Physics 410: Using Fortran and C in the Unix Environment

Please report all errors/typos. etc to choptuik@physics.ubc.ca

Last updated October 2005

Index

INTRODUCTION AND OVERVIEW

The purpose of these notes is to provide you with the basic information required to produce executables (programs) from **Fortran** and **C** source files within a Unix environment. At least initially, I recommend that you do all of your **Fortran 77** (hereafter referred to as **Fortran**) and **C** programming on the **lnx** machines, using the **Portland Group** (abbreviated **PG** or **PGI**) compilers and the discussion below is *somewhat* specific to this particular environment. However, the same basic ideas and techniques should be applicable on any Unix systems. Again, don't be afraid to refer to the **man** pages!

Before proceeding to specific discussions of the Unix commands we will use for program development, let us consider the basic job of a compiler and work through some simple examples. A compiler translates (or compiles) "high-level" code (such as **C** or **Fortran**) into a form that the hardware can more or less run directly. In brief then, a compiler's job is to convert source code into executables. In the simplest case, the source code will reside in a single source file: on Unix systems, and *by convention*, **Fortran** source code is prepared in files whose names have a **.f** extension. Here is a simple example:

```
% ls *.f
first.f

% cat first.f
      write(*,*) 'Hello World!'
      stop
      end
```

Here the **Fortran** source file **first.f** contains a complete **Fortran** program. We can produce an executable using the **pgf77** command as follows:

```
% pgf77 -g first.f -o first
```

The **Fortran** compiler silently does its work, producing an executable called **first**

```
% ls -l first
-rwxr-xr-x  1 phys410 phys410 61891 Oct 11 06:16 first*
```

which, provided that `.` is in our current path, we can execute simply by typing its name:

```
% first
 Hello World!
```

Note that there is a leading space (blank) before the **Hello World!** text; this is typical of the output from **Fortran** programs and dates back to the dark ages when the standard output device was a line printer, and the first character in any output line was reserved for "carriage control".

The **pgf77** command issued above requires a little further explanation. In addition to the source file **first.f**, we supply as arguments the option **-g** (debug option) that tells the compiler to include information in the executable to facilitate debugging, and the option **-o first** that tells **pgf77** to name the executable **first**. Thus, there are effectively 3 arguments to **pgf77** in the above example: (1) **-g**, (2) **first.f**, and (3) **-o first**. These can appear in any order on the command line, so the following invocations (among others), are equivalent to our original form:

```
% pgf77 -o first -g first.f
% pgf77 first.f -o first -g
```

If you don't specify a name for the executable using the **-o** option, **pgf77** will call your executable **a.out**:

```
% pgf77 -g first.f
% a.out
 Hello World!
```

However, I strongly recommend that you avoid using this default behaviour.

Let us consider a slightly more complex example, in which we introduce the concept of an "intermediate" level of code---known as *object code*---that is also compiled from source code, but which is not directly executable. In this example we have two **Fortran** source files, **greeting.f** that contains a **Fortran** *main* program and **sayhello.f** that contains a **Fortran** *subroutine* (or procedure) that the main program calls:

```
% cd ~phys410/f77/ex1
% ls *.f
greeting.f   sayhello.f
% cat greeting.f
c     This is the main program

      program  greeting

      call sayhello()

      stop
      end

% cat sayhello.f
c     This is the subroutine

      subroutine sayhello()
         write(*,*) 'Hello World!'
         return
      end
```

As in our previous example, we can generate an executable directly using the **pgf77** command; we simply pass *both* source files as arguments:

```
% pgf77 -g greeting.f sayhello.f -o greeting
greeting.f:
sayhello.f:
```

Note that this time, the **pgf77** command echoes the name of each source file (followed by a colon) as it is processed. Also note that the name of the executable produced is **greeting**:

```
% greeting
 Hello World!
```

If we check the contents of the directory:

```
% ls
Makefile  greeting*  greeting.f  greeting.o  sayhello.f  sayhello.o
```

we notice that in addition to the executable, **greeting**, the **pgf77** command created two files, **greeting.o** and **sayhello.o**, both having **.o** extensions. These are *object* files that, as mentioned above, we can view as an "intermediate" level between source code and executable code. Loosely speaking then, the process of translating source code into executable code in Unix can be separated into two phases:

- The *compilation* of source code into object code
- The *linking* of object code (including code in *library archives*) to produce an executable.

