Make-ing Life Easy:
A General Makefile Framework

Daniel Bosk

School of Computer Science and Communication
KTH Royal Institute of Technology, Stockholm

Department of Information and Communication Systems
Mid Sweden University, Sundsvall

18th January 2018

Contents

Part I

General building blocks

Chapter 1

portability.mk

1.1 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.

5a ⟨*portability.mk* 5a⟩≡

```
ifndef PORTABILITY_MK
PORTABILITY_MK=true

⟨system-specific configuration 5b⟩

⟨standard unix commands 6a⟩
⟨networking commands 7d⟩
⟨compressed files and archives 8b⟩

endif
```

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

5b ⟨*system-specific configuration* 5b⟩≡ (5a)

```
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.

1.2 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.

6a ⟨*standard unix commands* 6a⟩≡ (5a)

```
⟨file system commands 6b⟩
⟨printing file contents 6d⟩
⟨opening files depending on file type 6c⟩
⟨filtering and transforming file contents 7a⟩
⟨statistics on file contents 7c⟩
```

1.2.1 File system commands

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

6b ⟨*file system commands* 6b⟩≡ (6a)

```
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.

1.2.2 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.

6c ⟨*opening files depending on file type* 6c⟩≡ (6a)

```
XDGOPEN?= xdg-open
```

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

6d ⟨*printing file contents* 6d⟩≡ (6a)

```
CAT?=     cat
```

1.2.3 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.

7a ⟨*filtering and transforming file contents* 7a⟩≡ (6a) 7b▷
```
ifeq (${MAKE},gmake)
SED?=     gsed
SEDex?=   gsed -E
else
SED?=     sed
SEDex?=   sed -E
endif
```
Similarly, we let

7b ⟨*filtering and transforming file contents* 7a⟩+≡ (6a) ◁7a
```
ifeq (${MAKE},gmake)
GREP=     ggrep
GREPex=   ggrep -E
else
GREP=     grep
GREPex=   grep -E
endif
```

1.2.4 Statistics

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

7c ⟨*statistics on file contents* 7c⟩≡ (6a)
```
WC?=      wc
WCw?=     wc -w
```

1.3 Networking commands

We also need some network related commands.

7d ⟨*networking commands* 7d⟩≡ (5a)
  ⟨*fetching files* 7e⟩
  ⟨*remote execution* 8a⟩

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

7e ⟨*fetching files* 7e⟩≡ (7d)
```
CURL?=    curl
SFTP?=    sftp
SCP?=     scp -r
```

We also need commands for remote execution.

8a ⟨*remote execution* 8a⟩≡ (7d)
```
SSH?=      ssh
```

1.4 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.

8b ⟨*compressed files and archives* 8b⟩≡ (5a)
⟨*variables for compression programs* 9b⟩

⟨*variables for archive programs* 9e⟩
⟨*general pattern rule for archiving* 9d⟩
⟨*function to generate extraction targets* 10c⟩

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).

1.4.1 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[1].

8c ⟨*general pattern rule for compression* 8c⟩≡
```
define compress
%$(1): %
	${COMPRESS$(1)}
endef
```

[1] Note that the pattern rules in the code blocks ⟨*general pattern rule for compression* 8c⟩ and ⟨*general pattern rule for uncompression* 9a⟩ are not included in ⟨*compressed files and archives* 8b⟩ above, and thus not enabled by default. This is due to causing circular dependencies.

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

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

9a ⟨*general pattern rule for uncompression* 9a⟩≡

```
  define uncompress
  %: %$(1)
    ${UNCOMPRESS$(1)}
  endef
  $(foreach suf,${UNCOMPRESS_SUFFIXES},$(eval $(call uncompress,${suf})))
```

We note that due to 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).

9b ⟨*variables for compression programs* 9b⟩≡ (8b) 9c ▷

```
  UNCOMPRESS_SUFFIXES+= .gz .z
  GUNZIP?=              gunzip
  UNCOMPRESS.gz?=       ${GUNZIP} $<
  UNCOMPRESS.z?=        ${UNCOMPRESS.gz}
```

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

9c ⟨*variables for compression programs* 9b⟩+≡ (8b) ◁ 9b

```
  COMPRESS_SUFFIXES+=   .gz
  GZIP?=                gzip
  COMPRESS.gz?=         ${GZIP} $<
```

1.4.2 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.

9d ⟨*general pattern rule for archiving* 9d⟩≡ (8b)

```
  (%):
    ${ARCHIVE$(suffix $@)}
```

Unlike for the compression targets above (Section 1.4.1), 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.

9e ⟨*variables for archive programs* 9e⟩≡ (8b) 10a ▷

```
  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.

10a ⟨*variables for archive programs* 9e⟩+≡ (8b) ◁9e 10b▷
```
TAR?=          tar -u
PAX?=          pax -wzLx ustar
ARCHIVE.tar?= ${TAR} -f $@ $%
```
We can also create zip(1) archives.

10b ⟨*variables for archive programs* 9e⟩+≡ (8b) ◁10a 10d▷
```
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.

10c ⟨*function to generate extraction targets* 10c⟩≡ (8b)
```
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 1.4.1), 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 that 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).

10d ⟨*variables for archive programs* 9e⟩+≡ (8b) ◁10b 10e▷
```
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.

10e ⟨*variables for archive programs* 9e⟩+≡ (8b) ◁10d
```
ifeq ($(shell uname),Darwin)
UNZIP?=        unzip
else
```

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

Chapter 2

subdir.mk

2.1 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 allow parallel execution).

By default, for any goals passed on the command line we will add the directories in `SUBDIR` as prerequisites. This behaviour can be overridden by setting `SUBDIR_ALL` to anything different from `yes`. Like this we can add the subdirectories in `SUBDIR` as prerequisites manually to only a subset of desired targets.

2.2 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.

12 ⟨*subdir.mk* 12⟩≡
```
ifndef SUBDIR_MK
SUBDIR_MK=true

INCLUDE_MAKEFILES?=.

ifdef SUBDIR
⟨let the recipe for each subdirectory recurse into it 13a⟩
endif

SUBDIR_ALL?=yes
```

```
ifeq (${SUBDIR_ALL},yes)
⟨add subdirectories as prerequisites for the goals 13b⟩
endif

endif
```

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

13a ⟨*let the recipe for each subdirectory recurse into it* 13a⟩≡ (12)

```
.PHONY: ${SUBDIR}
${SUBDIR}:
	${MAKE} -C $@ -I ${INCLUDE_MAKEFILES} ${MAKECMDGOALS}
```

We also want to give the sub-make access to our `INCLUDE_MAKEFILES`, hence the `-I` option. This is mostly due to (backwards) compatibility with the MIUN versions (see Appendix A) of the makefiles, which pre-dates the `INCLUDE_MAKEFILES` construction.

To ensure these recipes are run we need to ensure that they are prerequisites to the goals. This also means that if no goals are given on the command-line, then we should use the default goal.

13b ⟨*add subdirectories as prerequisites for the goals* 13b⟩≡ (12)

```
ifneq (${MAKECMDGOALS},)
.PHONY: ${MAKECMDGOALS}
${MAKECMDGOALS}: ${SUBDIR}
else
${.DEFAULT_GOAL}: ${SUBDIR}
endif
```

We note that this will cause the default goal of each subdirectory to be built, not the same goal which is the default goal in the root.

Part II

Packaging and publishing

Chapter 3

pkg.mk

3.1 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.

15a ⟨*variables* 15a⟩≡ (17a) 15b ▷

```
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.

15b ⟨*variables* 15a⟩+≡ (17a) ◁ 15a 15c ▷

```
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.

15c ⟨*variables* 15a⟩+≡ (17a) ◁ 15b 15d ▷

```
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.

15d ⟨*variables* 15a⟩+≡ (17a) ◁ 15c 16a ▷

```
IGNORE_FILES?=      \(\.svn\|\.git\|CVS\)
PKG_IGNORE?=        ${IGNORE_FILES}
```

The installation path is controlled by the following variables.

16a ⟨*variables* 15a⟩+≡ (17a) ◁15d 16b▷
```
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`.

16b ⟨*variables* 15a⟩+≡ (17a) ◁16a 16c▷
```
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.

16c ⟨*variables* 15a⟩+≡ (17a) ◁16b 16d▷
```
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)?=        ${PKG_TARBALL}
PKG_TARBALL_FILES-$(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.

16d ⟨*variables* 15a⟩+≡ (17a) ◁16c 16e▷
```
$(foreach pkg,${PKG_PACKAGES},$(eval $(call variables,${pkg})))
```

3.1.1 Portability

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

16e ⟨*variables* 15a⟩+≡ (17a) ◁16d
```
ifneq (${MAKE},gmake)
INSTALL?=     ${SUDO} install -Dp
else
INSTALL?=     ${SUDO} install -CSp
endif
```

3.2 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.

17a ⟨*pkg.mk* 17a⟩≡
```
ifndef PACKAGE_MK
PACKAGE_MK=true

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

⟨variables 15a⟩
⟨an all-like target 17b⟩
⟨targets for packaging 17c⟩
⟨targets for cleaning 18a⟩
⟨targets for installation 18d⟩

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.

17b ⟨*an all-like target* 17b⟩≡ (17a)
```
.PHONY: package
package: $(foreach pkg,${PKG_PACKAGES},${PKG_TARBALL-${pkg}})
```

3.2.1 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 1.4.2.

17c ⟨*targets for packaging* 17c⟩≡ (17a)
```
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})))
```

3.2.2 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.

18a ⟨*targets for cleaning* 18a⟩≡ (17a) 18b ▷
```
.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.

18b ⟨*targets for cleaning* 18a⟩+≡ (17a) ◁ 18a
```
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})))
```

3.2.3 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.

18c ⟨*installation process* 18c⟩≡ (18d)
```
.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.

18d ⟨*targets for installation* 18d⟩≡ (17a) 18e ▷
```
.PHONY: install
install: post-install
⟨installation process 18c⟩
```

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.

18e ⟨*targets for installation* 18d⟩+≡ (17a) ◁ 18d
```
define post-install
.PHONY: post-install-$(1)
post-install: post-install-$(1)
```

```
post-install-$(1): do-install-$(1)
endef
$(foreach pkg,${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 19⟩
endef
$(foreach pkg,${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)}
endef
$(foreach pkg,${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.

