling code 69b>≡* (67)  
       *<general tangling rules 71a>*  
       *<special rules for different languages 71e>*

We will first write some general pattern rules, then supply ways to adapt this rule to the different languages.

### Handling underscores in filenames

A critical design decision concerns how we reference chunk names in the Makefile rules. Many programming languages (particularly Python) use underscores in filenames, such as `module_name.py` or `attachment_cache.py`.

When these filenames appear in chunk definitions like *<module\_name.py (never defined)>*, LaTeX interprets the underscores as subscript commands during documentation generation (weaving), causing compilation errors. The error manifests as ‘Missing \$ inserted’ because LaTeX expects math mode for subscripts.

**The `[[...]]` notation solution** Noweb provides the `[[...]]` notation specifically to handle special characters in code references. When we write *<module\_name.py (never defined)>* in a `.nw` file, noweb automatically escapes all LaTeX special characters (`_`, `&`, `%`, etc.) when weaving documentation. The brackets tell noweb: ‘treat this as code, not LaTeX’.

**Why we use it in Makefile rules** Since our Makefile rules must match the chunk names used in `.nw` files, and we want all Python files to use `[[...]]` notation (to handle underscores), we must specify `-R"[[filename]]` in the `notangle` command. The double quotes protect the brackets from shell interpretation, and `notangle` then looks for a chunk named `[[filename]]`.

This standardization means:

1. All Python chunk definitions use `<filename.py (never defined)>`
2. All Makefile rules use `-R"[$(notdir $@)]"`
3. Underscores work without escaping
4. Consistent pattern across the project

**Alternative approaches rejected** We considered but rejected:

**Escaping underscores** Writing `\\_` in chunk names requires escaping everywhere the chunk is referenced, making the `.nw` file harder to read.

**Using hyphens** Chunk names like `module-name.py` avoid LaTeX issues but require Makefile renaming rules (`-name.py` to `_name.py`), adding complexity.

**No special characters** Restricting filenames to avoid underscores breaks Python conventions where `module_name` is idiomatic.

The `[[...]]` notation approach handles all special characters uniformly and keeps `.nw` files readable.

We will use `notangle(1)`. Note that we use the `noweb` `[[...]]` notation to quote the chunk name. This is critical for handling filenames with underscores (common in Python) or other LaTeX special characters. Without the brackets, chunk names like `<module_name.py (never defined)>` would cause LaTeX to interpret the underscore as a subscript command, breaking documentation generation. The `[[...]]` notation tells `noweb` to escape all special characters properly.

70a `<variables 68b>+=` (67) `<69a 70b>`  
`NOTANGLEFLAGS?=`  
`NOTANGLE?= notangle ${NOTANGLEFLAGS} -R"[$(notdir $@)]" $(filter %.nw,$^) | \`  
 `${CPIF} $@ && noroots $(filter %.nw,$^)`

We will also use the command `cpif(1)`. This command only updates the files if they have changed. We need this since many files may reside in the same NOWEB source file, but only some of them are updated. Without `cpif`, `make` would normally *update all files* if *any has changed* — which is clearly undesirable.

70b `<variables 68b>+=` (67) `<70a 71d>`  
`CPIF?= cpif`

However, since we use this variable, `cpif(1)` can be substituted for `tee(1)` in desirable situations.

**General pattern rules** There are two general pattern rules that we will add.

71a  $\langle \text{general tangling rules 71a} \rangle \equiv$  (69b)  
 $\langle \text{tangle source files with suffix 71c} \rangle$   
 $\langle \text{tangle source files without suffix 71b} \rangle$

In the first one, we will tangle a file with suffix `.suf` from the source file with suffix `.suf.nw` and in the second a source file with suffix `.nw`.

We can start with the second. In this rule, we have a file with a supported suffix `.suf` depend on the NOWEB source file with suffix `.nw`. Then we let the recipe be set by the variable `NOTANGLE.suf`, which is the convention followed by `make(1)` [GNU16, Sect. 10.2].

71b  $\langle \text{tangle source files without suffix 71b} \rangle \equiv$  (71a)  
`$(addprefix %, ${NOWEB_SUFFIXES}): %.nw`  
`${NOTANGLE$(suffix $@)}`

For the other case, we add rules using the suffix prefixed to `.nw`.

71c  $\langle \text{tangle source files with suffix 71c} \rangle \equiv$  (71a)  
`define with_suffix_target`  
`%(1): %(1).nw`  
`${NOTANGLE$(suffix $@)}`  
`endef`  
`$(foreach suf, ${NOWEB_SUFFIXES}, $(eval $(call with_suffix_target, ${suf})))`

However, this rule does not capture some of the things we want, e.g., we cannot tangle a header file `.h` from a `.cpp.nw` file. We must add these rules manually, which we do below.