Without going into too much detail, the linking phase involves assembling the main routine and various subroutines (or procedures), which constitute a program, to produce an executable.

Although there is a separate linking command in Unix (usually called **ld**), you don't have to invoke it directly---the **pgf77** (or **pgcc**) command will do it for you, *provided you have issued a command that calls for the creation of an executable*. Such is the case in the two examples above where we generally used the **-o** option. However, we can also use **pgf77** to create an executable in two phases. First, by supplying the **-c** option to the compiler we request that **.f** files *only be compiled* into **.o** files:

```
% RM greeting *.o
% ls
Makefile     greeting.f   sayhello.f
```

```
% pgf77 -g -c greeting.f sayhello.f
greeting.f:
sayhello.f:

% ls
Makefile     greeting.f   greeting.o   sayhello.f   sayhello.o
```

Note that this last **pgf77** command did *not* create an executable. To make the executable, we supply **pgf77** with the names of the **.o** files that contain the object code that we wish to be "linked" to create an executable, and, as in our early examples, we use the **-o** option to give the executable a specific name:

```
% pgf77 -g greeting.o sayhello.o -o greeting

% ls
Makefile     greeting*    greeting.f   greeting.o   sayhello.f   sayhello.o

% greeting
 Hello World!
```

Here, **pgf77** basically passes all of its arguments to the actual loader command, **ld**, along with additional information for **ld** that is common to all **Fortran** programs. The loader then creates the executable.

Although the two phase process of first creating object files using the **-c** option, and then linking them together to create an executable file may seem awkward, there are advantages to this technique. For example, if we are working with a program consisting of many thousands of lines of source code contained in many distinct source files, and make a change to *one* of the source files, then by using separate compilation and link phases, we need only recompile (using **-c**) the single source file that was modified, then relink *all* of the object files to produce a new executable. For large programs this can significantly decrease the development cycle-time and hence is recommended practice. *However, for short programs, and in particular for programs that are entirely contained in a single source file, the first approach used above will suffice.*

We end this survey with a brief discussion of *libraries* (or *library archives*) in Unix. Libraries are closely related to object code, and you can think of them as collections of routines (procedures, functions) that have been converted into object code and that are ready to be included (linked to) by any program that wants to call them. A simple example will illustrate the idea:

```
% cd ~phys410/f77/ex0
% cat tdmach.f
c-----------------------------------------------
c    Simple illustration of use of libraries
c
c    dmach is a function defined in the LINPACK
c    library
c
c    /usr/local/PGI/lib/liblinpack.a
c
c    It computes and returns information
c    concerning the invoking machine’s floating
c    point  model.
c-----------------------------------------------
     program    tdmach

     implicit   none

     real*8     dmach

     write(0,*) dmach(1)

     stop
     end
```

This simple program invokes the function **dmach** to compute machine-epsilon (a concept we will discuss at a later date), and then outputs the value to standard error. However, **dmach** *is not* a basic part of the **f77** language. Thus, if we naively try to build an executable, we get an "undefined reference" error message:

```
% pgf77 -g tdmach.f -o tdmach
... /pgf77baaaaAAaaw.o(.text+0x52): In function ‘tdmach’:
/home/phys410/f77/ex0/tdmach.f:19: undefined reference to ‘dmach_’
```

which tells us that the loader, **ld**, (again, automatically invoked here by the **pgf77** command) was unable to locate object code for *any* routine named **dmach_**).

The fact that the loader tells us that it can't find a definition for **dmach_** rather than **dmach** requires a little digression. One can write programs in a variety of "high-level" languages: **C**, **C++**, **Fortran 77**, **Java** ..., and it is often convenient to be able to combine pieces of code written in *different* languages into a single executable. One problem that we wish to avoid in such "mixed mode" programming is the "collision" of the names of routines that are written using different languages, then subsequently linked together by the loader. For example, it should be possible for a programmer to write a subroutine named **foo** in **Fortran** , and a function also named **foo** in **C**, and have the loader be able to distinguish between them should they both be included in the same executable.

