Notes on MAKE

These are notes on reading Mecklenburg (2004) while rewriting the 14.1 facsheet make

"[...]you may notice that the order in which commands are executed by make are nearly the opposite to the order they occur in the *makefile*. This *top-down* style is common in *makefiles*. Usually the most general form of target is specified first in the details are left for later. The `make` program supports this style in many ways. Chief among these is `make`'s twophase execution model and recursive variables."[p.6] I think what he means here by *support* is *make possible*: so you can write it top down, but it will be built bottom-up, or whichever way is necessary, because it will be executed in two stages, so the first stage will make sure the order is correct for the second one.

A rule has, at a minimum, a target and a command. **If there are no prerequisites, then only the targets that do not exist yet are updated.**

`\# is the comment line`

\\: backslash escapes long lines.

Explicit rules

Can have more than one target, if they have the same prerequisites. "*if the targets are out of date, the same set of actions will be performed to update each one*"[p.11] - i.e. all targets are handled independently.

Wildcards

Make wildcards are identical to the Bourne shell's: `~, *, ?, [...]` and `[^...]`. so `*.*` expands to any file containing a period. `?` expands to any single character and `[...]` a character class, while `[^...]` is the opposite of the character class. aka *globbing*

There is something about wildcard expansion that i don't get yet: target and prerequisite expansion is perforemd by `make`, but wildcard expansion in the commands is performed by the subshell, which is sometimes important, and can be different.[p.12]

Phony targets

A target doesn't have to be a file, it can just be a label, to represent the command script you want to run. Phony targets will always be executed, because the target name will not exist, so the command will be run. But in case you have a file named `clean` or `all`, which would then stop the phony commands getting executed, you can explixityly declare a phony:

```
.PHONY : clean
clean:
  rm -f *.o
```

Phony targets are always out of date. Sometimes you want that: they will always be updated. But sometimes you don' - e.g. if it's a prerequisite, then the target will alwyas be compiled, even if it's cool.

Empty targets

Similar to phony, but here you use it to be performed only occasionally. So you have a rule who's target is an empty file–a cookie. not sure i get this quite..

Variables

all start with a `@`, but if they have more than one character, they are also in parenthesis.

Automatic variables

- `$@` - the target
- `$%` - the "filename element of an archive member specification"
- `$<` - the fileneme of the first prerequisite
- `$?` the names of all the newer prerequisites
- `$^` the names of all the prerequisites
- `$+` the names of all the prerequisites, including duplicates
- `$*` the target's stem, without the suffix - discouraged outside pattern rules

https://www.gnu.org/software/make/manual/make.html#index-_0024_0040

VPATH

Tells `make` where to search for files - instead of explicitly stating the directory each time. you can use `vpath` to be more precise in telling it which file patterns to search for where. e.g.

```
vpath %.eps fig
```

Tells it to look for .eps *targets or prerequisites* in the /fig folder. It only searched for targets and prerequisites, not for other e.g. variable definitions!

PATTERN RULES

Using the percent symbol, it can represent any number of characters (at least one), in any place in the pattern—but only once. `%.v` or `wrapper_%`

`make` first seaches for a matching pattern rule target. If it finds a match, it takes the pattern to be the stem, the `%`, Then it looks for prerequisites that have the stem in the correct place. If it exists, or can be made by applying another rule, the match is made and the rule is applied.

Static pattern rules

```
$(OBJECTS): %.o: %c
  $(CC) -c $(CFLAGE( $< -o $@
```

This is the same as a regular pattern rule, but it only applies to the matched files that are listed in the `$(OBJECTS)` variable. You use this whenever it is easier to list the target files explixitly, eather than to identify them by a suffix or other pattern. [.24]

Special targets

`.PHONY` are targets that are always built

`.INTERMEDIATE` if created, they will alwyays be deleted

VARIABLES AND MACROS

Two languages in one, a language describing targets and prerequisites, and a *macro* language for performing textual substitution, meaning you can define shorthand terms for longer sequences and use that in your program. The `make` macro processor will recognise them and replace them with the expanded forms. "A *macro variable* is expanded 'in place' to yield a text string that may then be expanded further"

Naming conventions:

No `#`, `:`, and `=`, case sensitive, single letter doesn't need parentheses, you can use curly or regular, doesn't matter.