**Rules for different languages** We will now cover specialized instances of the general pattern rules defined above. We will simply set the default variables.

71d  $\langle \text{variables 68b} \rangle + \equiv$  (67) <70b 73d>  
 $\langle \text{defaults for C and C++ 71f} \rangle$   
 $\langle \text{defaults for Haskell 72c} \rangle$   
 $\langle \text{defaults for Make 73a} \rangle$   
 $\langle \text{defaults for remaining 73b} \rangle$

As noted above, we need some special rules for the C and C++ header files, but no extra rules for any other language.

71e  $\langle \text{special rules for different languages 71e} \rangle \equiv$  (69b)  
 $\langle \text{rules for C and C++ 72a} \rangle$

For the languages of the C-family, we will use the `-L` option to get the line preprocessor-directive in the generated source — this will allow `gdb` and the compiler to point to lines in the NOWEB source file, and not to the generated file.

71f  $\langle \text{defaults for C and C++ 71f} \rangle \equiv$  (71d) 72b>  
`NOWEB_SUFFIXES+= .c .cc .cpp .cxx`  
`NOTANGLEFLAGS.c?= -L`



```

NOTANGLE.c?=          notangle ${NOTANGLEFLAGS.c} -R"[$(notdir $@)]" \
    $(filter %.nw,$^) | ${CPIF} $@ && noroots $(filter %.nw,$^)
NOTANGLEFLAGS.cc?=    ${NOTANGLEFLAGS.c}
NOTANGLE.cc?=         notangle ${NOTANGLEFLAGS.cc} -R"[$(notdir $@)]" \
    $(filter %.nw,$^) | ${CPIF} $@ && noroots $(filter %.nw,$^)
NOTANGLEFLAGS.cpp?=   ${NOTANGLEFLAGS.c}
NOTANGLE.cpp?=        notangle ${NOTANGLEFLAGS.cpp} -R"[$(notdir $@)]" \
    $(filter %.nw,$^) | ${CPIF} $@ && noroots $(filter %.nw,$^)
NOTANGLEFLAGS.cxx?=   ${NOTANGLEFLAGS.c}
NOTANGLE.cxx?=        notangle ${NOTANGLEFLAGS.cxx} -R"[$(notdir $@)]" \
    $(filter %.nw,$^) | ${CPIF} $@ && noroots $(filter %.nw,$^)

```

For C-family source code, we will assume that the header files (declarations) are written together with the definitions, so that we can extract both files from the same NOWEB source. However, for this we must add extra pattern rules.

72a     $\langle \text{rules for } C \text{ and } C++ \text{ 72a} \rangle \equiv$  (71e)

```

%.h: %.c.nw
    ${NOTANGLE.h}

%.hh: %.cc.nw
    ${NOTANGLE.hh}

%.hpp: %.cpp.nw
    ${NOTANGLE.hpp}

%.hxx: %.cxx.nw
    ${NOTANGLE.hxx}

```

Finally, we can define the variables used for tangling.

72b     $\langle \text{defaults for } C \text{ and } C++ \text{ 71f} \rangle + \equiv$  (71d)  $\triangleleft 71f$

```

NOWEB_SUFFIXES+=      .h .hh .hpp .hxx
NOTANGLEFLAGS.h?=     -L
NOTANGLE.h?=          notangle ${NOTANGLEFLAGS.h} -R"[$(notdir $@)]" \
    $(filter %.nw,$^) | ${CPIF} $@ && noroots $(filter %.nw,$^)
NOTANGLEFLAGS.hh?=    ${NOTANGLEFLAGS.h}
NOTANGLE.hh?=         notangle ${NOTANGLEFLAGS.hh} -R"[$(notdir $@)]" \
    $(filter %.nw,$^) | ${CPIF} $@ && noroots $(filter %.nw,$^)
NOTANGLEFLAGS.hpp?=   ${NOTANGLEFLAGS.h}
NOTANGLE.hpp?=        notangle ${NOTANGLEFLAGS.hpp} -R"[$(notdir $@)]" \
    $(filter %.nw,$^) | ${CPIF} $@ && noroots $(filter %.nw,$^)
NOTANGLEFLAGS.hxx?=   ${NOTANGLEFLAGS.h}
NOTANGLE.hxx?=        notangle ${NOTANGLEFLAGS.hxx} -R"[$(notdir $@)]" \
    $(filter %.nw,$^) | ${CPIF} $@ && noroots $(filter %.nw,$^)

```

The suffix rules for Haskell is similar to those for C and C++, due to the Glasgow Haskell Compiler (GHC) being very close to the C and C++ compilers.