To make a longish story relatively short, the way most **Unix** compiler suites, including the **PG** compilers, deal with this naming conundrum is to append an underscore to the name of any **Fortran** symbol that is to be communicated to the "external world"; i.e. to the loader. **C** external names, on the other hand, retain their source-code-defined identity, which is only reasonable since the vast bulk of any respectable Unix implementation is likely to be coded in **C**.

We now return to the failed **pgf77** invocation. To fix the problem, we need to add some additional options and arguments that are used in the linking phase of executable generation:

```
% pgf77 -g -L/usr/local/PGI/lib tdmach.f -llinpack -o tdmach
```

This time the link phase succeeds, so we can now execute **tdmach**:

```
% tdmach
   1.0842021724855044E-019
```

We will discuss libraries in a little more detail below. Here I will only point out that the rather cryptic options **-L/usr/local/PGI/lib** and **-llinpack**, combined with the fact that a file with the name

```
liblinpack.a
```

exists in the directory

```
/usr/local/PGI/lib
```

is sufficient to make things work. More specifically, **-llinpack** tells the loader that it should search for a file named

```
liblinpack.a    or    liblinpack.so
```

in one of the "standard" directories where libraries are maintained on the system, while **-L/usr/local/PGI/lib** tells the loader to append the directory **/usr/local/PGI/lib** to the list of "standard" directories. Armed with this information, the loader finds the library archive **/usr/local/PGI/lib/linpack.a**, inspects it, detects that it contains object code for a routine named **dmach_**, and links that object code into the executable. The search that the loader performs for a specific library archive is very much analogous to the resolution of names of commands using the path.

COMPILING AND LINKING FORTRAN PROGRAMS: pgf77

Use the **pgf77** command to compile and link **Fortran** programs.

USAGE EXAMPLES

The following command compiles and loads **pgm.f** with the debug option, creating the executable **pgm**.

```
% pgf77 -g pgm.f -o pgm
```

The first of the following commands compiles **main.f** and **subs.f** producing object files **main.o** and **subs.o**. The second command loads both object files creating the executable **main**.

```
% pgf77 -g -c main.f subs.f
% pgf77 -g main.o subs.o -o main
```

The following example is the same as the previous one, except that we now link to the library **/usr/local/PGI/lib/libp410f.a** using the **-L** and **-l** options

```
% pgf77 -g -c main.f subs.f
% pgf77 -g -L/usr/local/PGI/lib main.o subs.o -lp410f -o main
```

Note that **-L/usr/local/PGI/lib** adds the *directory* **/usr/local/PGI/lib** to the default search path the loader uses when searching for library archives. The option **-lp410f** tells the loader which specific archive it is seeking. Observe that the strings **lib** and **.a** are *automatically* pre- and post-pended, respectively, to create the actual filename of the archive (**libp410f.a** in this case).

USEFUL pgf77 OPTIONS

See **man pgf77** for additional information. Note that compiler options tend to be system-specific. Options similar to those described here should be available on most Unix **Fortran** implementations.

- **-g** Debug option. Required if you want to use **pgdbg** for program debugging.
- **-Mbounds** Enables array bound checking. Highly recommended for program development. Disable for production work.
- **-fast** Perform aggressive code optimization. Recommended for production work, after code has been thoroughly debugged and tested.

