Self-Documenting Makefiles!

Trevor K. McAllister-Day

tkmday@uw.edu

June 30, 2016

Abstract

The purpose of this document is to describe the standards for creating self-documenting makefiles (SDMF). As projects grow more complex and branch off into multiple makefiles, keeping track of variable-settings and targets becomes increasingly difficult. Using SDMF allows you to compile the locations of the variable, targets, and files accessible to all your makefiles into a single place that is searchable; reducing possible collisions between variable and target names.

1 Introduction

1.1 Purpose

This script is designed to collect the following information from any number of relevant makefiles you point it at:

- **Top-level descriptions:** "Header information" usually found at the top of a makefile, descriptions like

```
#* Top level makefile
#* make all will recursively make specified targets in
    each subject directory.
```

- **Variables:** Defined through `VAR=x` syntax in the makefile preamble. Often references to directory paths (`bin/`, `incoming/`, etc), sometimes capture the output of `$(...)` command expansion.
- **Targets:** What's actually called from the command line, things like `PrepSubject`. These are collected from the definition of `.PHONY`.

Targets not set in `.PHONY` will be considered files, not targets.

- **Files:** Intermediate targets not called directly from the command line, something like `mprage/T1_brain.nii.gz`.

Additionally, Picnic recognizes directives such as `#*NODOC` which skips the entire makefile and `#*SKIP` which skips the relevant target.

1.2 Organization

I've improved the directory structure for picnic; currently it only requires three files:

- `picnic`
- `makemakedoc.py`
- `tables.tex`

These files can be moved to a directory like `~/local/picnic`, which you can add to your path, allowing you to call Picnic from everywhere.

2 Calling the script

There are two primary ways to document your makefiles.

You can give it explicit direction to the makefiles you would like to document:

```
$ picnic PrepSubject.mk subdir/anotherfile.mk
```

Or, you can give it a directory from which it will collect all makefiles (in that directory and all subdirectories):

```
$ picnic -D lib/makefiles
```

Using the `-D` syntax will include any makefile that ends in `*.mk` or matches the regex `*[Mm]akefile*`.

2.1 Skipping some files

Because it's possible you would like to not document all makefiles in the directory you have given (for example, a reference makefile), there are two ways to exclude some makefiles:

- The option `-e`, to which you can pass a regex. All files matching said regex *will not* be documented.
- It is also possible to include a directive in the makefile itself which will cause the documenter to skip it entirely: `#*NODOC`. Picnic will check the entire document for `#*NODOC`, and if it finds it anywhere, the makefile will be skipped.

2.2 Options

Aside from the flags `-D` and `-e` discussed above, there is also the standard `-v` flag (for verbose), which mostly stops silencing the output of PDFLaTeX, and `-h`, which outputs the standard help message. `-e` is not compatible with the first (non-`D`) syntax.

There is also the option `-o`, which allows you to specify the name of the output file. If it does not already end in `.pdf`, it will be `basename`'d and `.pdf` will be added.

All options must come before the file list or the directory. If you place options after the file list or directory, Picnic will hang indefinitely. This is a high-priority bug. In the mean time, make sure to call picnic like:

```
$ picnic -o foo.pdf foo.mk
```

not

```
$ picnic foo.mk -o foo.pdf
```

Summary table:

`-D`	Document all the makefiles in this directory.
`-e`	Exclude all files matching this regex. (`-D` only)
`-o`	Specify the output filename.
`-v`	Run in verbose mode.
`-h`	Display help and exit.

2.3 Naming makefiles

In general, LaTeX doesn't like filenames with underscores in them. Picnic will do its best to handle them, but I recommend avoiding underscores where possible.

3 Makefile commenting syntax

Because we have four things to identify from the makefiles (description/directives, variable, targets, and files), I introduce four comment "keywords" to facilitate information extraction, and for ease of use with other utilities such as `grep`.

All makefile comments begin with `#`, so SDMF comments will also begin with `#`. The four comment styles are: `#*`, `#!`, `#?`, and `#>`.

The parser (I believe) can handle most symbols, with `&` being replaced by `and` and semicolons being replaced by commas. Please let me know if any symbols have escaped my notice.

3.1 #, ##, ...

(Any number of only octothorpes.)
Only octothorpes without any SDMF control characters will be ignored, allowing use of makefile-only comments, section headers and the like.
Example: `### QA ###` will be completely ignored by the parser.

3.2 #*

This comment style is for descriptions at the top of the makefile, or header information. It is also used for directives like `#*NODOC` and `#*SKIP` (at this time, the only directives).

3.3 #!

These comments explain what a variable is for. The actual value of the variable is captured by the parser, and as such, it is not necessary to include the value itself in the comments.
Example:

```
# top level of the project directory
PROJECT_DIR=/mnt/home/adrc/ADRC
```

... will result in an entry like:

Variable	Definition & Description	File
PROJECT_DIR	/mnt/home/adrc/ADRC top level of the project directory	foo.mk

3.4 #?

These comments are for targets, and should take the form of one-line descriptions of targets to be called from the command line, e.g. `all`, `PrepSubject`.
Example:

```
#? Make all the relevant PCASL registrations
PCASL: pcasl/pcasl_fnirt.nii.gz pcasl/pcasl_M0_to_T1_Warped.
   nii.gz
```

Target comments can be multiple-lines (each prefaced with `#?`), but they really *shouldn't* be.

3.5 #>

File (intermediary target) descriptions, any number of `#>` comments can be used before an intermediate target. I don't *recommend* using more than two or three. Newlines will not be

preserved, and will be replaced with semicolons.
Example:

```
#> Register m0 to t1 using fnirt
pcasl/pcasl_fnirt.nii.gz: pcasl/Pcasl_skstrip.nii.gz mprage/
   T1_brain.nii.gz
```