72c     $\langle \text{defaults for Haskell 72c} \rangle \equiv$  (71d)

```

NOWEB_SUFFIXES+=      .hs
NOTANGLEFLAGS.hs?=-L
NOTANGLE.hs?=-notangle ${NOTANGLEFLAGS.hs} -R"[[$(notdir $@)]]" \
    $(filter %.nw,$^) | ${CPIF} $@ && noroots $(filter %.nw,$^)

```

We also note that we do not need any suffix rule for `.lhs` files, for the same reason as for the weaving, GHC automatically tangles Haskell’s native literate files (`.lhs`).

Make also requires slightly modified flags. We need `-t2` to expand spaces into tabs, because `make(1)` requires tabs for indentation, not spaces.

```

73a  <defaults for Make 73a>≡ (71d)
      NOWEB_SUFFIXES+=      .mk
      NOTANGLEFLAGS.mk?=-t2
      NOTANGLE.mk?=-notangle ${NOTANGLEFLAGS.mk} -R"[[$(notdir $@)]]" \
          $(filter %.nw,$^) > $@ && noroots $(filter %.nw,$^)

```

For Python, LaTeX, shell scripts and Go, there is no special processing needed, we simply use the flags we set in the beginning. We will now iterate through *all* suffixes, including `.mk` and `.cpp` etc. that we have already defined options for. Since we use the `?=` operator, those we have already defined will be ignored. This has the added benefit that one can simply add `.suf` to `NOWEB_SUFFIXES` and this code will automatically generate the default settings.

```

73b  <defaults for remaining 73b>≡ (71d) 73c>
      NOWEB_SUFFIXES+=      .py .sty .cls .sh .go

      define default_tangling
      NOTANGLEFLAGS$(1)?=
      NOTANGLE$(1)?=-notangle ${NOTANGLEFLAGS$(1)} -R"[[$(notdir $@)]]" \
          ${$(filter %.nw,$$^)} | ${CPIF} $$@ && noroots ${$(filter %.nw,$$^)}
      endef

      $(foreach suffix,${NOWEB_SUFFIXES},$(eval $(call default_tangling,${suffix})))

```

However, we’d like to add an extra post-processing step for Python: we’d like to run a Python formatter (such as Black) on the generated source code.

```

73c  <defaults for remaining 73b>+≡ (71d) <73b
      NOTANGLE.py+=          && ${NOWEB_PYCODEFMT}

```

We’ll add Black as the default Python code formatter.

```

73d  <variables 68b>+≡ (67) <71d
      NOWEB_PYCODEFMT?=-    black $@

```

# Chapter 10

## haskell.mk

### 10.1 Introduction, usage and implementation

This is by far the shortest include file in this collection. What we provide here is a reasonable default set-up for make when working with Haskell. In summary, we provide the following.

74a  $\langle \text{haskell.mk} \text{ 74a} \rangle \equiv$   
     $\langle \text{default variables 74b} \rangle$   
     $\langle \text{suffix rules for Haskell programs 74c} \rangle$

The Glasgow Haskell Compiler is functionally equivalent to the GNU C Compiler when compiling C programs. It can also handle the linking step, which means that we can simply use GHC for the linking step.

74b  $\langle \text{default variables 74b} \rangle \equiv$  (74a)  
    LD= ghc

And then we can provide the following suffix rule for compiling Haskell programs.

74c  $\langle \text{suffix rules for Haskell programs 74c} \rangle \equiv$  (74a)  
    .SUFFIXES: .hs .lhs  
    .hs.o .lhs.o:  
        ghc \${HSFLAGS} -c \$<

For weaving Haskell code, if the code is written using Haskell's native literate language, then that code is directly compilable as LaTeX code. So we need not do any weaving for .lhs files.

**Part V**

**Assessment**

# Chapter 11

## exam.mk

### 11.1 Introduction and usage

Many courses use exams as the tool for assessment. Usually the exam is repeated a few times during the year and over the years. This is quite repetitive, so we want to make it as easy as possible. This makefile, `<exam.mk (never defined)>`, will automate as much as possible using the `examgen` program [Bos16]. (It is recommended that you read the documentation of `examgen` before you continue, or at least run `examgen -h`.)

We assume that the exams will have the following structure. There is a main TeX file called `exam-uniqueID.tex`. If this is not the case, one will automatically be created using `./exam-template.tex` as the base. This file contains the code which uses the exam document class and, in particular, contains the following code:

```
\begin{questions}  
  \input{questions-ID.tex}  
\end{questions}
```

The file `questions-ID.tex` will be automatically generated by the exam generator. The prefixes of the filenames can be controlled using the following variables:

```
76a <variables 76a>≡ (77d) 76b>  
    EXAM_NAME?=      exam  
    EXAM_TEMPLATE?=  template  
    EXAM_QNAME?=     questions
```

With this structure, we only need to keep track of the unique identifiers, ‘ID’ in the example. We will use `EXAM_IDS` as a space-separated list containing all IDs. (The default is a single ID, which is today’s date.)

```
76b <variables 76a>+≡ (77d) <76a 77a>  
    EXAM_IDS?=       $(shell date +%y%m%d)
```

Now let us proceed to the contents, i.e., `questions-ID.tex`. The intended learning outcomes (ILOs) of a course rarely changes, so usually several exams share the same set of ILOs. This means that we would like to generate exams with the same parameters for several exams, e.g., the same databases and the same tags. These parameters are given to `examgen` as a set of tags, i.e., a space-separated list.