The value of the variable trims leading spaces *but not trailing spaces!*

Variable types

Simply expanded

`:=` the right hand is evaluated (expanded) immediately when the file is read. (this is normal in most scripting languages) - if the variable has not yet been set it collapses to nothing (not an error)

OK, so I think I understand this a bit better now, if I add this to a knit-command variable, it doesn't work, since it expands it immediately, and therefore the `$@` are empty. (This should be a `define` statement anyway, but we'll get there in a moment).

Recursively expanded

`=` the right hand side is stored without evaluating or expanding until it is used. a.k.a. lazy expansion. So you can define the variable after you have used it.[p44] It also gets re-evalueated each time it is used

Conditional assignment

`?=` only assigns the value if it doesn't yet exist

Append

```
simple := $(simple) new stuff
```

This makes sense, because the simple variable is first expanded, then apended and then assinged. But with recursive variables it doesn't work: `recursive = $(recursive) new stuff` So `+=` is used to allow adding text to a recursive variable.

Macros

Can take multiple lines, a *canned sequence* = macro. using `define`.

```
define create-jar
 @echo Creating $@...
 $(RM)...
 ...test
endef
```

This is recursively expanded as well - which is obviously what you want, otherwise the $@ would be empty ;)

The `@` in front of the echo command means the commands are not echoed, so you only get the output. Apparently you can also suppress the whole macro's commands from being printed: `@$(create-jar)` But see command modifiers in chapter 5. Let me try this: either the command inside the macro or the whole macro? Same thing, no commands printed.

But `@echo` and then some text is really neat way of narrating the makefile, use it loads so that you know what is happening!

Also, not make related, but `render()` has a `quiet=TRUE` option that supresses printing of the pandoc command line.

Variable Rules

`make` runs in two phases:

1. reads the makefile and any included makefiles loads all the rules and variables into its internal database and creates a dependency graph . So in phase 1 a recursive variable or define macro is not expanded, just stored..:

- the left hand side of variable assingments are alwyas expanded in phase 1
- the righthand side of simple variable assignments `:=`
- the righthand side of simple append assignments `+=` IFF the left was originally a simple `:=` definition
- macros (`define`) the name is expanded immediately
- rules, targets and prerequisites are always immediately expanded

2. make analyses the dependency graph and determines which targets need to be updated and then executes the required command scripts (rules).

- the right hand side of recursive (lazy) variables (`=` and `?=`) are always deferred until phase 2.
- the righthand side of *recursive* append assignments `+=` is deferred until now if the left was originally a recursive `=` definition
- the macro body is only expanded when used
- commands are always deferred

So the rule is to always define variables and macros before they are used (even though lazy expansion allows you to do it later). But a variable used in a target or prerequisite must be defined before it is used.

OK, mini test:

```
1  all:
2      $(blabla)
3
4  define blabla
5   echo is this working?
6   touch $(var)
7  endef
8
9  DIR := .#
10 var := $(DIR)/figures/make.png
```

The only order that is important here is the order of the last two lines. the rest of it doesn't matter, because for both the commmand `$(blabla)` and the macro body the expansion is deferred until var has been expanded. But the last two (9 and 10) cannot be swapped, because they are simple assignmnents. But if you changed the var assingment to `=`, then you could place it anywhere. Because then the righthand side of a recursive assingmnent has the $(DIR) variable, but that is deferred until the second phase. This is funny, since I didn't get the difference until now. I kept trying to make the DIR assignment recursive, thinking it would mean it

would get assigned later. But that's not what happens. It is the expansion that happens later, and there is no expansion anywhere in that assignment. What we want to defer is the righthand side of var, that's the one that we want to be *lazy* and wait until we figure out what DIR is.