19 ⟨*default do-install recipe* 19⟩≡ (18e)

```
for f in ${PKG_INSTALL_FILES-$(1)}; do \
  [ -d "$$$$f" ] || ${INSTALL} -t ${PKG_PREFIX-$(1)}${PKG_INSTALL_DIR-$(1)}/ "$$$$f"; \
done
```

Chapter 4

pub.mk

4.1 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 to servers. We provide a general framework and then three different methods of publishing that can be plugged in.

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.

20a ⟨*variables* 20a⟩≡ (23f) 20b ▷

```
PUB_FILES?=
```

For convenience, we can also control files to ignore.

20b ⟨*variables* 20a⟩+≡ (23f) ◁ 20a 20c ▷

```
IGNORE_FILES?=      \(\.svn\|\.git\|CVS\)
PUB_IGNORE?=        ${IGNORE_FILES}
```

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

20c ⟨*variables* 20a⟩+≡ (23f) ◁ 20b 20d ▷

```
PUB_SERVER?=        localhost
```

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

20d ⟨*variables* 20a⟩+≡ (23f) ◁ 20c 20e ▷

```
PUB_DIR?=           ${PUBDIR}/${CATEGORY}
```

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

20e ⟨*variables* 20a⟩+≡ (23f) ◁ 20d 21a ▷

```
PUB_USER?=          ${USER}
PUB_GROUP?=         ${GROUP}
PUB_CHMOD?=         o+r
```

4.1.1 Publication methods

There are currently three methods for publication: `ssh`, `git`, and `at`. The default method is `ssh`.

21a ⟨*variables* 20a⟩+≡ (23f) ◁20e 21b▷
```
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`).

21b ⟨*variables* 20a⟩+≡ (23f) ◁21a 21c▷
```
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.

21c ⟨*variables* 20a⟩+≡ (23f) ◁21b 21d▷
```
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.

21d ⟨*variables* 20a⟩+≡ (23f) ◁21c 21e▷
```
PUB_BRANCH?=        master
```

4.1.2 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.

21e ⟨*variables* 20a⟩+≡ (23f) ◁21d 22a▷
```
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.

22a ⟨*variables* 20a⟩+≡ (23f) ◁21e 22b▷

```
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}
endef

$(foreach site,${PUB_SITES},$(eval $(call variables,${site})))
```

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

```
PUB_SITES=               main mirror1 mirror2
PUB_SERVER =             foo.bar
PUB_SERVER-mirror1 =     foo.bar.mirror1
PUB_SERVER-mirror2 =     foo.bar.mirror2
```

4.1.3 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.

22b ⟨*variables* 20a⟩+≡ (23f) ◁22a 23a▷

```
PUB_AUTOTAG?=       false
```

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

23a ⟨*variables* 20a⟩+≡ (23f) ◁22b 23b▷
```
PUB_VCS?=           git
```
The only thing needed more than this is any options that the user want to use.

23b ⟨*variables* 20a⟩+≡ (23f) ◁23a 23c▷
```
PUB_TAG_OPTS?=
```
The tag name is controlled with the following variable. The default value is today's date and the current time.

23c ⟨*variables* 20a⟩+≡ (23f) ◁23b 23d▷
```
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.

23d ⟨*variables* 20a⟩+≡ (23f) ◁23c 23e▷
```
PUB_AUTOCOMMIT?=    false
```
The command and options are similarly set with the following.

23e ⟨*variables* 20a⟩+≡ (23f) ◁23d 26c▷
```
PUB_COMMIT_OPTS?=   -av
```

4.2 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.

23f ⟨*pub.mk* 23f⟩≡
```
ifndef PUB_MK
PUB_MK=true

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

⟨variables 20a⟩
⟨target for publishing 24a⟩
⟨publication methods 24d⟩
⟨targets for automatic tagging and committing 28a⟩

endif
```
We will now cover the different parts below. The ⟨*variables* 20a⟩ block has been covered in the usage section, but the remaining are discussed below.

4.2.1 The general publication mechanism

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

24a ⟨*target for publishing* 24a⟩≡ (23f) 24b▷

```
.PHONY: publish
publish: $(foreach site,${PUB_SITES},publish-${site})
```

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

24b ⟨*target for publishing* 24a⟩+≡ (23f) ◁24a 24c▷

```
ifeq (${PUB_AUTOTAG},true)
publish: autotag
else ifeq (${PUB_AUTOCOMMIT},true)
publish: autocommit
endif
```

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

24c ⟨*target for publishing* 24a⟩+≡ (23f) ◁24b

```
define publish_target
.PHONY: publish-$(1)
publish-$(1): $(foreach file,${PUB_FILES-$(1)},${file})
  $$(call publish-${PUB_METHOD-$(1)},$(1))
endef

$(foreach site,${PUB_SITES},$(eval $(call publish_target,${site})))
```

4.2.2 Publication methods

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

24d ⟨*publication methods* 24d⟩≡ (23f)

⟨*helper functions* 25a⟩
⟨*ssh method* 25c⟩
⟨*at method* 26e⟩
⟨*git method* 27e⟩

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.

25a ⟨*helper functions* 25a⟩≡ (24d) 25b ▷
```
define chown
$(if ${PUB_GROUP-$(1)},\
  ${SSH} ${PUB_SERVER-$(1)}\
  ${CHOWN} ${PUB_USER-$(1)}:$(strip ${PUB_GROUP-$(1)}) ${PUB_DIR-$(1)};,)
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.

25b ⟨*helper functions* 25a⟩+≡ (24d) ◁ 25a
```
define chmod
$(if ${PUB_CHMOD-$(1)},\
  ${SSH} ${PUB_SERVER-$(1)}\
  ${CHMOD} ${PUB_CHMOD-$(1)} ${PUB_DIR-$(1)};,)
endef
```

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.

25c ⟨*ssh method* 25c⟩≡ (24d)
```
define publish-ssh
⟨create directory on server 25d⟩; \
⟨pack the files and pipe them to the server 25e⟩; \
$(call chown,$(1)) \
$(call chmod,$(1))
endef
```

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

25d ⟨*create directory on server* 25d⟩≡ (25c 26e)
```
${SSH} ${PUB_SERVER-$(1)} ${MKDIR} ${PUB_DIR-$(1)}
```

Next is the packing of the files.

25e ⟨*pack the files and pipe them to the server* 25e⟩≡ (25c)
```
⟨generate file list 26a⟩ | \
⟨pack the files 26b⟩ | \
⟨extract the files on the server 26d⟩
```

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.

26a ⟨*generate file list* 26a⟩≡ (25e 26e)
```
[ -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.

26b ⟨*pack the files* 26b⟩≡ (25e 26e)
```
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.

26c ⟨*variables* 20a⟩+≡ (23f) ◁23e 27a▷
```
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.

26d ⟨*extract the files on the server* 26d⟩≡ (25e)
```
${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 publish them in the destination at some predefined time.

26e ⟨*at method* 26e⟩≡ (24d)
```
define publish-at
⟨create directory on server 25d⟩; \
⟨create temporary directory 26f⟩; \
⟨generate file list 26a⟩ | \
⟨pack the files 26b⟩ | \
⟨extract the files in the temporary directory 27b⟩; \
⟨add at-job on the server 27c⟩
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.

26f ⟨*create temporary directory* 26f⟩≡ (26e)
```
TMPPUB=$$(${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.

27a ⟨*variables* 20a⟩+≡ (23f) ◁26c 27d▷
```
$(foreach site,${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.

27b ⟨*extract the files in the temporary directory* 27b⟩≡ (26e)
```
${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.

27c ⟨*add at-job on the server* 27c⟩≡ (26e)
```
${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.

27d ⟨*variables* 20a⟩+≡ (23f) ◁27a
```
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`.

27e ⟨*git method* 27e⟩≡ (24d)
```
define publish-git
git archive ${PUB_BRANCH-$(1)} ${PUB_FILES-$(1)} \
  | ${SSH} ${PUB_SERVER-$(1)} ${UNPAX} -s ",^,$(strip ${PUB_DIR-$(1)}),"; \
$(call chown,$(1)) \
$(call chmod,$(1))
endef
```

4.2.3 Automatically committing and tagging

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.

28a ⟨*targets for automatic tagging and committing* 28a⟩≡ (23f)

```
⟨commit and tag functions 28b⟩

.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.

28b ⟨*commit and tag functions* 28b⟩≡ (28a)

```
⟨autocommit for git, svn and cvs 28c⟩
⟨autotag for git and cvs 28d⟩
⟨autotag for svn 28e⟩
```

The commit functions are as expected for all three VCSs.

28c ⟨*autocommit for git, svn and cvs* 28c⟩≡ (28b)

```
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.

28d ⟨*autotag for git and cvs* 28d⟩≡ (28b)

```
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 is not as easy though. The outline is as follows.

28e ⟨*autotag for svn* 28e⟩≡ (28b)

```
⟨helper functions for svn tagging 29b⟩

define autotag-svn
⟨find the root of repo 29a⟩
⟨go to root and create tag 29c⟩
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.

29a ⟨*find the root of repo* 29a⟩≡ (28e)

```
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).

29b ⟨*helper functions for svn tagging* 29b⟩≡ (28e)

```
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
endef
```

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

29c ⟨*go to root and create tag* 29c⟩≡ (28e)

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

Chapter 5

transform.mk

5.1 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.

5.2 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.

30 ⟨*transform.mk* 30⟩≡

```
ifndef TRANSFORM_MK
TRANSFORM_MK=true

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

⟨variables 31a⟩
⟨suffix rule for transformations 31b⟩
⟨target generation for transformations 32a⟩
⟨suffix rules for camera-ready source 33b⟩
⟨suffix rules for encrypted files 34d⟩

endif
```

We will now explore now these are implemented.

5.3 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`.

31a ⟨*variables* 31a⟩≡ (30) 31e▷

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

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

31b ⟨*suffix rule for transformations* 31b⟩≡ (30)

```
⟨transformations 31c⟩
.SUFFIXES: ${TRANSFORM_SRC} ${TRANSFORM_DST}
$(foreach src,${TRANSFORM_SRC},$(foreach dst,${TRANSFORM_DST},${src}${dst})):
  ⟨transformation recipe 31d⟩
```

The ⟨*transformations* 31c⟩ 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.

31c ⟨*transformations* 31c⟩≡ (31b) 32b▷

```
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 ⟨*transformation recipe* 31d⟩. 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.