77a `<variables 76a>+≡` (77d) `<76b 77b>`  
`EXAM_TAGS?= ILO1 ILO2 ... ILOn`

`examgen` also needs to get the questions from somewhere, we will use `EXAM_DBS` as a space-separated list of question database files. The default value is all previous exams<sup>1</sup>.

77b `<variables 76a>+≡` (77d) `<77a 77c>`  
`EXAM_DBS?= $(foreach id,${EXAM_IDS},${EXAM_QNAME}-${id}.tex)`

Sometimes we might want a different set of tags or databases per exam. E.g. we want to generate one exam per student, where each student has an individual set of ILOs to be assessed on. For this reason we allow `EXAM_TAGS-ID` to override the contents of `EXAM_TAGS` when dealing with ID.

We can also pass specific flags to the `examgen` program using the `EXAM_FLAGS` variable. We set the default value as follows.

77c `<variables 76a>+≡` (77d) `<77b`  
`EXAM_FLAGS?= -NCE`

Note that the flags can be target-specific too, i.e., by setting `EXAM_FLAGS-ID`.

We conclude with a usage example.

**Example 2.** This will generate two exams: `exam-161014.pdf` and `exam-dbosk.pdf`.

The first will be generated from the `questions.tex` database with the complete tag set. The second will be generated from the same database, but only using the tag ‘ILOn’.

`EXAM_IDS= 161014 dbosk`

`EXAM_TAGS= ILO1 ILO2 ... ILOn`

`EXAM_DBS= questions.tex`

`EXAM_TAGS-dbosk= ILOn`

## 11.2 Implementation

We want to create a makefile `<exam.mk 77d>` (never defined) for inclusion. The file will have the following outline:

77d `<exam.mk 77d>≡`  
`<variables 76a>`

<sup>1</sup>This also includes all future exams, but `examgen` will ignore those since they do not yet exist.

$\langle$ generate targets for exam TeX files 78a $\rangle$   
 $\langle$ generate targets for exam PDF files 78c $\rangle$   
 $\langle$ generate targets for questions 79d $\rangle$

As suggested above, each exam `exam-ID.pdf` depends on at least two files: `exam-ID.tex` and `questions-ID.tex`. First, we will describe how to generate `exam-ID.tex` from a template found in `./exam-template.tex` ( $\langle$ generate targets for exam TeX files 78a $\rangle$ ). Then we will describe how we automatically generate these targets by iterating over the list in `EXAM_IDS` ( $\langle$ generate targets for exam PDF files 78c $\rangle$ ). Finally, we will describe how we generate the questions.

### 11.2.1 Generating targets for exam TeX files

As mentioned in Section 11.1, we will generate the file `exam-ID.tex` based on `exam-template.tex` if it doesn't exist. The `exam` prefix was controlled by `EXAM_NAME` and the template suffix controlled by `EXAM_TEMPLATE`. Thus we get this target structure:

78a  $\langle$ generate targets for exam TeX files 78a $\rangle \equiv$  (77d)

```

define exam_tex_files
  ${EXAM_NAME}-${(1)}.tex:
    ${CAT} ${EXAM_NAME}-${EXAM_TEMPLATE}.tex | \
       $\langle$ sed substitutions for exam with id in (1) 78b $\rangle$  \
    > $$@
endef
$(foreach id,${EXAM_IDS},$(eval $(call exam_tex_files,${id})))

```

Note that we don't have any dependency. Normally, it would be natural to add the template `exam-template.tex` as a dependency, however, we don't want to update old exams just because we update the template — that should only affect future exams.

### 11.2.2 Substituting fields from template

To generate the exam file from the template, we perform the following substitutions:

78b  $\langle$ sed substitutions for exam with id in (1) 78b $\rangle \equiv$  (78a)

```

${SED} \
-e "s/<EXAM_DATE>/${EXAM_DATE}-${(1)}/g" \
-e "s/<EXAM_ID>/${(1)}/g" \
-e "s/<EXAM_QNAME>/${EXAM_QNAME}/g"

```

### 11.2.3 Generating targets for exam PDFs

We will not provide any recipe for compiling the PDFs, that is left for the user or the use of `tex.mk`. What we will do is the following:

78c  $\langle$ generate targets for exam PDF files 78c $\rangle \equiv$  (77d)

$\langle$ define target-specific variables 79a $\rangle$   
 $\langle$ define callable exam definition 79b $\rangle$   
 $\langle$ call the exam definition for each ID 79c $\rangle$

We want the possibility of overriding EXAM\_NAME and EXAM\_QNAME for certain targets. We let the user set them, but if unset we set them to the default values.