You're expanding the content of the variable, not the name

Target and Pattern specific Variables

"variables usually have only one value during the execution of a makefile" but what if you wanted to redefine a variable for one exception case?

Of course you could duplicate the code, but that's bad practice. So *target-specific variables*:

`target...: variable = value` and then the variable (doesn't need to exist beforehand) will be assinged the value *for the duration of the processing of the target and any of its prerequisites.* So only for the target `make` is currenlty building you can have a *local* variable, so not global.

But be careful, because it applies to the prereqs, but so if you have a file that is a prereq to two targets, and each has a different target-specific variable, then things could get tricky and out of hand quickly!

Getting variables into the makefile

- of course you can define them in the makefile directly - or in an included file
- you can also pass them from the command line, simply by inlcuding a `=` or `:=` assignment directly. This *overrides* any assignments made inside the file!

Conditional processing

The condition—based on which make decides whether or not to process the section—can be in the form of "is defined" or "is equal to" forms. As a general form you have this:

```
if-condition
 # if it is true
else
  # if it is false
endif
```

And possible if-conditions are

- `ifdef` vairable name or `ifndef` variable name, to test the (non)existance of a particular variable - where you don't use the $()
- `ifeq` and `ifneq` test which is expressed as `"a" "b"` or `(a,b)` (but careful with the whitespace!) One option to avoid issues with whitespace is to alwyasl use the `strip` function, at least when you are comparing variables, that might expand to include spaces: `ifeq "$(strip $(var))" "xx"`

You can use them inside rules, inside command scripts, macros, variable definitions..

Include

Usually for common definitions or sth from another file you can e.g.

`include definitions.mk`

Include can be a file that is only compiled with the makefile, which is neat: the first time `make` finds an include command but doesn't find the file, it will continue its pass, and if the rest of the file has instructions

for creating the file, it will pass it again and include it then. Only then, so if it cannot find the file, nor can it create it, will it exit with an error.

Standard `make` vairables

In addition to automatic variables, there are also a few other useful ones:

- `MAKE_VERSION`, useful because the `eval` and `value` functions weren't supported in 3.79, so you might want to check for that.
- `CURDIR` current directory, normally where the make file was invoked from.
- `MAKEFILE_LIST` a list of all the make files make has read and included. The last name on the list is the makefile's own name
- `MAKECMDGOALS` lists all targets for the current execution that were specified in the command line. I'm not clear on that example at all, but it's probably not a much needed thing.
- `.VARIABLES` contains all the variables (except target specific ones) although there's a pile here, not sure how to use that either.

Functions

Look like variable references but have one or more parameters separated by commas.

User-defined functions

I guess the difference with macros is that you can have parameters with a function. Variables and macros can be passed as arguments to the function.

`$(call macro-name[, param1...])`

So `call` is a built in make funciton that expands the first argument and replaces the occurences of `$1`, `$2` etch with the parameters given. (The macro doesn't have to have a `$n` reference, but then there's no point in using `call` then. Anyway, fairly unusually you use the macro-name here directly, not with `$()`.

Built-in funcitons

General form:

`$(function-name arg1[,argn])` where the leading whitespace gets trimmed for the first argument only.

String functions

`$(filter pattern...,text)`

Retruns a list of words from `text` that matches the pattern.

```
words := he the hen other the%

get-the:
    @echo he matches: $(filter he,$(words))
    @echo %he matches: $(filter %he,$(words))
    @echo he% matches: $(filter he%,$(words))
    @echo %he% matches: $(filter %he%,$(words))
```

But you can only use one % character at a time, which means you cannot match substrings within words, alhtough you can apparently do that with looping.

`$(filter-out pattern...,text)`

Is the opposite of filter, selecting all the ones that don't match.