31d ⟨*transformation recipe* 31d⟩≡ (31b 32a)

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

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

31e ⟨*variables* 31a⟩+≡ (30) ◁31a 32e▷

```
$(foreach suf,${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.

32a ⟨*target generation for transformations* 32a⟩≡ (30)

```
define target_recipe
$(1):
  ⟨transformation recipe 31d⟩
endef
$(foreach target,${TRANSFORM_TARGETS},$(eval $(call target_recipe,${target})))
```

Note that we use the same recipe as above.

5.3.1 Removing solutions

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

32b ⟨*transformations* 31c⟩+≡ (31b) ◁31c 32c▷

```
NoSolutions?= ${SED} "/\\\\begin{solution}/,/\\\\end{solution}/d"
```

5.3.2 Removing excessive build instructions

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

32c ⟨*transformations* 31c⟩+≡ (31b) ◁32b 32d▷

```
ExportFilter?=    ${SED} "/#export \\(false\\|no\\)/,/#export \\(true\\|yes\\)/d"
OldExportFilter?= ${SED} "/#export no/,/#endexport/d"
```

5.3.3 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.

32d ⟨*transformations* 31c⟩+≡ (31b) ◁32c

```
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.

32e ⟨*variables* 31a⟩+≡ (30) ◁31e 33a▷

```
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].

33a ⟨*variables* 31a⟩+≡ (30) ◁32e 34c▷

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

5.4 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[1]. For now, we will use some functions which requires parameters — the transformations above must not require any parameter — so the outline looks like this:

33b ⟨*suffix rules for camera-ready source* 33b⟩≡ (30)

```
⟨function to substitute bibliography for bbl 33c⟩
⟨function to fill filecontents environments 33d⟩
⟨function to insert biblatex bbl code 34a⟩

.SUFFIXES: .tex .cameraready.tex
.tex.cameraready.tex:
  ⟨camera-ready recipe 34b⟩
```

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`).

33c ⟨*function to substitute bibliography for bbl* 33c⟩≡ (33b)

```
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.

33d ⟨*function to fill filecontents environments* 33d⟩≡ (33b)

```
define filecontent
```

[1]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)"
endef
```

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

34a ⟨*function to insert biblatex bbl code* 34a⟩≡ (33b)

```
define _the_bblcode
\\\\makeatletter\\\\def\\\\blx@bblfile@biber{\\\\blx@secinit\\\\begingroup\\\\blx@bblstart
endef

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

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

34b ⟨*camera-ready recipe* 34b⟩≡ (33b)

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

5.5 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.

34c ⟨*variables* 31a⟩+≡ (30) ◁33a

```
GPG?=                 gpg
TRANSFORM_ENC?=          ${GPG} -aes
TRANSFORM_RECIPIENTS?=   me
TRANSFORM_DEC?=          ${GPG} -d
```

This will yield the following suffix rules.

34d ⟨*suffix rules for encrypted files* 34d⟩≡ (30)

```
.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[2] [Aye]. The set up we must do for this to work is to set a Git attribute.

35 ⟨*gitattributes* 35⟩≡

```
*.asc     filter=git-crypt
```

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

[2]Install on Ubuntu by running `sudo apt install git-crypt`.

Part III

Papers and documents

Chapter 6

tex.mk

6.1 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 languages like C.

We provide several suffix rules. First, for ordinary documents, i.e. to compile a `.tex` file to `.pdf`, `.ps` or `.dvi`. Second, for classes and packages, i.e. to compile a DocTeX `.dtx` file to `.pdf`, `.ps` or `.dvi` and an `.ins` file to `.cls` or `.sty`. The suffix rules we provide here follows the conventions set out in [GNU16, Sect. 10.2].

The latex(1) and pdflatex(1) commands are controlled by

37a ⟨*variables* 37a⟩≡ (39a) 37b ▷

```
LATEX?=       latexmk -dvi
PDFLATEX?=    latexmk -pdf
```

Possible flags are controlled by

37b ⟨*variables* 37a⟩+≡ (39a) ◁ 37a 37c ▷

```
LATEXFLAGS?=  -use-make
```

The output directory is by default

37c ⟨*variables* 37a⟩+≡ (39a) ◁ 37b 37d ▷

```
TEX_OUTDIR?=  ltxobj
```

We note that we add this output directory to the search path for prerequisites [see GNU16, Sect. 4.5.1]. The reason for this is that we might need some out the objects as prerequisites for other files.

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.

37d ⟨*variables* 37a⟩+≡ (39a) ◁ 37c 38a ▷

```
TEX_BBL?=
```

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

38a ⟨*variables* 37a⟩+≡ (39a) ◁37d 38b▷
```
BIBTEX?=      bibtexu
BIBTEXFLAGS?=
```

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

38b ⟨*variables* 37a⟩+≡ (39a) ◁38a 38c▷
```
BIBER?=       biber
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.

38c ⟨*variables* 37a⟩+≡ (39a) ◁38b 38d▷
```
TEX_IND?=
```

The indexing-related programs are the following.

38d ⟨*variables* 37a⟩+≡ (39a) ◁38c 38e▷
```
MAKEINDEX?=   makeindex
MAKEIDXFLAGS?=
XINDY?=       texindy
XINDYFLAGS?=
```

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

38e ⟨*variables* 37a⟩+≡ (39a) ◁38d 38f▷
```
TEX_PYTHONTEX?=
```

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

38f ⟨*variables* 37a⟩+≡ (39a) ◁38e 49a▷
```
PYTHONTEX?=   pythontex3
PYTHONTEXFLAGS?=
```

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,
- `acmproc` for the Association for Computing Machinery (ACM) special interest group proceedings,
- `acmsmall` and `acmlarge` for the ACM journal formats,
- `rfc` or `rfc.bib` for an up-to-date bibliography containing all Internet Engineering Task Force (IETF) Request For Commentss (RFCs).

6.2 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.

39a ⟨*tex.mk* 39a⟩≡

```
ifndef TEX_MK
TEX_MK=true

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

⟨variables 37a⟩
⟨targets for documents 40a⟩
⟨targets for class and package files 45a⟩
⟨targets for external classes 46a⟩
⟨targets for cleaning 39b⟩

endif
```

We include `portability.mk` (Chapter 1) to get portable settings for several common utilities.

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.

39b ⟨*targets for cleaning* 39b⟩≡ (39a)

```
.PHONY: clean clean-tex
clean: clean-tex

clean-tex:
  ⟨clean recipe 39c⟩

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

distclean-tex:
  ⟨distclean recipe 40c⟩
```

We will add to the recipes in the remainder of the chapter. However, as latexmk(1) is set as the default in Section 6.1, we can already add the following line to the cleaning recipe.

39c ⟨*clean recipe* 39c⟩≡ (39b)

```
-latexmk -C -output-directory=${TEX_OUTDIR}
${RM} -R ${TEX_OUTDIR}
${RM} *.pytxcode
${RM} pythontex-files-*
```

6.3 Targets for documents

Now we will treat how to compile documents.

40a ⟨*targets for documents* 40a⟩≡ (39a)
  ⟨*auxillary files* 40d⟩
  ⟨*bibliography files* 41a⟩
  ⟨*indices files* 42a⟩
  ⟨*PythonTeX files* 43a⟩
  ⟨*document files* 44a⟩
  ⟨*target for latexmkrc* 40b⟩

These will be discussed in the following sections. However, since we use latexmk(1) by default (Section 6.1), we will discuss the relevant ⟨*latexmkrc* 42e⟩ entries in parallel. We supply a target to easily use our ⟨*latexmkrc* 42e⟩ with latexmk(1).

40b ⟨*target for latexmkrc* 40b⟩≡ (40a)
```
latexmkrc:
	[ -e $@ -o "${INCLUDE_MAKEFILES}" = "." ] || \
	${LN} -s ${INCLUDE_MAKEFILES}/latexmkrc $@
```

We also add the corresponding line for cleaning.

40c ⟨*distclean recipe* 40c⟩≡ (39b)
```
[ ! -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).

6.3.1 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.

40d ⟨*auxillary files* 40d⟩≡ (40a)
```
${TEX_OUTDIR}/%.aux: %.tex
```
  ⟨*try to create output directory* 40e⟩
  ⟨*run pdflatex* 40f⟩

To create the output directory we simply try mkdir(1).

40e ⟨*try to create output directory* 40e⟩≡ (40–42)
```
${MKDIR} ${TEX_OUTDIR}
```

We can the simply run latex(1) with the specified output directory. Note that it does not matter whether we run latex(1) or pdflatex(1) to generate the `.aux` file.

40f ⟨*run pdflatex* 40f⟩≡ (40–42 45d)
```
${PDFLATEX} -output-directory=${TEX_OUTDIR} ${LATEXFLAGS} $<
```

6.3.2 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).

41a ⟨*bibliography files* 41a⟩≡ (40a) 41f ▷

```
⟨bbl target for bibtex 41b⟩
⟨bbl target for biber 41d⟩
```

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.

41b ⟨*bbl target for bibtex* 41b⟩≡ (41a)

```
${TEX_OUTDIR}/%.bbl: ${TEX_OUTDIR}/%.aux
    ⟨compile bbl with bibtex 41c⟩
```

To compile the `.bbl` file using bibtex(1) and to put the output files in the desired output directory, we can do this.

41c ⟨*compile bbl with bibtex* 41c⟩≡ (41b)

```
${BIBTEX} ${BIBTEXFLAGS} $<
${MV} $@ ${@:.bbl=.blg} ${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.

41d ⟨*bbl target for biber* 41d⟩≡ (41a) 41e ▷

```
${TEX_OUTDIR}/%.bcf: %.tex
    ⟨try to create output directory 40e⟩
    ⟨run pdflatex 40f⟩
```

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.

41e ⟨*bbl target for biber* 41d⟩+≡ (41a) ◁ 41d

```
${TEX_OUTDIR}/%.bbl: ${TEX_OUTDIR}/%.bcf
    ${BIBER} -O $@ ${BIBERFLAGS} $<
```

As mentioned in Section 6.1, we can automatically add the `.bbl` file as a prerequisite if the variable `TEX_BBL` is set.

41f ⟨*bibliography files* 41a⟩+≡ (40a) ◁ 41a 43b ▷

```
ifneq (${TEX_BBL},)
%.pdf: ${TEX_OUTDIR}/%.bbl
endif
```

6.3.3 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 `nomencl` 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.

