Introduction to GNU Make

Anthony Scemama <scemama@irsamc.ups-tlse.fr>

Labratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)

Compiling

- Compiler : translates from a language to another (usually machine language)

`x.f90`, `x.c`, `x.ml`, …

Source file (writter by the user)

`x.mod`

Module file (in Fortran only)

`x.o`

Object file : Source file translated to binary

`x.a`

Archive. Static library.

`x.so`

Dynamic library.

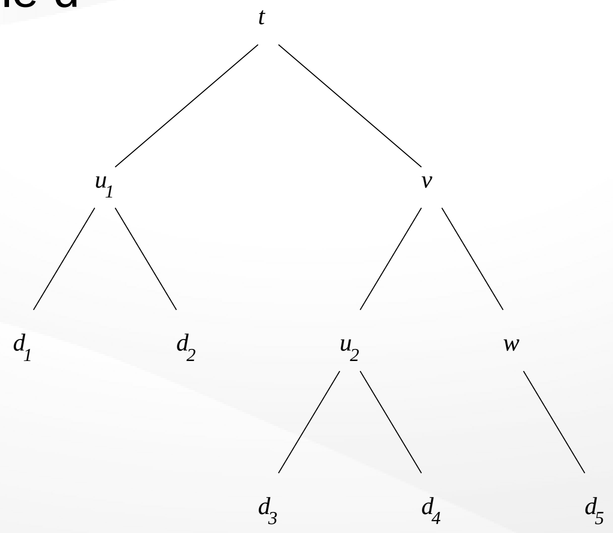
- Compiling : `x.f90` -> `x.o`. `x.o` contains only the code defined in `x.f90`.
- Library : Collection of `*.o` files "glued" together in one file.
- Linking : Assembling multiple object files and libraries in one executable file.

- Static library : Equivalent as a large `.o` file. Stored in the binary at link time
- Dynamic library : Only a pointer to the library is stored in the binary. The library is loaded at *execution* time.
  - Upgrading the library doesn't require to recompile the programs that need it
  - Sizes of executables are smaller
  - Simplifies licencing problems : binary files can be given without libraries

Example

Consider this program

- File `t.f90` uses module `u1` and `v`
- File `v.f90` uses module `u2` and `w`
- File `u1.f90` uses module `d`
- File `u2.f90` uses module `d`
- File `w.f90` uses module `d`



Make

- The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.
- You need a file called a *makefile* to tell make what to do.
- Rules are given in the makefile
- The compilation is done by running `make`
- The `-j N` option will use N processes to compile in parallel
- Internally a dependency tree of the files is built, and the last modification dates are accessed to check if a file needs to be re-built.

Rules

Rules have the following shape

```
target : prerequisites
        recipe
...
```

- `target` : Name of the file that is generated
- `prerequisites` : File needed to create the target
- `recipe` : Commands to create the target

Simple example:

```
my_program: main.o func1.o func2.o
        gfortran -o my_program main.o func1.o func2.o

main.o: main.f
        gfortran -c main.f

func1.o: func1.f
        gfortran -c func1.f

func2.o: func2.f
        gfortran -c func2.f

clean:
        rm my_program *.o
```

Make has *implicit* rules. These are default rules to build `.o` files from source file name extensions.

This works:

```
my_program: main.o func1.o func2.o
        gfortran -o my_program main.o func1.o func2.o
main.o: main.f
func1.o: func1.f
func2.o: func2.f

.PHONY: clean
clean:
        rm my_program *.o
```

(A phony target is one that is not really the name of a file)

Pattern rules can be defined, and this is the most common way to use make.

The `%` symbol represents the pattern to match. For example:

```
%.o: %.f
	gfortran -c $< -o $@
```

defines a rule to compile all files ending with `.f`. In this example, the automatic variables `$@` and `$<` are used to substitute the names of the target file and the source file in each case where the rule applies.

```
my_program: main.o func1.o func2.o
	gfortran -o my_program $^
%.o: %.f
	gfortran -c $< -o $@
.PHONY: clean
clean:
	rm my_program *.o
```

The automatic variable `$^` corresponds to the names of all the prerequisites, with spaces between them.

Variables

Variables make Makefiles simpler. Machine-dependent data can be defined in variables:

```
TARGET=my_program
F90=gfortran
F90_FLAGS=-O2
OBJ=main.o func1.o func2.o
RM=rm -f

$(TARGET): $(OBJ)
        $(F90) -o $(TARGET) $(OBJ)

%.o: %.f
        $(F90) $(F90_FLAGS) -c $< -o $@

.PHONY: clean
clean:
        $(RM) $(TARGET) $(OBJ)                                              .
```

Built-in functions can simplify variable definitions:

```
SRC=$(wildcard *.f)
OBJ=$(patsubst %.f, %.o, $(SRC))
```

- `$(wildcard *.f)` is equivalent to `*.f` in the shell.
- `$(patsubst %.f, %.o, $(SRC))` will return `$(SRC)` with all filenames finishing with `.f` substituted by `.o`.

All machine-dependent data can be moved into another file which will be included in the makefile:

`make.inc`:

```
TARGET=my_program
F90=gfortran
F90_FLAGS=-O2
RM=rm -f
```

Makefile :

```
include make.inc
SRC=$(wildcard *.f)
OBJ=$(patsubst %.f, %.o, $(SRC))

$(TARGET): $(OBJ)
	$(F90) -o $(TARGET) $(OBJ)

%.o: %.f
	$(F90) $(F90_FLAGS) -c $< -o $@

.PHONY: clean
clean:
	$(RM) $(TARGET) $(OBJ)
```