COMPILING AND LINKING C PROGRAMS: pgcc

Use the **pgcc** command to compile and link **C** programs

USAGE EXAMPLES

The following command compiles and loads **pgm.c** with the debug option, creating the executable **pgm**.

```
% pgcc -g pgm.c -o pgm
```

The first of the following commands compiles **main.c** and **routines.c**, producing object files **main.o** and **routines.o**. The second command loads both object files creating the executable **main**.

```
% pgcc -g -c main.c routines.c
% pgcc -g main.o routines.o -o main
```

The following example is the same as the previous one, except that we now link to the libraries **/usr/local/PGI/lib/libp410f.a** and **/usr/lib/libmath.a** using the **-L** and **-l** options

```
% pgcc -g -c main.c routines.c
% pgcc -g -L/usr/local/PGI/lib routines.o subs.o -lp410f -lm -o main
```

Note that when multiple libraries are specified in the load phase, as they are above, the loader searches each library *exactly once* for unresolved symbols and searches in the order specified on the command line. Thus, if the program above calls a routine in the **p410f** library, and that routine calls a routine in the **m** library (standard **C** math support) then

```
% pgcc -g -L/usr/local/lib routines.o subs.o -lm -lp410f -o main
```

will result in a load error.

#include'ing FILES FROM NON-STANDARD LOCATIONS

Statements of the form

```
#include "mytypes.h"
#include <stdio.h>
```

are **C** pre-processor directives that effectively include the contents of a file in-place in the **C** source. In the first form, where the filename is enclosed in double quotes ("), the specified file must reside in the working directory. In the second case, where the filename is enclosed in angle-brackets (**<>**), the preprocessor searches for the file in the ‘‘standard include directory’’, **/usr/include**. Additional directories that are to be searched for **#include** files may be specified with the **-I** option. Thus, assuming that **mytypes.h** lives in **/home/matt/include** and that the source code for **myinclude.c** contains the statement

```
#include <mytypes.h>
```

then the **pgcc** command

```
% pgcc -I/home/matt/include myinclude.c -o myinclude
```

will ensure that the file is properly included.

USEFUL pgcc OPTIONS

See **man pgcc** for additional information. Note that compiler options tend to be system-specific. Options similar to those described here should be available on most Unix **C** implementations.

- **-g** Debug option. Recommended if you want to use **pgdbg** for debugging.
- **-fast** Perform aggressive code optimization. Recommended for production work, after code has been thoroughly debugged and tested.

USING AND CREATING LIBRARIES

Libraries (library archives) in Unix have, by convention, names that begin with **lib** and end with **.a**:

```
% cd /usr/lib
% ls lib*.a
libImlib.a         libfl.a            libimlib-bmp.a    libpcreposix.a
libSDL.a           libfontconfig.a    libimlib-gif.a    libpthread_nonshared.a
libSDL_gfx.a       libform.a          libimlib-jpeg.a   libreadline.a
libSDL_image.a     libg.a             libimlib-png.a    librpcsvc.a
                            .
                            .
                            .
```

Link to libraries located in standard locations (notably **/lib** and **/usr/lib**) using the **-l** option to either **pgf77** or **pgcc**:

```
% pgcc  -cpgm.o  -lm -lX11 -o cpgm
% pgf77 f77pgm.o -lblas -o f77pgm
```

Use the **-L** option to prepend a directory to the default search path for library archives. Thus, assuming that I have a library named **/home/matt/lib/libvutil.a**, the following **pgf77** command will link (if necessary) to that archive:

```
% pgf77 -L/home/matt/lib pgm.o -lvutil -o pgm
```

CREATING LIBRARIES

Create and maintain library archives using the **ar** (archive) command.

Typically one creates an archive file from one or more object files. Thus assuming that the following object files reside in the working directory

```
% ls *.o
procs1.o procs2.o procsio.o
```

then the following **ar** command will create or overwrite a library archive file **libmylib.a** containing all routines defined by the 3 object files and will ensure that the archive has a ‘‘table-of-contents’’ as required by the loader:

```
% ar r libmylib.a procs1.o procs2.o procsio.o
```

Note that the **r** immediately following **ar** in the above is an option (replace) to the **ar** command: i.e. **ar** options do *not* have to begin with a minus sign. Also note that on some systems, **ar** will not automatically add a table of contents. In such cases there is usually a command **ranlib** that will do the job.

See **man ar** for more information.

DEBUGGING PROGRAMS: pgdbg

- Click **HERE** for more information (eventually!)