42a ⟨*indices files* 42a⟩≡ (40a) 42b ▷
```
${TEX_OUTDIR}/%.idx: %.tex
  ⟨try to create output directory 40e⟩
  ⟨run pdflatex 40f⟩
```

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

42b ⟨*indices files* 42a⟩+≡ (40a) ◁ 42a 42c ▷
```
${TEX_OUTDIR}/%.ind: ${TEX_OUTDIR}/%.idx
  ${XINDY} -o $@ ${XINDYFLAGS} $<
```

As mentioned in Section 6.1, we can automatically add the `.ind` file as a prerequisite if the variable `TEX_IND` is set.

42c ⟨*indices files* 42a⟩+≡ (40a) ◁ 42b 42d ▷
```
ifneq (${TEX_IND},)
%.pdf: ${TEX_OUTDIR}/%.ind
endif
```

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

42d ⟨*indices files* 42a⟩+≡ (40a) ◁ 42c
```
${TEX_OUTDIR}/%.nlo: %.tex
  ⟨try to create output directory 40e⟩
  ⟨run pdflatex 40f⟩

${TEX_OUTDIR}/%.nls: ${TEX_OUTDIR}/%.nlo
  ⟨try to create output directory 40e⟩
  ${MAKEINDEX} -o $@ ${MAKEIDXFLAGS} -s nomencl.ist $<
```

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)[1].

42e ⟨*latexmkrc* 42e⟩≡ 43c ▷
```
add_cus_dep( 'nlo', 'nls', 0, 'makenlo2nls' );
sub makenlo2nls {
  system( "makeindex -s nomencl.ist -o \"$_[0].nls\" \"$_[0].nlo\"" );
}
```

[1]URL: `http://mirrors.ctan.org/support/latexmk/example_rcfiles/nomenclature_latexmkrc`

6.3.4 PythonTeX

Occasionally we use PythonTeX. We also provide a target for the required files.

43a ⟨*PythonTeX files* 43a⟩≡ (40a)

```
pythontex-files-%/%.pytxcode: %.tex
  ${PYTHONTEX} ${PYTHONTEXFLAGS} $<
```

As mentioned in Section 6.1, we can automatically add the `.pytxmcr` file as a prerequisite if the variable `TEX_PYTHONTEX` is set.

43b ⟨*bibliography files* 41a⟩+≡ (40a) ◁41f

```
ifneq (${TEX_PYTHONTEX},)
%.pdf: ${TEX_OUTDIR}/pythontex-files-%/%.pytxcode
endif
```

If we use latexmk(1), then we must also add instructions for this in ⟨*latexmkrc* 42e⟩. The following code is fetched from the latexmk example-files on CTAN[2].

43c ⟨*latexmkrc* 42e⟩+≡ ◁42e

```
#  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
#  trickery 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;
```

[2]URL: http://mirrors.ctan.org/support/latexmk/example_rcfiles/pythontex-latexmkrc

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

6.3.5 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 hard link between them after compilation — this allows make(1) to track modification times correctly.

44a ⟨*document files* 44a⟩≡ (40a)

```
%.pdf ${TEX_OUTDIR}/%.pdf: %.tex
  ⟨compile PDF 44b⟩
  -${LN} ${TEX_OUTDIR}/$@ $@

%.dvi ${TEX_OUTDIR}/%.dvi: %.tex
  ⟨compile DVI 44c⟩
  -${LN} ${TEX_OUTDIR}/$@ $@
```

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

44b ⟨*compile PDF* 44b⟩≡ (44a 45c)

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

And the same for DVI files.

44c ⟨*compile DVI* 44c⟩≡ (44a 45c)

```
${LATEX} -output-directory=${TEX_OUTDIR} ${LATEXFLAGS} $<
```

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

6.4 Targets for class and package files

There are two parts concerning class and package files.

45a ⟨*targets for class and package files* 45a⟩≡ (39a)
```
  ⟨compile sty and cls files 45b⟩
  ⟨compile class and package documentation 45c⟩
```

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.

45b ⟨*compile sty and cls files* 45b⟩≡ (45a)
```
  %.cls %.sty: %.ins
    ${LATEX} $<
```

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

45c ⟨*compile class and package documentation* 45c⟩≡ (45a) 45d ▷
```
  %.pdf ${TEX_OUTDIR}/%.pdf: %.dtx
    ⟨compile PDF 44b⟩
    -${LN} ${TEX_OUTDIR}/$@ $@

  %.dvi ${TEX_OUTDIR}/%.dvi: %.dtx
    ⟨compile DVI 44c⟩
    -${LN} ${TEX_OUTDIR}/$@ $@
```

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

45d ⟨*compile class and package documentation* 45c⟩+≡ (45a) ◁ 45c
```
  ${TEX_OUTDIR}/%.aux: %.dtx
    ⟨run pdflatex 40f⟩

  ${TEX_OUTDIR}/%.bcf: %.dtx
    ⟨run pdflatex 40f⟩

  ${TEX_OUTDIR}/%.idx: %.dtx
    ⟨run pdflatex 40f⟩
```

6.5 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.

46a ⟨*targets for external classes* 46a⟩≡ (39a)
  ⟨*a general downloader* 46b⟩
  ⟨*Springer LNCS* 48a⟩
  ⟨*biblatex LNCS style* 48c⟩
  ⟨*ACM classes* 48d⟩
  ⟨*the RFC bibliography* 50b⟩
  ⟨*PoPETS* 51b⟩

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.

46b ⟨*a general downloader* 46b⟩≡ (46a)

```
define download_archive
⟨targets for symlinks 46c⟩
⟨targets for desired files 47b⟩
⟨targets for archive 47c⟩
⟨target for cleaning 47g⟩
endef
define download_repo
⟨targets for symlinks 46c⟩
⟨targets for desired files 47b⟩
⟨targets for repo 47d⟩
⟨target for cleaning 47g⟩
endef
```

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.

46c ⟨*targets for symlinks* 46c⟩≡ (46b) 47a ▷

```
$(foreach file,${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.

47a ⟨*targets for symlinks* 46c⟩+≡ (46b) ◁46c

```
.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.

47b ⟨*targets for desired files* 47b⟩≡ (46b)

```
$(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.

47c ⟨*targets for archive* 47c⟩≡ (46b) 47e▷

```
$(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.

47d ⟨*targets for repo* 47d⟩≡ (46b) 47f▷

```
$(addprefix ${TEX_EXT_DIR-$(1)}/,${TEX_EXT_FILES-$(1)}):
  ${LN} ${TEX_EXT_DIR-$(1)}/${TEX_EXT_SRC-$(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 to fetch it from in both cases. In the case of an archive we do the following.

47e ⟨*targets for archive* 47c⟩+≡ (46b) ◁47c

```
${TEX_EXT_DIR-$(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.

47f ⟨*targets for repo* 47d⟩+≡ (46b) ◁47d

```
${TEX_EXT_DIR-$(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.

47g ⟨*target for cleaning* 47g⟩≡ (46b)

```
.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)}
```

6.5.1 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[3]. 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.

We use the downloader described above.

48a ⟨*Springer LNCS* 48a⟩≡ (46a) 48b ▷

```
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?=    ftp://ftp.springer.de/pub/tex/latex/llncs/latex2e/llncs2e.zip
TEX_EXT_EXTRACT-lncs?=${UNZIP} $$< -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`.

48b ⟨*Springer LNCS* 48a⟩+≡ (46a) ◁ 48a

```
.PHONY: llncs
llncs: lncs
```

6.5.2 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.

```
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.

48c ⟨*biblatex LNCS style* 48c⟩≡ (46a)

```
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/neapel/biblatex-lncs.git

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

6.5.3 ACM classes

We provide targets for the following ACM classes.

48d ⟨*ACM classes* 48d⟩≡ (46a)
  ⟨*ACM SIG proceedings* 49b⟩

[3]It would be most desirable that they were made available in CTAN under an open license.

⟨*ACM small standard* 49e⟩
⟨*ACM large standard* 50a⟩

Special Interest Group proceedings

The structure is similar as above. We use a variable to control the destination, current directory by default.

49a ⟨*variables* 37a⟩+≡ (39a) ◁38f

```
TEX_EXT_DIR-acmproc?=     acm
```

Then we download the class file to that directory.

49b ⟨*ACM SIG proceedings* 49b⟩≡ (48d) 49c▷

```
${TEX_EXT_DIR-acmproc}/acm_proc_article-sp.cls:
  ${CURL} -o $@ http://www.acm.org/sigs/publications/acm_proc_article-sp.cls
acm_proc_article-sp.cls: ${TEX_EXT_DIR-acmproc}/acm_proc_article-sp.cls
  ${LN} $^ $@
```

We also add the phony target `acmproc` to easily add this class as a prerequisite for a document.

49c ⟨*ACM SIG proceedings* 49b⟩+≡ (48d) ◁49b 49d▷

```
.PHONY: acmproc
acmproc: acm_proc_article-sp.cls
```

The cleaning will only need to remove the single document class that we downloaded.

49d ⟨*ACM SIG proceedings* 49b⟩+≡ (48d) ◁49c

```
.PHONY: distclean clean-acmproc
distclean: clean-acmproc
clean-acmproc:
  ${RM} acm_proc_article-sp.cls
  ${RM} ${TEX_EXT_DIR-acmproc}/acm_proc_article-sp.cls
```

ACM small standard

We can use the downloader for the small standard.

49e ⟨*ACM small standard* 49e⟩≡ (48d)

```
TEX_EXT_FILES-acmsmall?=  acmsmall.cls
TEX_EXT_DIR-acmsmall?=    acm
TEX_EXT_SRC-acmsmall?=    v2-acmsmall.zip
TEX_EXT_URL-acmsmall?=    http://www.acm.org/publications/latex_style/v2-acmsmall.zip
TEX_EXT_EXTRACT-acmsmall?=${UNZIP} $< -d ${TEX_EXT_DIR-acmsmall}

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

ACM large standard

As for the small standard, we can use the downloader.

50a ⟨*ACM large standard* 50a⟩≡ (48d)
```
TEX_EXT_FILES-acmlarge?=  acmlarge.cls
TEX_EXT_DIR-acmlarge?=    acm
TEX_EXT_SRC-acmlarge?=    v2-acmlarge.zip
TEX_EXT_URL-acmlarge?=    http://www.acm.org/publications/latex_style/v2-acmlarge.zip
TEX_EXT_EXTRACT-acmlarge?=${UNZIP} $< -d ${TEX_EXT_DIR-acmlarge}

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

6.5.4 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.