`$(findstring string,text)`

Only returns the search string, not the word it finds it in. Second, the string cannot contain search strings. So def not a grep style function. But useful in conditional checking if sth is there.

`$(subst search-string,replace-string,text)`

Straightforward, find/replace cool for e.g. extensions and similar. But careful with the spaces! if there's a leading space in the replace-string, you'll get the spaces as well!

`$(patsubst search-pattern,replace-pattern,text)`

A wildcard version of `subst`, as usual with a single %.

Substitution references—not sure how they compare, but they are a 'portable way to perform the same substitution': `$(variable:search=replace)`. But it does work. But only if the search string is at the end of the word.

`$(words text)`

Returns the number of words in the text. (There seems to be no way to count the number of characters in a string). All of make is really operating with strings.

`$(word n, text)`

Returns the nth word in texxt. Combine words and word to get the last word in the text.

`$(firstword text)`

Returns the first word,

`$(wordlist start,end,text)`

Returns the words from start to end, inclusive.

Miscelaneous functions

`$(sort list)`

Sorts all the arguments and removes duplicates, and strips leading and trailing blanks.

`$(shell command)`

The output of the shell command is returned as the value of the function. Stripped of any errors.

Using all of these together..

For example, make doesnt understand numbers, only strings. so to compare two numbers you can use filter to compare them as strings.

Strings seem to be just pasted together automatically, e.g. `var := test-$(shell date +%F).tar.gz` becomes `test-2018-02-09.tar.gz`

Filename functions

`$(wildcard pattern(s)...)`

The other wildcards we discussed in the context of targets and prerequisites and commands. But what about e.g. for variable definitions?

This is a filename function!

So `$(wildcard *.Rmd)` should find all Rmd files in the folder. And you can add mulitple patterns here as well. And use the same globbing characters (`*`, `?`, `[...]` and `[^...]`)

Also good for testing if a file exists - if it doesn't the variable is empty.

```
md-exists = $(wildcard *.md)
```

`$(dir list...)`

Returns the dir portion of the files in list. Eg if you are looking for the folders that have `.Rmd` files: `md-exists = $(sort $(dir $(shell find . -name '*.Rmd')))`

`$(notdir name...)`

Returns the filename protion of a file path.

`$(suffix name...)`

Returns the suffix

`$(basename name...)`

Returns the file name without the suffix

`$(addsuffix suffix, name...)`

`$(addprefix prefix, name...)`

`$(join prefix-list, suffix-list...)`

Flow control functions

`$(if condtion, then-part, else-part)`

So this is different than the conditionla directives `ifeq`, `ifneq` , `ifdef` and `ifndef` that we discussed before. Why? I guess because you have more options for conditions: here the condition is true if it expands into any charactetrs at all, even just spaces. So if the expansion of the condition is empty, then the else part gets expanded.

`$(error message)`

Stops with the error message. An example is this test if you have the correct make version installed.

```
$(if $(filter $(MAKE_VERSION), 3.81)„$(error This is not cool))
```

`$(foreach variable,list,body)`

expand text repeatedly and substitute different values into each expansion.

```
letters := $(foreach letter,a b c d,$(letter))
show-words:
    # letters has $(words $(letters)) words: '$(letters)'
```

This works when you do `make show-words` but it's not explained why the comment is actually expanded? Ah, OK, because it is indented it is a command, which means it gets pushed to bash, which treats it as a comment—although i'm still not clear about why it expands the variables? `@echo` works just as well, perhaps less confusing.

But it just goes through the list and expands the body each time, which in this case is the variable `$(letter)` itself.