79a  $\langle$ define target-specific variables 79a $\rangle \equiv$  (78c)

```

define target_variables
EXAM_NAME-$(1)?=    ${EXAM_NAME}
EXAM_QNAME-$(1)?=   ${EXAM_QNAME}
endef
$(foreach id,${EXAM_IDS},$(eval $(call target_variables,${id})))

```

We do the same for the actual targets. As stated above, we only set the dependencies and leave the recipe to the user (or tex.mk).

79b  $\langle$ define callable exam definition 79b $\rangle \equiv$  (78c)

```

define exam_target
${EXAM_NAME-$(1)}-$(1).pdf: ${EXAM_NAME-$(1)}-$(1).tex
${EXAM_NAME-$(1)}-$(1).pdf: ${EXAM_QNAME-$(1)}-$(1).tex
endef

```

Now we call the above variable and ask make(1) to evaluate it as code.

79c  $\langle$ call the exam definition for each ID 79c $\rangle \equiv$  (78c)

```

$(foreach id,${EXAM_IDS},$(eval $(call exam_target,${id})))

```

## 11.2.4 Generating targets for exam questions

We also said above that the file questions-ID.tex will automatically be generated by examgen. We will now provide the target that accomplishes just that. (Since the exam depends on this file, we will automatically generate the questions when we try to make the exam — if it does not already exist.) The structure of the code will be similar as for the exam.

79d  $\langle$ generate targets for questions 79d $\rangle \equiv$  (77d)

$\langle$ define target-specific questions variables 79e $\rangle$   
 $\langle$ define the questions target 80 $\rangle$

The ID-specific variables are defined analogously to those for the exam. The variables that are relevant to make specific are the following.

79e  $\langle$ define target-specific questions variables 79e $\rangle \equiv$  (79d)

```

define questions_variables
EXAM_TAGS-$(1)?=    ${EXAM_TAGS}
EXAM_DBS-$(1)?=     ${EXAM_DBS}
EXAM_FLAGS-$(1)?=   ${EXAM_FLAGS}
endef
$(foreach id,${EXAM_IDS},$(eval $(call questions_variables,${id})))

```



Finally, we can define target as follows. The target file ‘questions-ID.tex’ depends on the questions databases to exist. Then the recipe simply runs **examgen** with the set parameters.

```

80  <define the questions target 80>≡ (79d)
    define questions_target
    .PRECIOUS: ${EXAM_QNAME-${1}}-${1}.tex
    ${EXAM_QNAME-${1}}-${1}.tex:
        examgen ${EXAM_FLAGS-${1}} -d ${EXAM_DBS-${1}} -t ${EXAM_TAGS-${1}} > $$@
    endef
    $(foreach id,${EXAM_IDS},$(eval $(call questions_target,${id})))

```

# Chapter 12

## results.mk

### 12.1 Introduction and usage

The problem case is the following. We have a Moodle system where we do grading and everything related to a course, i.e., we have individual assignments. Then we must report the grades to a national database. The entries in this database is according to parts set in the course syllabus, each part can contain one or more assignments. This makefile uses the data that can be extracted from Moodle and some settings, then it converts the data to a form which is reportable to the student office, where the report is *manually* entered into the national database.

Since the data must be manually entered into the database, we require that the reports we send are not overlapping. E.g. if we report all grades by the end of a course, but some students complete their assignments late and are graded after the first report, then we must generate a second report which only contains the new results.

The input is a file which is exported from Moodle (a tab-separated CSV-file). The output is also a tab-separated CSV-file, reflecting the current state of what has been reported to the national database. The output could thus simply be a copy of the input, it will be used for comparison the next time we generate a report. Finally, we will output a temporary file, the report to be sent to the student office for registration. For this we will use three variables that can be set on the command-line:

```
81 <variables 81>≡ (82a) 82b>
    in?=          new.csv
    out?=         reported.csv
    report?=      report.pdf
```

The structure will be that of a makefile used for inclusion in a main Makefile. The structure is thus similar to most makefiles, we first need `<variables 81>`, then `<targets 82d>`. Since this will be a file to include, we do not want to include the

same contents twice, in any form of accidental recursive inclusion, so we use a C-like construction.

```
82a  <results.mk 82a>≡
      ifndef MIUN_RESULTS_MK
      MIUN_RESULTS_MK=true

      <variables 81>
      <targets 82d>

      INCLUDE_MAKEFILES?= .
      include ${INCLUDE_MAKEFILES}/miun.depend.mk

      endif
```

So to use this file, simply input it in your Makefile by adding the line `include results.mk` at the end of the file, in the same fashion as the inclusion of `miun.depend.mk` above.