50b ⟨*the RFC bibliography* 50b⟩≡ (46a) 50e ▷
```
rfc.bib:
  ⟨download rfc.bib 50c⟩
  ⟨change misc to techreport 50d⟩

${TEXMF}/tex/latex/rfc.bib:
  mkdir -p ${TEXMF}/tex/latex/
  ⟨download rfc.bib 50c⟩
  ⟨change misc to techreport 50d⟩
```

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.

50c ⟨*download rfc.bib* 50c⟩≡ (50b)
```
${CURL} -o - http://tm.uka.de/~bless/rfc.bib.gz 2>/dev/null \
  | ${UNCOMPRESS} - > $@ ; \
```

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

50d ⟨*change misc to techreport* 50d⟩≡ (50b)
```
${SED} -i "s/@misc/@techreport/" $@
```

We also provide a phony target for these two files.

50e ⟨*the RFC bibliography* 50b⟩+≡ (46a) ◁ 50b 51a ▷
```
.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.

51a ⟨*the RFC bibliography* 50b⟩+≡ (46a) ◁50e

```
.PHONY: distclean clean-rfc
distclean: clean-rfc
clean-rfc:
  ${RM} rfc.bib
```

6.5.5 Proceedings of the Privacy Enhancing Technologies Symposium

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

51b ⟨*PoPETS* 51b⟩≡ (46a)

```
TEX_EXT_FILES-popets?=by-nc-nd.pdf dg-degruyter.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 7

doc.mk

7.1 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 7.2.1 for details. We also provide a target `wc` which counts the words of its prerequisites (Section 7.2.2). Finally, we also provide a set of suffix rules for automatic conversion between different formats, see Section 7.2.4 for details about the formats.

7.2 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.

52 ⟨*doc.mk* 52⟩≡

```
ifndef DOC_MK
DOC_MK=true

⟨variables 53a⟩
⟨target for printing 53b⟩
⟨target for word counting 54a⟩
⟨target for to-do lists 54b⟩
⟨suffix rules for format conversion 54d⟩

endif
```

7.2.1 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.

53a ⟨*variables* 53a⟩≡ (52) 53c ▷

```
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 7.2.4) 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.

53b ⟨*target for printing* 53b⟩≡ (52)

```
.PHONY: print
print:
	$(foreach doc,$^,\
	  $(if ${LPR-${doc}},${LPR-${doc}},${LPR}) \
	  $(if ${LPRFLAGS-${doc}},${LPRFLAGS-${doc}},${LPRFLAGS}) \
	  ${doc};)
```

7.2.2 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.

53c ⟨*variables* 53a⟩+≡ (52) ◁ 53a 54c ▷

```
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.

54a ⟨*target for word counting* 54a⟩≡ (52)
```
.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});)
```

7.2.3 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').

54b ⟨*target for to-do lists* 54b⟩≡ (52)
```
.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.

54c ⟨*variables* 53a⟩+≡ (52) ◁53c 55a▷
```
TODO?=        ${GREP} "\(XXX\|TODO\|FIXME\)"
TODOFLAGS?=
```

7.2.4 Format conversion

The format conversion is done using pattern rules. This means that whenever we need a file in a certain format, we simply keep the name but change the suffix ('file extension'). The conversions that are implemented are the following:

54d ⟨*suffix rules for format conversion* 54d⟩≡ (52)
⟨*PDF to PS* 55b⟩
⟨*PS to PDF* 55c⟩
⟨*DVI to PS* 55e⟩
⟨*ODT to PDF* 55g⟩

⟨*SVG to PDF* 56b⟩
⟨*SVG to PS* 56c⟩
⟨*DIA to TeX* 56e⟩

⟨*MD to TeX* 56g⟩
⟨*TeX to text* 57b⟩

Document formats

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

55a ⟨*variables* 53a⟩+≡ (52) ◁54c 55d▷
```
PDF2PS?=      pdf2ps
PDF2PSFLAGS?=
PS2PDF?=      ps2pdf
PS2PDFFLAGS?=
```
This allows us to specify the rule as follows.

55b ⟨*PDF to PS* 55b⟩≡ (54d)
```
%.ps: %.pdf
  ${PDF2PS} ${PDF2PSFLAGS} $<
```
We also have the other way around.

55c ⟨*PS to PDF* 55c⟩≡ (54d)
```
%.pdf: %.ps
  ${PS2PDF} ${PS2PDFFLAGS} $<
```
We do similarly for DVI-files that we want to convert to PostScript.

55d ⟨*variables* 53a⟩+≡ (52) ◁55a 55f▷
```
DVIPS?=       dvips
DVIPSFLAGS?=
```
With those variables we let

55e ⟨*DVI to PS* 55e⟩≡ (54d)
```
%.ps: %.dvi
  ${DVIPS} ${DVIPSFLAGS} $<
```
There is no good conversion program for the Open Document Format (ODF) files. We will use LibreOffice.

55f ⟨*variables* 53a⟩+≡ (52) ◁55d 56a▷
```
ODT2PDF?=     soffice -convert-to pdf
ODT2PDFFLAGS?=-headless
```
This yields the following suffix rule.

55g ⟨*ODT to PDF* 55g⟩≡ (54d)
```
%.pdf: %.odt
  ${ODT2PDF} ${ODT2PDFFLAGS} $<
```

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.

56a ⟨*variables* 53a⟩+≡ (52) ◁55f 56d▷
```
INKSCAPE?=      inkscape
INKSCAPEFLAGS?= -D -z -export-latex
```

56b ⟨*SVG to PDF* 56b⟩≡ (54d)
```
%.pdf: %.svg
  ${INKSCAPE} ${INKSCAPEFLAGS} -file=$< -export-pdf=$@
```

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

56c ⟨*SVG to PS* 56c⟩≡ (54d)
```
%.ps: %.svg
  ${INKSCAPE} ${INKSCAPEFLAGS} -file=$< -export-ps=$@

%.eps: %.svg
  ${INKSCAPE} ${INKSCAPEFLAGS} -file=$< -export-eps=$@
```

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.

56d ⟨*variables* 53a⟩+≡ (52) ◁56a 56f▷
```
DIA?=           dia
DIAFLAGS?=
```

That gives the suffix rule as follows.

56e ⟨*DIA to TeX* 56e⟩≡ (54d)
```
%.tex: %.dia
  ${DIA} ${DIAFLAGS} -e $@ -t pgf-tex $<
```

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.

56f ⟨*variables* 53a⟩+≡ (52) ◁56d 57a▷
```
MD2TEX?=        pandoc -f markdown -t latex
MD2TEXFLAGS?=
```

This gives the following suffix rule.

56g ⟨*MD to TeX* 56g⟩≡ (54d)
```
%.tex: %.md
  ${MD2TEX} ${MD2TEXFLAGS} < $< > $@
```

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.

57a ⟨*variables* 53a⟩+≡ (52) ◁56f

```
TEX2TEXT?=       detex
TEX2TEXTFLAGS?=
```

This gives us the following suffix rule.

57b ⟨*TeX to text* 57b⟩≡ (54d)

```
%.txt: %.tex
	${TEX2TEXT} ${TEX2TEXTFLAGS} $< > $@
```

Part IV

Literate programming

Chapter 8

noweb.mk

8.1 Introduction and usage

The `noweb.mk` include provides suffix rules for weaving and tangling (produce documentation and code, respectively). To use it correctly there are some suffix naming conventions.

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. This means that some jobs must be done using specific recipe.

We assume that there is a main TeX file which will include the woven documentation. So all invocations of `noweave` use the `-n` option. Furthermore, some language-specialized suffixes, such as `cxx.nw`, will use `noweave` options suitable for that language.

8.2 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.

59 ⟨*noweb.mk* 59⟩≡
```
ifndef NOWEB_MK
NOWEB_MK = true

⟨variables 60a⟩
⟨suffix rules for weaving documentation 60c⟩
⟨suffix rules for tangling code 60d⟩

endif
```

8.2.1 Weaving documentation

We will use the `noweave` command to weave the documentation.

60a ⟨*variables* 60a⟩≡ (59) 60b▷

```
NOWEAVE?=        noweave
```

The default options that we will use can be controlled with the following variable.

60b ⟨*variables* 60a⟩+≡ (59) ◁60a 60e▷

```
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.

60c ⟨*suffix rules for weaving documentation* 60c⟩≡ (59)

```
.SUFFIXES: .nw .tex $(addsuffix .nw,${NOWEB_SUFFIXES})
.nw.tex $(addsuffix .nw.tex,${NOWEB_SUFFIXES}):
	${NOWEAVE} ${NOWEAVEFLAGS} $< > $@
```

For 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.

8.2.2 Tangling code

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

60d ⟨*suffix rules for tangling code* 60d⟩≡ (59)

⟨*general tangling rules* 61a⟩
⟨*special rules for different languages* 61e⟩

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

We will use notangle(1).

60e ⟨*variables* 60a⟩+≡ (59) ◁60b 60f▷

```
NOTANGLE?=       notangle
NOTANGLEFLAGS?= -t2
```

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.

60f ⟨*variables* 60a⟩+≡ (59) ◁60e 61d▷

```
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.

61a ⟨*general tangling rules* 61a⟩≡ (60d)
⟨*tangle source files with suffix* 61c⟩
⟨*tangle source files without suffix* 61b⟩

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].

61b ⟨*tangle source files without suffix* 61b⟩≡ (61a)
```
$(addprefix %,${NOWEB_SUFFIXES}): %.nw
	${NOTANGLE$(suffix $@)}
```

The case with suffixes can paradoxically be done without introducing the suffixes.

61c ⟨*tangle source files with suffix* 61c⟩≡ (61a)
```
define with_suffix_target
%$(1): %$(1).nw
	$${NOTANGLE$$(suffix $$@)}
endef
$(foreach suf,${NOWEB_SUFFIXES},$(eval $(call with_suffix_target,${suf})))
```

The reason for this is that the suffix is now captured by the pattern on both sides, i.e. for target *and* prerequisite. However, this rule does not capture some thing 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.

61d ⟨*variables* 60a⟩+≡ (59) ◁60f
⟨*defaults for C and C++* 62a⟩
⟨*defaults for Haskell* 62d⟩
⟨*defaults for Python* 63a⟩
⟨*defaults for Make* 63b⟩
⟨*defaults for shell scripts* 63c⟩

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

61e ⟨*special rules for different languages* 61e⟩≡ (60d)
⟨*rules for C and C++* 62b⟩

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.

62a ⟨*defaults for C and C++* 62a⟩≡ (61d) 62c▷
```
NOWEB_SUFFIXES+=     .c .cc .cpp .cxx
NOTANGLEFLAGS.c?=    ${NOTANGLEFLAGS} -L
NOTANGLE.c?=         ${NOTANGLE} ${NOTANGLEFLAGS.c} -R$@ $< | ${CPIF} $@
NOTANGLEFLAGS.cc?=   ${NOTANGLEFLAGS.c}
NOTANGLE.cc?=        ${NOTANGLE} ${NOTANGLEFLAGS.cc} -R$@ $< | ${CPIF} $@
NOTANGLEFLAGS.cpp?=  ${NOTANGLEFLAGS.c}
NOTANGLE.cpp?=       ${NOTANGLE} ${NOTANGLEFLAGS.cpp} -R$@ $< | ${CPIF} $@
NOTANGLEFLAGS.cxx?=  ${NOTANGLEFLAGS.c}
NOTANGLE.cxx?=       ${NOTANGLE} ${NOTANGLEFLAGS.cxx} -R$@ $< | ${CPIF} $@
```

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.

62b ⟨*rules for C and C++* 62b⟩≡ (61e)
```
%.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.