This second thing I got to work, but don't really understand.

```
VARIABLE_LIST := SOURCES OBJECTS HOME

$(foreach i,$(VARIABLE_LIST), \
    $(if $($i),, \
        $(shell echo $i has no value >test.txt )))
```

This next one worked: looks for the search-string in the word-list and returns the word

```
# $(call grep-str, search-string, word-list)
define grep-str
$(strip \
    $(foreach w, $2, \
        $(if $(findstring $1, $w), \
            $w)))
endef

words := count_words.c counter.c lexer.l lexer.h counter.h

find-words:
    @echo $(call grep-str,un,$(words))
```

Less important miscelaneous functions

`$(strip text)`

Removes all leading and trailing whitespace, and replaces all internal whitespace with a single space. Although I'm not sure it removes all the spaces!? Oh, it removes them if they are not in "".

`$(origin variable)`

Tests the origin of a variable. Can't get the example to work though

`$(warning text)`

Similar to error, but doesn't exit

Suffix rules

These are old rules but kept for backwards compatibility. I found one in the dapper invoice repository. Here's how it looks:

```
# this is adding .tex and .pdf to the list of suffixes which you can use
# in suffix rules - see it below
.SUFFIXES: .tex .pdf

# this is a suffix rule, telling make that every time you see a .tex file
# you can make a .pdf file using this rule:
# the rule seems to be to run xelatex twice on the $* which is the target's stem
.tex.pdf:
    xelatex $* && xelatex $*
```

Substitution references:

Also found this in the https://github.com/mkropat/dapper-invoice. This is an abreviation of a pattern substitution for compatibility. Another way of doing the same thing is with the $(addsuffix) command:

```
# this is a substitution reference, so take what is in $(REPORT) and add a .pdf suffix.
pdf: $(REPORT:%=%.pdf)
# this could be done instead with
pdf: $(addsuffix REPORT,.pdf)
# or
pdf: $(patsubst %,%.pdf,$(REPORT))
```

Advanced user defined functions

`call` functions bind the supplied parameters to the variables $1, $2, etc. the special case is that the name of the currenlty executing function i.e. the variable name is accessible through $0. Using this info we write a debugging function:

```
# $(debug-enter)
debug-enter = $(if $(debug_trace), $(warning Entering $0($(echo-args))))

# $(debug-leave)
debug-leave = $(if $(debug_trace), $(warning leaving $0))

comma := ,
echo-args = $(subst ' ','$(comma) ',$(foreach a, 1 2 3 4 5 6 7 8 9, '$($a)'))
```

Which worked fine, then inside a function you use `debug-enter` and `debug-leave` and it tells you where you are and what arguments (the first 9) are available

`eval` and `value`

`eval` is completely different to all built in functions. it's purpose is to feed text directly to the `make` parser.

Totally do not get the explanation here!

Hooking functions

You can make functions more reusable by adding hooks to them. A *hook* is a function reference that can be redefined by a user to perform their own custom tasks during a standard operation. Also don't get the explanation here.

Passing parameters

Nah, too complicated for my lady brain..

OK, mini-test

Let's see if I can fix the whole DHS factsheet:

1. `00-data-catalog.R`, this downloads the catalog, *temporarily* i here remove all the rows except for one since i don't want this thing taking long. and save it as an rds. This looks good. I am operating under the assumption that the order of rules is irrelevant. Let me test that. Yeah, doesn't matter. But I'll do it top down, so every new rule i'll add above the previous one.
2. The `01-import.R` file takes the catalog from step 1, cleans it up and usis it to download and extract the raw data which it then moves to the interim folder. In the meantime it also counts the number of cases, adds it to the catalog, which also goes to the interim folder.

The question is why do i split up the target `catalog.rds` and `%.rds`, is this necessary?