For the purpose of reporting the results, we need to provide some identifiers. Usually this comes in the form of a course identifier. We also need to know where to send the results, so we can automate as much as possible. We will dedicate two variables for this, which can be set in a Makefile.

```
82b  <variables 81>+≡ (82a) <81 82c>
      RESULTS_COURSE?=   course identifier
      RESULTS_EMAIL?=    iksexp@miun.se
```

These variables are later used to form the command for sending the results to the student office. By default we use Mutt<sup>1</sup>.

```
82c  <variables 81>+≡ (82a) <82b 83>
      RESULTS_MAILER?=  mutt -s "resultat ${RESULTS_COURSE}" -a ${report} - ${RESULTS_EMAIL}
```

We provide a target `report` which processes the input, generates the new report and emails it to the designated address above.

```
82d  <targets 82d>≡ (82a) 82e>
      .PHONY: report
      report:
      <report recipe 87e>
```

Finally, we also provide a way to clean up all temporary files. We provide a target `clean-results` which we add as a dependency to the `clean` target, which is left to the user to use for whatever other cleaning is specified in the Makefile.

```
82e  <targets 82d>+≡ (82a) <82d 84a>
      .PHONY: clean clean-results
      clean: clean-results
```

---

<sup>1</sup>We could also use Thunderbird by setting `RESULTS_MAILER?=thunderbird -compose "to=${RESULTS_EMAIL},subject='resultat ${RESULTS_COURSE}',attachment='file://${report}'"`.

```
clean-results:
    <clean recipe 84b>
```

We will populate the <clean recipe 84b> as we go.

### 12.1.1 Portability

To improve the portability of the code, we use the following variables instead of the respective commands directly:

```
83  <variables 81>+≡ (82a) <82c 84f>
    LOCALC?=  localc -norestore
    RM?=      /bin/rm -Rf
    MV?=      /bin/mv
    DIFF?=    diff
    JOIN?=    join
    CUT?=     cut
    SORT?=    sort
    HEAD?=    head
    TAIL?=    tail
    SED?=     sed
    GREP?=    grep
    CAT?=     cat
    CP?=      cp -R
    PAGER?=   less
    PASTE?=   paste
    LN?=      ln
```

There is currently an unknown bug causing the join command to not work with tabs, although that exact code has worked previously, so the resulting report file will be *space separated*.

## 12.2 Processing Moodle's output

This section covers the technical details of how to process the data exported from Moodle. We have the input file, given as `${in}`, then we want the a report of changes to send to the student office (Section 12.2.2). There are different identifiers used in the national database and in Moodle. So we need to extract the identifiers in Moodle and convert to those in the national database (Section 12.2.3). Then we can send the report and update our local representation of what is reported to the national database, i.e., `${out}`.

### 12.2.1 Transforming Moodle's output

The first thing we need to do is to transform Moodle's output. The output format varies a lot, it changes with the mood of the system administrator. So the code in this section changes the most.

We will now create a temporary file `${out}.diff` based on `${in}`.

```
84a  <targets 82d>+≡ (82a) <82e 84g>
      ${out}.new: ${in}
      <new recipe 84c>
```

This means we should also add `${out}.diff` to the recipe of clean.

```
84b  <clean recipe 84b>≡ (82e) 86f>
      ${RM} ${out}.new
```

We are now going to process the data, we will do this by piping the data through a series of commands. The columns we are interested in are 1–3 and 6 to the end.

```
84c  <new recipe 84c>≡ (84a) 84d>
      ${CUT} -f 1-3,6- ${in} | \
```

For some reason the students' usernames are appended to their lastnames — in addition to having a separate column for usernames. Obviously we want to filter this away.

```
84d  <new recipe 84c>+≡ (84a) <84c 84e>
      ${SED} "s/ (\([a-z]\{4\}[0-9]\{4\}\))/" \
```

Some of the data in Moodle are quite long, so we would like to do some rewrites. For this purpose we will add a list of regular expressions that will be applied. We store this list as a space separated list of regular expressions in `${RESULTS_REWRITES}`. This means that we also must avoid spaces in the regular expressions, thus the first thing we do is to remove all spaces in the data.

```
84e  <new recipe 84c>+≡ (84a) <84d
      $(if ${RESULTS_REWRITES},| ${SED} "s/ //g", ) \
      $(foreach regex,${RESULTS_REWRITES},| ${SED} ${regex}) \
      > $@
```