62c ⟨*defaults for C and C++* 62a⟩+≡ (61d) ◁62a
```
NOWEB_SUFFIXES+=     .h .hh .hpp .hxx
NOTANGLEFLAGS.h?=    ${NOTANGLEFLAGS} -L
NOTANGLE.h?=         ${NOTANGLE} ${NOTANGLEFLAGS.h} -R$@ $< | ${CPIF} $@
NOTANGLEFLAGS.hh?=   ${NOTANGLEFLAGS.h}
NOTANGLE.hh?=        ${NOTANGLE} ${NOTANGLEFLAGS.hh} -R$@ $< | ${CPIF} $@
NOTANGLEFLAGS.hpp?=  ${NOTANGLEFLAGS.h}
NOTANGLE.hpp?=       ${NOTANGLE} ${NOTANGLEFLAGS.hpp} -R$@ $< | ${CPIF} $@
NOTANGLEFLAGS.hxx?=  ${NOTANGLEFLAGS.h}
NOTANGLE.hxx?=       ${NOTANGLE} ${NOTANGLEFLAGS.hxx} -R$@ $< | ${CPIF} $@
```

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.

62d ⟨*defaults for Haskell* 62d⟩≡ (61d)
```
NOWEB_SUFFIXES+=     .hs
NOTANGLEFLAGS.hs?=   ${NOTANGLEFLAGS} -L
NOTANGLE.hs?=        ${NOTANGLE} ${NOTANGLEFLAGS.hs} -R$@ $< | ${CPIF} $@
```

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`).

For Python, there is no special processing needed, we simply use the flags we set above.

63a ⟨*defaults for Python* 63a⟩≡ (61d)
```
NOWEB_SUFFIXES+=    .py
NOTANGLEFLAGS.py?=  ${NOTANGLEFLAGS}
NOTANGLE.py?=       ${NOTANGLE} ${NOTANGLEFLAGS.py} -R$@ $< > $@
```

It is the same case for makefiles.

63b ⟨*defaults for Make* 63b⟩≡ (61d)
```
NOWEB_SUFFIXES+=    .mk
NOTANGLEFLAGS.mk?=  ${NOTANGLEFLAGS}
NOTANGLE.mk?=       ${NOTANGLE} ${NOTANGLEFLAGS.mk} -R$@ $< > $@
```

And also for shell scripts.

63c ⟨*defaults for shell scripts* 63c⟩≡ (61d)
```
NOWEB_SUFFIXES+=    .sh
NOTANGLEFLAGS.sh?=  ${NOTANGLEFLAGS}
NOTANGLE.sh?=       ${NOTANGLE} ${NOTANGLEFLAGS.sh} -R$@ $< > $@
```

Chapter 9

haskell.mk

9.1 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.

64a ⟨*haskell.mk* 64a⟩≡
```
⟨default variables 64b⟩
⟨suffix rules for Haskell programs 64c⟩
```

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.

64b ⟨*default variables* 64b⟩≡ (64a)
```
LD=   ghc
```

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

64c ⟨*suffix rules for Haskell programs* 64c⟩≡ (64a)
```
.SUFFIXES: .hs .lhs
.hs.o .lhs.o:
  ghc ${HSFLAGS} -c $<
```

Part V

Assessment

Chapter 10

exam.mk

10.1 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* 67d⟩, 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`. 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:

66a ⟨*variables* 66a⟩≡ (67d) 66b▷

```
EXAM_NAME?=   exam
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.)

66b ⟨*variables* 66a⟩+≡ (67d) ◁66a 67a▷

```
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.

67a ⟨*variables* 66a⟩+≡ (67d) ◁66b 67b▷

```
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[1].

67b ⟨*variables* 66a⟩+≡ (67d) ◁67a 67c▷

```
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.

67c ⟨*variables* 66a⟩+≡ (67d) ◁67b

```
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
```

10.2 Implementation

We want to create a makefile ⟨*exam.mk* 67d⟩ for inclusion. The file will have the following outline:

67d ⟨*exam.mk* 67d⟩≡
  ⟨*variables* 66a⟩
  ⟨*generate targets for exams* 68a⟩
  ⟨*generate targets for questions* 68e⟩

[1]This also includes all future exams, but `examgen` will ignore those since they do not yet exist.

As suggested above, each exam `exam-ID.pdf` depends on at least two files: `exam-ID.tex` and `questions-ID.tex`. We will automatically generate these targets by iterating over the list in `EXAM_IDS`. We will not provide any recipe, that is left for the user or the use of `tex.mk`. What we will do is the following:

68a ⟨*generate targets for exams* 68a⟩≡ (67d)
 ⟨*define target-specific variables* 68b⟩
 ⟨*define callable exam definition* 68c⟩
 ⟨*call the exam definition for each ID* 68d⟩

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.

68b ⟨*define target-specific variables* 68b⟩≡ (68a)

```
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`).

68c ⟨*define callable exam definition* 68c⟩≡ (68a)

```
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.

68d ⟨*call the exam definition for each ID* 68d⟩≡ (68a)

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

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.

68e ⟨*generate targets for questions* 68e⟩≡ (67d)
 ⟨*define target-specific questions variables* 68f⟩
 ⟨*define the questions target* 69⟩

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

68f ⟨*define target-specific questions variables* 68f⟩≡ (68e)

```
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.

69 ⟨*define the questions target* 69⟩≡ (68e)

```
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 11

results.mk

11.1 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:

70 ⟨*variables* 70⟩≡ (71a) 71b ▷

```
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* 70⟩, then ⟨*targets* 71d⟩. 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.

71a ⟨*results.mk* 71a⟩≡

```
ifndef MIUN_RESULTS_MK
MIUN_RESULTS_MK=true

⟨variables 70⟩
⟨targets 71d⟩

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.

71b ⟨*variables* 70⟩+≡ (71a) ◁70 71c▷

```
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[1].

71c ⟨*variables* 70⟩+≡ (71a) ◁71b 72▷

```
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.

71d ⟨*targets* 71d⟩≡ (71a) 71e▷

```
.PHONY: report
report:
  ⟨report recipe 76e⟩
```

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`.

71e ⟨*targets* 71d⟩+≡ (71a) ◁71d 73a▷

```
.PHONY: clean clean-results
clean: clean-results
```

[1]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 73b⟩
```

We will populate the ⟨*clean recipe* 73b⟩ as we go.

11.1.1 Portability

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

72 ⟨*variables* 70⟩+≡ (71a) ◁71c 73f▷

```
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*.

11.2 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 `${in}`, then we want the a report of changes to send to the student office (Section 11.2.2). There are different identifiers used in the national database and in Moodle. So we need to extract the identifiers in Moodle and convert to those in the national database (Section 11.2.3). Then we can send the report and update our local representation of what is reported to the national database, i.e. `${out}`.

11.2.1 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}`.

73a ⟨*targets* 71d⟩+≡ (71a) ◁71e 73g▷
```
${out}.new: ${in}
    ⟨new recipe 73c⟩
```
This means we should also add `${out}.diff` to the recipe of clean.

73b ⟨*clean recipe* 73b⟩≡ (71e) 75f▷
```
${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.

73c ⟨*new recipe* 73c⟩≡ (73a) 73d▷
```
${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.

73d ⟨*new recipe* 73c⟩+≡ (73a) ◁73c 73e▷
```
${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.

73e ⟨*new recipe* 73c⟩+≡ (73a) ◁73d
```
$(if ${RESULTS_REWRITES},| ${SED} "s/ //g", ) \
$(foreach regex,${RESULTS_REWRITES},| ${SED} ${regex}) \
> $@
```
We let the default rewrites be

73f ⟨*variables* 70⟩+≡ (71a) ◁72 74b▷
```
RESULTS_REWRITES+=  "s/Godkänd(G)/G/g" "s/Underkänd(U)/U/g"
RESULTS_REWRITES+=  "s/Komplettering(Fx)/Fx/g"
RESULTS_REWRITES+=  "s/\"//g"
```

11.2.2 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.

73g ⟨*targets* 71d⟩+≡ (71a) ◁73a 74e▷
```
${out}.diff: ${out}.new
    ⟨diff recipe 74a⟩
```

We are not interested in the headers of the table, so we skip that line.

74a ⟨*diff recipe* 74a⟩≡ (73g) 74c ▷
```
${GREP} -v "^.\?First \?name" ${out}.new | \
```

We are not interested in reporting students' failed results, so we filter out those. In `${RESULTS_FAILED}` we keep a regular expression for the grades that shall be removed. We then use `${RESULTS_FAILED_regex}` to match against the data.

74b ⟨*variables* 70⟩+≡ (71a) ◁ 73f 76b ▷
```
RESULTS_FAILED?=       -\|Fx\?\|U
RESULTS_FAILED_regex= "\( \|  \|,\)\"\?\(${RESULTS_FAILED}\)\"\?\(  .*\)*$$"
```

The code to filter the data will thus be:

74c ⟨*diff recipe* 74a⟩+≡ (73g) ◁ 74a 74d ▷
```
$(if ${RESULTS_FAILED},${GREP} -v ${RESULTS_FAILED_regex} |,) \
```

Finally, we want to compare this result with the old result and keep only the modified lines. For the convenience of the secretaries we will also sort the data on the third column (i.e. family name).

74d ⟨*diff recipe* 74a⟩+≡ (73g) ◁ 74c
```
${DIFF} ${@:.diff=} - | ${SED} -n "/^> /s/^> //p" | ${SORT} -k 3 > $@
```

Due to the difference comparison we require that the new file depends on the old file, and that the old file actually exists, so we add this as a dependency.

74e ⟨*targets* 71d⟩+≡ (71a) ◁ 73g 74f ▷
```
${out}.diff: ${out}
```

We will have to ensure that the old file exists, i.e. create it if it does not exist. We need it for comparisons, an empty file perfectly represents previously non-existent results, so we create a symbolic link to `/dev/null`.

74f ⟨*targets* 71d⟩+≡ (71a) ◁ 74e 74g ▷
```
${out}:
    [ -r $@ ] || ${LN} -s /dev/null $@
```

11.2.3 Extracting identifiers for reporting

In Moodle, every student is identified by a unique username. In the national database of grades, every student is uniquely identified by their civic registration number. What we want to do here is to extract the username from the exported data, then supply the corresponding civic registration numbers. We are only interested in the usernames of those students for whom the results changed, so we can use the `${out}.diff` file from above.

In this case we can use one of make's suffix-based constructions. We will take a file with suffix '.csv.diff' and create a file '.csv.diff.id'.

74g ⟨*targets* 71d⟩+≡ (71a) ◁ 74f 75c ▷
```
.SUFFIXES: .csv .csv.diff .csv.diff.id
.csv.diff.csv.diff.id:
    ⟨identifier recipe 75a⟩
```

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.

75a ⟨*identifier recipe* 75a⟩≡ (74g) 75b ▷