3. The `02-clean.R` file has even more dependencies: it needs the interim `%.rds` files as well as the UN codes and the `FunDataExtractor.R`. This file then outputs the processed `%.rds` files as well as the `catalog.csv`
4. Now same problem as before, the `%.rds` now don't work as a prerequisite for `02-clean`, and if i instead use `$(wildcard *rds)` then it seemst to work, but then it doens't work with the makefile to .dot code, because it has a space in the target/prerequisite, which splits it into two. But I solve this by declaring a new variable. `$(DT/I/.rds)`. This seems to work. So the `%` patters as targets, i need to avoid that completley.
5. OK, now the plotting, oddly if the final goal is `$(FIG/.eps)` it doens't work—there aren't any files—but if it's a sinlge `catalog.final.csv` file—that also doesnt' exixt—it does work
6. It seems that with multiple targets I can't ever get it to work right. Best option is to have an additional single target, and that one runs the code to produce the multiple targets—even if it is just a touch command.

AAH, OK, so seems that the single dependency lines without any recipe, they break the chain, you need to touch them, otherwise nothing happens. (Which is weird)

7. OK, the R side of things seems to be working, now the poster is LaTeX-dvips-ps2pdf, so here goes:

- `latex -interaction=nonstopmode --output-directory=$(@D) --aux-directory=$(@D) $<`
- `dvips  -Pdownload35 -o $@ $<`

- `ps2pdf %.ps`

These all worked OK, until it turned out that i forgot about bibtex, and realised that that was going to be a lot trickier because the .aux files are updated on every run of latex... apparenlty `latexmk` is what i should be using instead.

`latexmk` options:

- `latexmk -pdfps -use-make -c $<`
- `-pdfps` makes it use latex instead of pdflatex.
- `-use-make` I think you have to do if you're in make? sth do with missing files, not sure, but it didn't work without.
- `-c` cleans up temp files
- sth to define output dir?
- Then this suddenly stopped working... So i went back to plan 1.

OK, plan 1 works:

```
latex -interaction=nonstopmode --output-directory=$(@D) --aux-directory=$(@D) $<
bibtex $(basename $@)
latex -interaction=nonstopmode --output-directory=$(@D) --aux-directory=$(@D) $<
latex -interaction=nonstopmode --output-directory=$(@D) --aux-directory=$(@D) $<
```

Latex and bibtex, this works great. Now the proble is that a `a0header.ps` file was created in the folder and th next `dvips` run needs to find it, but can't, because it's in the presentations folder. So i'll solve that by temporarily moving it into the main folder and then deleting it. An alternative actually is the option `-h` which seems to let you specify the name of the header file? Whatever, it's working this way, i can't be botheredL:

```
cp docs/presentations/a0header.ps a0header.ps
dvips  -Pdownload35 -o -h docs/presentations/a0header.ps $@ $<
rm a0header.ps
```

OK, but looks like i still have the problem with mulitple targets. https://stackoverflow.com/questions/2973445/gnu-makefile-rule-generating-a-few-targets-from-a-single-source-file `$(subst .,%,$(varA) $(varB)): input.in`

And a very elaborate write up https://www.gnu.org/savannah-checkouts/gnu/automake/manual/html_node/Multiple-Outputs.html.. But the main risks here are academic, so maybe i can just get away with using a single .rds and a single .eps file as a clue. It seems like this is actually OK, it's a bit kludgy, but that's the best I can do atm.

Tips and tricks

- [https://www.r-bloggers.com/rstudio-and-makefiles-mind-your-options/] here explains the tabs are not spaces issue in RStudio. but this seems not to be true, like i have the "instert spaces" option turned on and it works fine?
- a good r/latex makefile tutorial http://augustogarcia.me/statgen-esalq/Makefile-Tutorial/
- [https://github.com/petebaker/r-makefile-definitions] - rules for R related stuff, so you wouldn't need to write recipes for them, they would be implicit rules.
- Need to come up with a trick to close the pdf file if it (invariably) is open..
- when renderring Rmd or R files - avoid using `out_dir` and or `output_file` instead compile to current directory and move later [[https://github.com/rstudio/rmarkdown/issues/861]]

TODO automatic variables `$(@D)` et al. # references

Mecklenburg, Robert. 2004. *Managing Projects with Gnu Make.* " O'Reilly Media, Inc."