We let the default rewrites be

```
84f  <variables 81>+≡ (82a) <83 85b>
      RESULTS_REWRITES+= "s/Godkänd(G)/G/g" "s/Underkänd(U)/U/g"
      RESULTS_REWRITES+= "s/Komplettering(Fx)/Fx/g"
      RESULTS_REWRITES+= "s/\"//g"
```

## 12.2.2 Extracting the changes

Now we want to find what has changed since the last time we exported the grades. For this we will create a file `${out}.diff` which contains only the changed rows.

```
84g  <targets 82d>+≡ (82a) <84a 85e>
      ${out}.diff: ${out}.new
      <diff recipe 85a>
```



Now we want to extract the usernames and get the identifiers from the national database. We simply extract the list of usernames (the third column in the data) and pipe it to a pager.

```
86a  <identifier recipe 86a>≡ (85g) 86b>
      @echo "-- userids showed in ${PAGER} --"
      ${CAT} $< | ${CUT} -f 3 | ${PAGER}
```

Now we let the user paste the list of both identifiers.

```
86b  <identifier recipe 86a>+≡ (85g) <86a
      @echo "-- paste username <tab> personnummer, end with C-d on a blank line (EOF) --"
      ${CAT} > $@
```

## 12.2.4 Generating the report

Now we have the changes in `${out}.diff` and a mapping from usernames to civic registration numbers in `${out}.diff.id`. To create the report, we only have to join these files and convert the result to PDF format.

To convert a CSV-file to PDF we will use LibreOffice and one of make's suffix rules.

```
86c  <targets 82d>+≡ (82a) <85g 86d>
      .SUFFIXES: .csv .pdf
      .csv.pdf:
      ${LOCALC} $<
```

Now we can add a target using this conversion.

```
86d  <targets 82d>+≡ (82a) <86c 86e>
      ${report:.pdf=.pdf}: ${report:.pdf=.csv}
```

The above target lets us create a PDF-formatted report from a CSV-file, so now we have to create that CSV-file using `${out}.diff` and `${out}.diff.id`. We also need the table headers from `${in}`.

```
86e  <targets 82d>+≡ (82a) <86d 87c>
      ${report:.pdf=.csv}: ${in} ${out}.diff ${out}.diff.id
      <report.csv recipe 86g>
```

Since the target for `report.csv` will automatically generate `${out}.diff` and `${out}.diff.id` we would better add them to the clean recipe in addition to the `${report:.pdf=.csv}` file.

```
86f  <clean recipe 84b>+≡ (82e) <84b 87d>
      ${RM} ${out}.diff ${out}.diff.id
      ${RM} ${report:.pdf=.csv}
```

Now we want the header back, so we can get it properly formatted from `${out}.new`. However, we do not want all the excess columns for the grades: we only need one column with a summary.

```
86g  <report.csv recipe 86g>≡ (86e) 87a>
      ${HEAD} -n 1 ${out}.new | \
      ${CUT} -f -${RESULTS_COLUMNS} > $@
```

Next, we simply join the two tables on the username column and sort the list on the column of the family name. We want to cut the excess columns here as well, the number of columns is controlled by `RESULTS_COLUMNS`.

```
87a  <report.csv recipe 86g>+= (86e) <86g
      ${JOIN} -1 1 -2 3 ${out}.diff.id ${out}.diff | ${CUT} -d " " -f 2- | \
      ${SORT} -k 2 | ${CUT} -d " " -f -${RESULTS_COLUMNS} » $@
```

By default we let the default number of columns be four, i.e., first and last name, civic identification number and finally one grade.

```
87b  <variables 81>+= (82a) <85b
      RESULTS_COLUMNS?= 4
```

## 12.3 Sending and storing the results

Now we will do the actual reporting. As stated in Section 12.1 we have a target `report` for this purpose.

```
87c  <targets 82d>+= (82a) <86e
      .PHONY: report
      report: ${report} ${in}
```

Since this will trigger the creation of `report` we must add it to the clean recipe.

```
87d  <clean recipe 84b>+= (82e) <86f
      ${RM} ${report}
```

If there are no new results, then we do not want to send any report. The first thing we do is thus to check for new results. If there are none, we will say so to the user.

```
87e  <report recipe 87e>+= (82d) 87f>
      if [ ! -s ${out}.diff ]; then \
      echo "No new results to report" >&2; \
```

Otherwise, if there are new results, we will output them using the pager and then send them using the email program. If the emailing succeeds, then we want to store the results, but not if it fails (hence the conditional).

```
87f  <report recipe 87e>+= (82d) <87e
      else \
      ${PAGER} ${report}; \
      ${RESULTS_MAILER} && \
      ${MV} ${out}.new ${out}; \
      fi
```



Part VI

Appendices

## Appendix A

# Dockerfile: an execution environment

We need an environment to run in. The makefiles should work on most systems without any problem. However, here we will create a Docker Image with an environment that is guaranteed to work and can be used for, e.g., continuous integration. This image is built using a *Dockerfile* (never defined).