```
@echo "-- userids showed in ${PAGER} --"
${CAT} $< | ${CUT} -f 3 | ${PAGER}
```

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

75b ⟨*identifier recipe* 75a⟩+≡ (74g) ◁ 75a

```
@echo "-- paste username <tab> personnummer, end with C-d on a blank line (EOF) --"
${CAT} > $@
```

11.2.4 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.

75c ⟨*targets* 71d⟩+≡ (71a) ◁ 74g 75d ▷

```
.SUFFIXES: .csv .pdf
.csv.pdf:
  ${LOCALC} $<
```

Now we can add a target using this conversion.

75d ⟨*targets* 71d⟩+≡ (71a) ◁ 75c 75e ▷

```
${report:.csv=.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}`.

75e ⟨*targets* 71d⟩+≡ (71a) ◁ 75d 76c ▷

```
${report:.pdf=.csv}: ${in} ${out}.diff ${out}.diff.id
  ⟨report.csv recipe 75g⟩
```

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.

75f ⟨*clean recipe* 73b⟩+≡ (71e) ◁ 73b 76d ▷

```
${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.

75g ⟨*report.csv recipe* 75g⟩≡ (75e) 76a ▷

```
${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}.

76a ⟨*report.csv recipe* 75g⟩+≡ (75e) ◁75g

```
${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.

76b ⟨*variables* 70⟩+≡ (71a) ◁74b

```
RESULTS_COLUMNS?=   4
```

11.3 Sending and storing the results

Now we will do the actual reporting. As stated in Section 11.1 we have a target report for this purpose.

76c ⟨*targets* 71d⟩+≡ (71a) ◁75e

```
.PHONY: report
report: ${report} ${in}
```

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

76d ⟨*clean recipe* 73b⟩+≡ (71e) ◁75f

```
${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.

76e ⟨*report recipe* 76e⟩≡ (71d) 76f▷

```
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).

76f ⟨*report recipe* 76e⟩+≡ (71d) ◁76e

```
else \
  ${PAGER} ${report}; \
  ${RESULTS_MAILER} && \
  ${MV} ${out}.new ${out}; \
fi
```

Part VI

Appendices

Appendix A

MIUN-compatibility layer

A.1 Introduction

This entire makefile collection grew out of a set of generic makefiles for handling course material that I started to develop at Mid Sweden University (MIUN), starting back in 2011. In 2014 I started my PhD in KTH Royal Institute of Technology and I started to add some research oriented parts and by 2016 I refactored everything and rewrote it using literate programming. A lot of my material still depends on the original MIUN structure, the aim of this file is to map the old 'MIUN API' to the current one.

We will create a file ⟨*miun.compat.mk* 78⟩ which will implement the old API using the new. We let ⟨*miun.compat.mk* 78⟩ implement each old module as a code block.

78 ⟨*miun.compat.mk* 78⟩≡

```
ifndef MIUN_COMPAT_MK
MIUN_COMPAT_MK=true

⟨miun.subdir.mk 79a⟩
⟨miun.package.mk 79b⟩
⟨miun.pub.mk 80a⟩
⟨miun.export.mk 80b⟩
⟨miun.tex.mk 81⟩
⟨miun.docs.mk 82⟩
⟨miun.course.mk 83⟩
⟨miun.results.mk 84a⟩
⟨miun.depend.mk 84b⟩

endif
```

This way we can create separate files containing only the individual parts too. Then we do not have to modify old `Makefile`s to include `miun.compat.mk`.

A.2 miun.subdir.mk

The code to recurse through subdirectories is essentially the same, it has no API, so we simply include it using the recommended way.

79a ⟨*miun.subdir.mk* 79a⟩≡ (78)

```
ifndef MIUN_SUBDIR_MK
MIUN_SUBDIR_MK=true

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

endif # MIUN_SUBDIR_MK
```

A.3 miun.package.mk

We will base the old interface on revision 448 in the original internal MIUN repo, i.e. dated 2014-11-18 16:14:21Z. The current version of `pkg.mk` already maps the interface quite well, so we only have to map the last few parts and include the new include file.

79b ⟨*miun.package.mk* 79b⟩≡ (78)

```
ifndef MIUN_PACKAGE_MK
MIUN_PACKAGE_MK=true

ifdef TARBALL_NAME
PKG_TARBALL?=${TARBALL_NAME}.tar.gz
endif

ifdef DOCS_FILES
PKG_PACKAGES=  main docs

PKG_INSTALL_FILES-docs?=${DOCS_FILES}
PKG_INSTALL_DIR-docs?=${DOCSDIR}
endif

.PHONY: all
all: package

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

endif # MIUN_PACKAGE_MK
```

A.4 miun.pub.mk

We will base the old interface on revision 448 in the original internal MIUN repo, i.e. dated 2014-11-18 16:14:21Z. The current version of `pub.mk` already maps the interface quite well, so we only have to set the old defaults, map the last few variables and include the new include file.

80a ⟨*miun.pub.mk* 80a⟩≡ (78)

```
ifndef MIUN_PUB_MK
MIUN_PUB_MK=true

SERVER?=    ver.miun.se
PUBDIR?=    /srv/web/svn
CATEGORY?=
TMPDIR?=    /var/tmp
PUB_GROUP?= svn

ifdef NO_COMMIT
PUB_AUTOCOMMIT?=${NO_COMMIT}
endif

ifdef COMMIT_OPTS
PUB_COMMIT_OPTS?=${COMMIT_OPTS}
endif

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

endif # MIUN_PUB_MK
```

A.5 miun.export.mk

We will base the old interface on revision 287 in the original internal MIUN repo, i.e. dated 2013-01-21 23:27:17Z.

80b ⟨*miun.export.mk* 80b⟩≡ (78)

```
ifndef MIUN_EXPORT_MK
MIUN_EXPORT_MK=true

TRANSFORM_SRC=    .tex
TRANSFORM_DST=    .exporttex

TRANSFORM_LIST.exporttex=         NoSolutions
TRANSFORM_LIST-Makefile.export=   OldExportFilter ExportFilter

INCLUDE_MAKEFILES?=.
```

```
include ${INCLUDE_MAKEFILES}/transform.mk

endif # MIUN_EXPORT_MK
```

A.6 miun.tex.mk

We will base the old interface on revision 450 in the original internal MIUN repo, i.e. dated 2014-11-26 12:21:11Z. It is only the `submission` target that is not implemented from that version.

81 ⟨*miun.tex.mk* 81⟩≡ (78)

```
ifndef MIUN_TEX_MK
MIUN_TEX_MK=true

TEX_OUTDIR?=  .

TEXMF?=       ${HOME}/texmf

ifneq (${USE_LATEXMK},yes)
LATEX?=       latex
PDFLATEX?=    pdflatex
endif

ifneq (${USE_BIBLATEX},yes)
TEX_BBL=      yes
endif

solutions?=   no
handout?=     no

TRANSFORM_SRC=  .tex

ifeq (${solutions},yes)
TRANSFORM_DST+= .solutions.tex
TRANSFORM_LIST.solutions.tex=PrintAnswers

%.pdf: %.solutions.pdf
  ${LN} $< $@
endif

ifeq (${handout},yes)
TRANSFORM_DST+= .handout.tex
TRANSFORM_LIST.handout.tex=Handout

%.pdf: %.handout.pdf
```

```
  ${LN} $< $@
endif

.PHONY: all
all: ${DOCUMENTS}

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

endif # MIUN_TEX_MK
```

A.7 miun.docs.mk

We will base the old interface on revision 423 in the original internal MIUN repo, i.e. dated 2014-05-09 09:36:13Z.

82 ⟨*miun.docs.mk* 82⟩≡ (78)

```
ifndef MIUN_DOCS_MK
MIUN_DOCS_MK=true

DOCUMENTS?=
PUB_FILES?=   ${DOCUMENTS}
SERVER?=      ver.miun.se
PUBDIR?=      /srv/web/svn/dokument
CATEGORY?=

ifdef PRINT
LPR?=         ${PRINT}
endif

.PHONY: all
all: ${DOCUMENTS}

.PHONY: print
print: ${DOCUMENTS:.pdf=.ps}

.PHONY: clean-docs
clean-docs:
ifneq (${DOCUMENTS},)
  ${RM} ${DOCUMENTS}
endif

.PHONY: clean
clean: clean-docs
```

```
.PHONY: todo
todo: $(wildcard *)

INCLUDE_MAKEFILES?=.
include ${INCLUDE_MAKEFILES}/miun.tex.mk
include ${INCLUDE_MAKEFILES}/miun.pub.mk

endif # MIUN_DOCS_MK
```

A.8 miun.course.mk

We will base the old interface on revision 423 in the original internal MIUN repo, i.e. dated 2014-05-09 09:36:13Z.

83 ⟨*miun.course.mk* 83⟩≡ (78)