ORGANIZING AND AUTOMATING PROGRAM BUILDS: make

The **make** program (utility) is primarily used to organize and automate compilation and linking of programs. By convention, given a directory containing source code that is to be compiled and linked to produce one or more executables, input for **make** is prepared in a file named **makefile** or **Makefile** in that directory---I will tend to use the latter convention in this course.

The basic idea behind **make** is to view executables (compiled from **Fortran 77** source, e.g.) as **targets** that, in general, have one or more **dependencies**. Dependencies are typically files themselves, and may themselves have (further) dependencies. This view captures the notion, for example, that an executable **foo** is constructed from (depends on) the object file **foo.o**, which in turn is constructed from (depends on) the source file **foo.f**. A makefile, then, generically consists of definitions of one or more targets; equivalently, things to build, or things to ’make’. Each target definition usually consists of

1. The name of the target (typically the name of an executable file)
2. A list of dependencies, possibly null (again, usually a list of files on which the target depends)
3. A set of actions (generally Unix commands, and specifically, compilation and loading commands) that define precisely how the target is ’made’

Each target and its dependencies are separated by a colon (**:**), *and must appear on the same line.* The set of commands that defines how the target is made follows on subsequent lines, each of which *must begin with a TAB character*. Here is a simple example (**Makefile** fragment) that says that (executable) **foo** depends on **foo.f**, and then defines how to make the executable from the source file using a *single* **pgf77** command.

```
foo: foo.f
        pgf77 -g foo.f -o foo
 ^
 |-------- TAB character
```

make presumes that once something is made, it doesn't have to be re-made until one of the dependencies changes: whether or not a particular dependency (file) has or has not been changed since a previous make can be deduced via the time of last modification of the file, a statistic that Unix maintains for all files.

(Observe that the idea here is to provide a mechanism to ensure that executables and the like *are* up to date, so that we are less likely to change some small piece of the code and forget to rebuild the application. We are less concerned about "wasted" work that results from re-making something that really didn't have to be re-made, which might happen, for example, if we edited a file, made a change, saved the file, then undid the change, and re-saved the file. In such an instance, the time of last modification of the file is the time of the last save, even though one might be inclined to view the file as *not* having been modified.)

Before proceeding to an examination of a **Makefile** typical of the sort that will be used in the course (and with which, therefore, you should become familiar!), we need to discuss three important preliminary points:

Important point 1: I re-emphasize that all command-lines that define how a particular target is made **MUST begin with a TAB character**: be especially careful if you "cut and paste" parts of makefiles (such as those reproduced below) into you own---the cut will generally "convert" the TAB to spaces, and you will *NOT* end up with a valid makefile. Whenever you see a message such as

```
Makefile:12: *** missing separator.  Stop.
```

it's a safe bet that at or near line 12 (in this example), there's an action (command) line that begins with something other than a TAB!

Important point 2: In order to keep our makefiles as portable as possible (so that, as much as possible, we can use the same makefile on different machines running different versions of Unix), we will use certain specific **environment variables** to communicate with **make**. One of the nice features of **make** is that it automatically inherits all environment variables defined at the instant the **make** command is issued as **make** "macros", which, for the purposes of this course we can view as **make** "local variables". We simply have to keep in mind the slightly peculiar syntax of evaluation of variables (macros) in **make**. So for example, to access the value of the **HOME** environment variable in a makefile, we use the construct

```
    $(HOME)
```

rather than the form

```
    $HOME
```

that we use in the shell.

To set, or define, a makefile macro, we use the following syntax

```
<macroname>  = <macro definition>
```

where **<macroname>** should adhere to the "usual" rules for an identifier (variable) name, and where **<macro definition>** is an *arbitrary* string (do *not* enclose the definition string in quotes, even if it contains white space.) Here is a simple example that defines a macro **MYLIB** in terms of the pre-defined (inherited from the environment) macro, **HOME**:

```
MYLIB = $(HOME)/lib
```

Assuming, for concreteness, that **$(HOME)** evaluates to **/home/matt**, **$(MYLIB)** will then evaluate to **/home/matt/lib**.