The *Dockerfile* (never defined) will have the following structure:

89a *Dockerfile* 89a)≡  
    *base image to use* 89b)  
    *image info* 89c)  
    *install packages* 90a)  
    *copy makefiles* 90c)

We will base the environment on Ubuntu.

89b *base image to use* 89b)≡ (89a)  
    FROM ubuntu:latest  
    ENV DEBIAN\_FRONTEND noninteractive

We will provide some basic information about who maintains this image and where more information can be found.

89c *image info* 89c)≡ (89a)  
    MAINTAINER Daniel Bosk <dbosk@kth.se>  
    LABEL se.bosk.daniel.makefiles.version="\$Id\$"  
    LABEL se.bosk.daniel.makefiles.url="https://github.com/dbosk/makefiles"

We install the basic packages needed. To make as small a Docker image as we can, we will do as much as possible with as few RUN commands as possible<sup>1</sup>.

---

<sup>1</sup>See <https://hackernoon.com/tips-to-reduce-docker-image-sizes-876095da3b34> for a discussion on how to reduce the size of images.

We will put TeXLive in its own layer, to avoid redoing that whenever we update this image.

90a       $\langle install\ packages\ 90a \rangle \equiv$  (89a) 90b  $\triangleright$

```
RUN apt-get update -y && \
    apt-get install -no-install-recommends -y \
        texlive-* \
        latexmk \
        xindy \
        biber \
        bibtool \
    && \
    apt-get purge -fy *-doc && \
    apt-get autoremove -y && \
    rm -Rf /var/lib/apt/lists/* && \
    rm -Rf /usr/share/doc && \
    rm -Rf /usr/share/man
```

We will now concentrate smaller packages to a separate RUN command.

90b       $\langle install\ packages\ 90a \rangle + \equiv$  (89a)  $\triangleleft$  90a

```
RUN apt-get update -y && \
    apt-get install -no-install-recommends -y \
        curl \
        git \
        gnuplot \
        imagemagick \
        inkscape \
        make \
        noweb \
        pandoc \
        python3-matplotlib \
        python3-numpy \
        python3-pygments \
        python3-scipy \
        python3-pip \
        qrencode \
        unzip \
    && \
    apt-get purge -fy *-doc && \
    rm -Rf /var/lib/apt/lists/* && \
    rm -Rf /usr/share/doc && \
    rm -Rf /usr/share/man
```

We copy every makefile into the image's directory `/usr/local/include` to make them generally available inside the environment.

90c       $\langle copy\ makefiles\ 90c \rangle \equiv$  (89a)

```
COPY doc.mk /usr/local/include
```

```
COPY exam.mk /usr/local/include
COPY haskell.mk /usr/local/include
COPY latexmkrc /usr/local/include
COPY noweb.mk /usr/local/include
COPY pkg.mk /usr/local/include
COPY portability.mk /usr/local/include
COPY pub.mk /usr/local/include
COPY results.mk /usr/local/include
COPY subdir.mk /usr/local/include
COPY tex.mk /usr/local/include
COPY transform.mk /usr/local/include
```

# Bibliography

- [Aye] Andrew Ayer. *git-crypt: transparent file encryption in git*. Accessed on 23rd October 2016. URL: <https://www.agwa.name/projects/git-crypt/>.
- [Bos16] Daniel Bosk. *examgen: An exam generator*. v3.1. 2016. URL: <https://github.com/dbosk/examgen/releases/tag/v3.1>.
- [CP11] Brian Carpenter and Craig Partridge. *Recommendations of a committee on RFC citation issues*. Internet draft. Internet Engineering Task Force, Feb. 2011. URL: <https://tools.ietf.org/html/draft-carpenter-rfc-citation-recs-01#section-5.2>.
- [GNU16] GNU Project. *GNU Make Manual*. Free Software Foundation. May 2016. URL: <https://www.gnu.org/software/make/manual/>.
- [Gre16] Enrico Gregorio. *The package imakeidx*. v1.3d. May 2016.
- [Hir15] Philip Hirschhorn. *Using the exam document class*. 2.5. May 2015.
- [KC16] Philip Kime and François Charette. *biber: A backend bibliography processor for biblatex*. v2.5. May 2016.
- [Leh+16] Philipp Lehman, Philip Kime, Audrey Boruvka and Joseph Wright. *The biblatex package*. v3.4. May 2016.
- [Nie16] Clemens Niederberger. *ACRO*. v2.6a. Aug. 2016.
- [Tal16] Nicola L.C. Talbot. *User manual for glossaries.sty v4.25*. June 2016.
- [TWM15] Till Tantau, Joseph Wright and Vedran Miletic. *The Beamer class*. 3.36. Mar. 2015.