```
ifndef MIUN_COURSE_MK
MIUN_COURSE_MK=true

DOCUMENTS?=
PUB_FILES?=   ${DOCUMENTS}
SERVER?=      ver.miun.se
PUBDIR?=      /srv/web/svn/courses
CATEGORY?=

.PHONY: all
all: ${DOCUMENTS}

.PHONY: clean-course
clean-course:
ifneq (${DOCUMENTS},)
  ${RM} ${DOCUMENTS}
endif

.PHONY: clean
clean: clean-course

INCLUDE_MAKEFILES?=.
include ${INCLUDE_MAKEFILES}/miun.docs.mk
include ${INCLUDE_MAKEFILES}/miun.export.mk

endif # MIUN_COURSE_MK
```

A.9 miun.results.mk

We will base the old interface on revision 448 in the original internal MIUN repo, i.e. dated 2014-11-18 16:14:21Z.

84a ⟨*miun.results.mk* 84a⟩≡ (78)

```
ifndef MIUN_RESULTS_MK
MIUN_RESULTS_MK=true

in?=              ${COURSE}.txt
out?=              reported.csv
report?=          new_results.pdf

RESULTS_COURSE?=  ${COURSE}
RESULTS_EMAIL?=   ${EXPADDR}

MAILER?=  thunderbird -compose \
  "to=${EXPADDR},subject='resultat ${COURSE}',attachment='file://${report}'"
RESULTS_MAILER?=  ${MAILER}

REWRITES?=  "s/Godkänd(G)/G/g" "s/Underkänd(U)/U/g" "s/Komplettering(Fx)/Fx/g"
RESULTS_REWRITES?=${REWRITES}

FAILED?=  -\|Fx\?\|U
RESULTS_FAILED?=  ${FAILED}

FAILED_regex= " \(${FAILED}\)\( .*\)*$$"
RESULTS_FAILED_regex?=${FAILED_regex}

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

endif # MIUN_RESULTS_MK
```

A.10 miun.depend.mk

We will base the old interface on revision 464 in the original internal MIUN repo, i.e. dated 2015-05-26 12:51:03Z. The dependencies for Springer, ACM etc. has been removed.

84b ⟨*miun.depend.mk* 84b⟩≡ (78)

```
ifndef MIUN_DEPEND_MK
MIUN_DEPEND_MK=true

CONF?=  /etc/mk.conf
-include ${CONF}
```

```
.PHONY: dvips
ifeq (${MAKE},gmake)
dvips:
  which dvips || sudo pkg_add ghostscript
else
dvips:
  which dvips || sudo apt-get install texlive-full
endif

.PHONY: pdf2ps
ifeq (${MAKE},gmake)
pdf2ps:
  which pdf2ps || sudo pkg_add ghostscript
else
pdf2ps:
  which pdf2ps || sudo apt-get install texlive-full
endif

.PHONY: latex
ifeq (${MAKE},gmake)
latex:
  which latex || sudo pkg_add texlive_texmf-full
else
latex:
  which latex || sudo apt-get install texlive-full
endif

.PHONY: latexmk
ifeq (${MAKE},gmake)
latexmk:
  which latexmk || sudo pkg_add latexmk
else
latexmk:
  which latexmk || sudo apt-get install latexmk
endif

.PHONY: pax
ifeq (${MAKE},gmake)
pax:
  which pax
else
pax:
  which pax || sudo apt-get install pax
endif
```

```
.PHONY: sed
ifeq (${MAKE},gmake)
SED=  gsed
SEDex=  gsed -E
sed gsed:
  which gsed || sudo pkg_add gsed
else
sed gsed:
  which sed
endif

.PHONY: grep
ifeq (${MAKE},gmake)
GREP=     ggrep
GREPex=   ggrep -E
grep ggrep:
  which ggrep || sudo pkg_add ggrep
else
grep ggrep:
  which grep
endif

.PHONY: git
ifeq (${MAKE},gmake)
git:
  which git || sudo pkg_add git git-svn
else
git:
  which git || sudo apt-get install git git-svn
endif

.PHONY: wget
ifeq (${MAKE},gmake)
wget:
  which wget || sudo pkg_add wget
else
wget:
  which wget || sudo apt-get install wget
endif

.PHONY: localc
ifeq (${MAKE},gmake)
localc:
  which localc || sudo pkg_add libreoffice
else
localc:
  which localc || sudo apt-get install libreoffice
endif

.PHONY: update

update: update-rfc

.PHONY: rfc remove-rfc update-rfc clean-rfc

rfc: rfc.bib

remove-rfc::
  ${RM} -f ${TEXMF}/tex/latex/rfc.bib

update-rfc: remove-rfc ${TEXMF}/tex/latex/rfc.bib

.PHONY: clean-depends
#clean: clean-depends
clean-depends: clean-rfc
clean-rfc:
  ${RM} rfc.bib

${TEXMF}/tex/latex/rfc.bib: ${wget-depend}
  mkdir -p ${TEXMF}/tex/latex/
  wget -O - http://tm.uka.de/~bless/rfc.bib.gz 2>/dev/null | \
    uncompress - > ${@}

rfc.bib:
  if [ -e ${TEXMF}/tex/latex/rfc.bib ]; then \
    ln -s ${TEXMF}/tex/latex/rfc.bib rfc.bib ; \
  else \
    wget -O - http://tm.uka.de/~bless/rfc.bib.gz 2>/dev/null | \
    uncompress - > ${@} ; \
  fi


update: latexmkrc miun.tex.mk miun.course.mk miun.docs.mk
update: miun.export.mk miun.pub.mk miun.package.mk
update: miun.subdir.mk miun.results.mk

latexmkrc miun.tex.mk \
miun.course.mk miun.docs.mk miun.export.mk miun.pub.mk \
miun.package.mk miun.subdir.mk miun.results.mk:
  wget -O $@ http://ver.miun.se/build/$@

clean-depends:
  ${RM} latexmkrc miun.tex.mk miun.course.mk miun.docs.mk miun.export.mk
  ${RM} miun.pub.mk miun.package.mk miun.subdir.mk miun.results.mk

update: miunmisc miunart miunasgn miunbeam miunexam
update: miunlett miunprot miunthes

### MIUN Miscellanous package and Logo ###

miunmisc-depend?=   ${TEXMF}/tex/latex/miun/miunmisc/miunmisc.sty
logo-depend?=   ${TEXMF}/tex/latex/miun/miunmisc/MU_logotyp_int_sv.eps \
          ${TEXMF}/tex/latex/miun/miunmisc/MU_logotyp_int_CMYK.eps

${miunmisc-depend} ${logo-depend}:
  wget -O /tmp/miunmisc.tar.gz \
    http://ver.miun.se/latex/packages/miunmisc.tar.gz
  cd /tmp && tar -zxf miunmisc.tar.gz
  cd /tmp/miunmisc && ${MAKE} install

#.PHONY: miunmisc miunlogo
#miunmisc: ${miunmisc-depend}
#miunlogo: miunmisc

### MIUN Article class ###

miunart-depend?=  ${TEXMF}/tex/latex/miun/miunart/miunart.sty
${miunart-depend}:
  wget -O /tmp/miunart.tar.gz \
    http://ver.miun.se/latex/packages/miunart.tar.gz
  cd /tmp && tar -zxf miunart.tar.gz
  cd /tmp/miunart && ${MAKE} install

#.PHONY: miunart
#miunart: ${miunart-depend} miunlogo

### MIUN Assignment class ###

miunasgn-depend?=   ${TEXMF}/tex/latex/miun/miunasgn/miunasgn.sty
${miunasgn-depend}:
  wget -O /tmp/miunasgn.tar.gz \
    http://ver.miun.se/latex/packages/miunasgn.tar.gz
  cd /tmp && tar -zxf miunasgn.tar.gz
  cd /tmp/miunasgn && ${MAKE} install

#.PHONY: miunasgn
#miunasgn: ${miunasgn-depend} miunlogo
```

```
### MIUN Beamer class ###

miunbeam-depend?=   ${TEXMF}/tex/latex/miun/miunbeam/miunbeam.sty
${miunbeam-depend}:
  wget -O /tmp/miunbeam.tar.gz \
    http://ver.miun.se/latex/packages/miunbeam.tar.gz
  cd /tmp && tar -zxf miunbeam.tar.gz
  cd /tmp/miunbeam && ${MAKE} install

#.PHONY: miunbeam
#miunbeam: ${miunbeam-depend} miunlogo

### MIUN Exam class ###

miunexam-depend?=   ${TEXMF}/tex/latex/miun/miunexam/miunexam.sty
${miunexam-depend}:
  wget -O /tmp/miunexam.tar.gz \
    http://ver.miun.se/latex/packages/miunexam.tar.gz
  cd /tmp && tar -zxf miunexam.tar.gz
  cd /tmp/miunexam && ${MAKE} install

#.PHONY: miunexam
#miunexam: ${miunexam-depend} miunlogo

### MIUN Letter class ###

miunlett-depend?=   ${TEXMF}/tex/latex/miun/miunlett/miunlett.sty
${miunlett-depend}:
  wget -O /tmp/miunlett.tar.gz \
    http://ver.miun.se/latex/packages/miunlett.tar.gz
  cd /tmp && tar -zxf miunlett.tar.gz
  cd /tmp/miunlett && ${MAKE} install

#.PHONY: miunlett
#miunlett: ${miunlett-depend} miunlogo

### MIUN Protocol class ###

miunprot-depend?=   ${TEXMF}/tex/latex/miun/miunprot/miunprot.sty
${miunprot-depend}:
  wget -O /tmp/miunprot.tar.gz \
    http://ver.miun.se/latex/packages/miunprot.tar.gz
  cd /tmp && tar -zxf miunprot.tar.gz
  cd /tmp/miunprot && ${MAKE} install

#.PHONY: miunprot
```

```
#miunprot: ${miunprot-depend} miunlogo

### MIUN Thesis class ###

miunthes-depend?=   ${TEXMF}/tex/latex/miun/miunthes/miunthes.sty
${miunthes-depend}:
  wget -O /tmp/miunthes.tar.gz \
    http://ver.miun.se/latex/packages/miunthes.tar.gz
  cd /tmp && tar -zxf miunthes.tar.gz
  cd /tmp/miunthes && ${MAKE} install

#.PHONY: miunthes
#miunthes: ${miunthes-depend} miunlogo

endif # MIUN_DEPEND_MK
```

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 Miletić. *The Beamer class*. 3.36. Mar. 2015.