The specific environment variables that we will use (as pre-defined macros for **make**), along with their recommended settings on the **lnx** machines are as follows:

```
# 'F77' defines the name of the Fortran 77 compiler.
setenv  F77        'pgf77'

# 'F77FLAGS' defines options that are used in BOTH the compilation and load phases
setenv  F77FLAGS   '-g'

# 'F77CFLAGS' defines flags that are used ONLY in the compilation phase
setenv  F77CFLAGS  '-c'
```

```
# 'F77LFLAGS' defines flags that are used ONLY in the load phase
setenv  F77LFLAGS  '-L/usr/local/PGI/lib'

# 'LIBBLAS' defines the name of the BLAS (Basic Linear Algebra Software) library
setenv  LIBBLAS    '-lblas'
```

Recommended settings of the above variables on a variety of machines used in the course can be found **HERE**.

Note that these variables should generally be set in your **.cshrc** (or equivalent) so that

1. They will be automatically set every time you create a new shell.
2. *All* of your **Makefile**'s will inherit the correct macro settings.

On the **lnx** machines, I have modified the system-wide **tcsh** start-up file, **/etc/csh.cshrc** so that the variables *are* set as above; i.e. you do *not* have to add anything to your **.cshrc**.

Important point 3: If you want to "break" a long line in a makefile over two or more lines, you can use the "usual" shell "continuation construct" ("usual" since it also works in the **csh**, **tcsh**, **bash** etc.), which is simply a backslash (\) *immediately* followed by a new-line. Thus, in a makefile, the macro assignments

```
FILELIST = a.f b.f c.f
```

and

```
FILELIST = a.f \
           b.f \
           c.f
```

are identical.

Note again that there can be **NO** white space following the \ character, or **make** will become confused.

Given the above preamble, we can now turn to an example **Makefile** that can be found in the directory **~phys410/f77/ex3** on the **lnx** machines. (Observe, however, that the on-line version has been stripped of the comments included in the version below).

```
############################################################
# Lines beginning with a '#' are comments
############################################################

############################################################
# The .IGNORE: directive tells 'make' to keep going if
# one or more commands executed as a result of the 'make'
# do not complete successfully.  The default is to bail
# out.
############################################################
.IGNORE:

############################################################
#  Note that this 'Makefile' assumes that the following
#  environment variables are set:
#
#     F77         ->  name of f77 compiler
#     F77FLAGS    ->  generic f77 flags
#     F77CFLAGS   ->  f77 flags for compilation phase
#     F77LFLAGS   ->  f77 flags for load phase
############################################################
# Define some macros for
#
#    F77_COMPILE:  The command which will be used to compile
#                  Fortran source
#    F77_LOAD:     The command which will be used to load
#                  Fortran object files, link to libraries
#                  and create executables.
#
# In this Makefile, macros are used almost precisely like shell
# or environment variables.  Note, however, that macros are
# evaluated with the $(MACRONAME) construct: the () are CRUCIAL.
# Also note that ALL environment variables (HOME, DISPLAY,
```

```
# etc.) are automatically available as macros with the
# same name.  Thus, for example, $(HOME) will evaluate to
# your home directory.  We will use this feature to create
# Makefiles which are portable across systems provided that
# the appropriate environment variables are set properly
# (typically in your '~/.cshrc') on the various systems.
############################################################
F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS)
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)

############################################################
# The following defines a GENERIC target (rule)
# which tells 'make' how to produce a '.o' file from
# a '.f' file.  'Make' will automatically use such a rule
# unless a specific target overrides it.
############################################################
.f.o:
	$(F77_COMPILE) $*.f

############################################################
# Define a macro for all the executables in the directory
############################################################
EXECUTABLES = fdemo2 mysum tdvfrom tdvto

############################################################
# Since this is the first SPECIFIC target in the makefile,
# if 'make' is invoked with no arguments, this is the target
# which will be made.  Since $(EXECUTABLES) evaluates to
# 'fdemo2 mysum tdvfrom tdvto', 'make' will make each of
# 'fdemo2', 'mysum', 'tdvfrom' and 'tdvto' in turn
############################################################
all: $(EXECUTABLES)

############################################################
# The target 'fdemo2' depends on the object file 'fdemo2.o'.
# When 'fdemo2' is being made, 'make' figures out that it
# first needs to make 'fdemo2.o' from 'fdemo2.f' using
# the generic rule above.  Once the dependencies of any
# given  target have been updated, the commands which
# follow the target are executed in turn.  In this case,
# the object file is simply loaded and the executable
# 'fdemo2' is created.  Again note that each command line
# MUST BEGIN WITH A TAB.  Continue long lines with \
# (backslash, followed by carriage return, with no
# intervening spaces).
############################################################
fdemo2: fdemo2.o
	$(F77_LOAD) fdemo2.o -o fdemo2

mysum: mysum.o
	$(F77_LOAD) mysum.o -o mysum

############################################################
# A little more complicated example since there are 2
# dependencies ('tdvfrom.o' and 'dvfrom.o') and we
# link to the 'p410f' library
############################################################
tdvfrom: tdvfrom.o dvfrom.o
	$(F77_LOAD) tdvfrom.o dvfrom.o -lp410f -o tdvfrom

tdvto: tdvto.o dvto.o
	$(F77_LOAD) tdvto.o dvto.o -lp410f -o tdvto

############################################################
# Makefiles often have a 'clean' target which cleans
# up object files, executables and other files which
# tend to consume precious disk space, and which can
# always be reconstructed (via 'make' of course!)
############################################################
```

```
clean:
        rm *.o
        rm $(EXECUTABLES)
```

Here’s the same example with the comments removed:

```
.IGNORE:

F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS)
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)

.f.o:
        $(F77_COMPILE) $*.f

EXECUTABLES = fdemo2 mysum tdvfrom tdvto

all: $(EXECUTABLES)

fdemo2: fdemo2.o
        $(F77_LOAD) fdemo2.o -o fdemo2

mysum: mysum.o
        $(F77_LOAD) mysum.o -o mysum

tdvfrom: tdvfrom.o dvfrom.o
        $(F77_LOAD) tdvfrom.o dvfrom.o -lp410f -o tdvfrom

tdvto: tdvto.o dvto.o
        $(F77_LOAD) tdvto.o dvto.o -lp410f -o tdvto

clean:
        rm *.o
        rm $(EXECUTABLES)
```

and here’s some output from **make** generated using this makefile:

```
% make
pgf77 -g -c fdemo2.f
pgf77 -g -L/usr/local/PGI/lib fdemo2.o -o fdemo2

pgf77 -g -c mysum.f
pgf77 -g -L/usr/local/PGI/lib mysum.o -o mysum

pgf77 -g -c tdvfrom.f
pgf77 -g -c dvfrom.f
pgf77 -g -L/usr/local/PGI/lib tdvfrom.o dvfrom.o -lp410f -o tdvfrom

pgf77 -g -c tdvto.f
pgf77 -g -c dvto.f
pgf77 -g -L/usr/local/PGI/lib tdvto.o dvto.o -lp410f -o tdvto
```

Notice how **make** "echoes" each command (action) to standard output as it is executed, and also note that I’ve added an occasional blank line in the above for readability.

Since the first specific target in the makefile is **all**, the above invocation is equivalent to

```
% make all
```

The Unix **touch** command simulates modification of its file arguments by setting the last-modified time of its arguments to the current time. Thus having previously made everything,

```
% touch dvto.f
% make
pgf77 -g -c dvto.f
pgf77 -g -L/usr/local/PGI/lib tdvto.o dvto.o -lp410f -o tdvto
```

we see that **make** re-makes only those targets that depend on the single modified file. Note that we can easily make a single target by supplying the target as the sole argument to **make**:

```
% make clean
```

```
rm *.o
rm fdemo2 mysum tdvfrom tdvto

% make fdemo2
pgf77 -g -c fdemo2.f
pgf77 -g -L/usr/local/PGI/lib fdemo2.o -o fdemo2
```

Finally if **make** deduces that a target is up to date, it will generally tell you so:

```
% make fdemo2
make: `fdemo2' is up to date.
```

See **man make** or the suggested Unix references for more information.
