Writing a Fortran 90 Makefile Maker in Perl

Michael Wester
Cotopaxi, 1801 Quincy, SE
Albuquerque, New Mexico 87108, USA
wester@math.unm.edu

One of the more useful tools for developing and maintaining large programs is the make utility [Tal89]. make allows for the easy compilation of programs split up into multiple source files. It does this by following the directives in a user created file named "Makefile" (or "makefile") which specifies what object files and libraries are needed to create the program executable, what source files are needed to create each object file, perhaps what object files are needed to create the libraries, etc. These specifications may be given either explicitly, or implicitly by a set of rule patterns.

The great power of make is that it will only perform an action when necessary. For example, a source file will only be compiled into an object file if the object file does not already exist or is older than the source file. If the object is newer than the source, make assumes that the object file was created from the source file and so no recompilation is necessary. This saves considerable time when dealing with large programs broken up into many pieces.

One common quirk is that an object may depend on more than one source file. For example, in C and most implementations of FORTRAN 77, a source file may `include` other source files. These source files may in turn `include` further sources, and so on. This structure of dependencies is commonly known as a dependency tree and is handled in a Makefile like so:

```
a.o: a.f
a.f: b.f c.f
b.f: d.f e.f
```

The 'target' object file a.o is to be compiled from the source file a.f, which itself depends on the sources in b.f and c.f. b.f has its own dependencies as well.

Fortran 90 [Ada92] adds a further complication to the dependency scheme. Fortran 90 source files may `use` a module; however, the name of the file containing the module need have no relationship to the module's name (in fact, a file can contain several modules). Moreover, some Fortran 90 compilers (Cray's and Parasoft's, for example) require explicit references in the compile line to the files containing modules `used` by the particular source file being compiled. The modules also need to be compiled before the files that `use` them; however,

this is taken care of automatically in the Makefile because the files at the bottom of the dependency tree, that is, the files that depend on no other file, will always be compiled first.

For a large program, creating a Makefile is a straightforward but tedious process, and therefore, a perfect candidate for automation. The Perl program makemake[1] is a Fortran 90 (and also a FORTRAN 77 and C) Makefile maker. Executing makemake *program_name* in a directory containing files with the extension .f90 (and/or .f, .F, .c) will build a Makefile that describes how to compile *program_name* as indicated above.

An example of a Makefile generated by makemake is displayed in Figure 1. PROG, SRCS, OBJS and LIBS are macros local to the Makefile that define the name of the program executable, the lists of source and object files,[2] and any libraries needed for creating the final executable. CC, CFLAGS, etc. are standard Makefile macros that define the various compilers that may be used and the flags that will be passed to them. After all the macro definitions come the actual rulesets.

make, make all and make *program_name* are all equivalent and will attempt to produce the program executable. The rule $(PROG): $(OBJS) says that the executable depends on the object files, which will be created first. The following line indicates explicitly how the object files are then actually linked to generate the program once up to date object files are available. make clean allows the user to start afresh by deleting any files created explicitly or implicitly by the Makefile.

The next two rules add .f90 to the list of suffixes that make knows about and describe how to create (compile) a .o file from a .f90 file.[3] Finally, the Makefile ends with a series of object file dependencies. Note that only dependencies on `include` and module containing files are listed. If the source file has the same root name as the object file (e.g., file.f90 is compiled into file.o—the usual situation), the dependency will be implicit and so the source file may be omitted from the dependency list. This permits the construction of a lean Makefile, containing a minimal set of explicit dependencies.

Perl [Wal90] is an appropriate language in which to write a Makefile maker. Perl (originally, the Practical Extraction and Report Language) has language elements derived from the UNIX shells, the awk and sed (stream editor) utilities as well as a variety of other sources, making it an extremely flexible and powerful string processing language which interacts well with the operating system.

Figure 2 displays the makemake program. A quick glance reveals that Perl is block structured and allows for user defined functions. There is also a plethora of special characters present that make the program look a little like a syntactic soup. Table 1 shows how some special characters are used to indicate subprogram references and various types of variables. (These prefix conventions are used in a very consistent manner in Perl. As an example, @ARGV is the list of arguments supplied to the program [this is what is called an array context], while $ARGV[0] is the first element of this list [now, ARGV is being used in a scalar context].)

For those familiar with UNIX C-shell programming, much of Perl's syntax is fairly obvious. The first non-comment line in makemake associates the file handle MAKEFILE with the

[1] Available via the World Wide Web at the URLs ftp://math.unm.edu/pub/wester/utilities/makemake and http://www.swcp.com/fortran/.

[2] The \'s at the ends of lines are used to indicate continuation.

[3] Rules for .f, .F and .c are already builtin.

```
PROG =  toms999

SRCS =  checkpoint.f90 condensed.f90 control.f90 cubic.f90 dynamic_h.f90 \
        errorcodes.f90 evaluatef.f90 extended.f90 factored.f90 fscale.f90 \
        general.f90 h0v.f90 initialize.f90 inner_product.f90 integers.f90 \
        interface77.f90 linesearch.f90 long.f90 low.f90 min_codes.f90 \
        min_defaults.f90 min_states.f90 minimize.f90 minimizef.f90 \
        myallocate.f90 normal.f90 num_constants.f90 print.f90 product.f90 \
        qnewton.f90 reals.f90 reqdprec.f90 restart.f90 srank1.f90 strings.f90 \
        sum.f90 supp_codes.f90 supp_defs.f90 supp_states.f90 support.f90 \
        systemstate.f90 test_functions.f90 testdone.f90 toms.f90 \
        true_false.f90 updateh.f90

OBJS =  checkpoint.o condensed.o control.o cubic.o dynamic_h.o errorcodes.o \
        ...
        test_functions.o testdone.o toms.o true_false.o updateh.o

LIBS =

CC = cc
CFLAGS = -O
FC = f77
FFLAGS = -O
F90 = f90
F90FLAGS = -O
LDFLAGS = -s

all: $(PROG)

$(PROG): $(OBJS)
        $(F90) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)

clean:
        rm -f $(PROG) $(OBJS) *.mod

.SUFFIXES: $(SUFFIXES) .f90

.f90.o:
        $(F90) $(F90FLAGS) -c $<

checkpoint.o: general.o min_defaults.o min_states.o reqdprec.o systemstate.o \
        true_false.o
condensed.o: general.o inner_product.o min_codes.o min_states.o reals.o \
        reqdprec.o srank1.o sum.o
control.o: min_codes.o min_states.o reqdprec.o
cubic.o: general.o min_states.o num_constants.o reals.o reqdprec.o \
        true_false.o
dynamic_h.o: general.o min_codes.o min_states.o product.o reals.o reqdprec.o \
        sum.o
errorcodes.o: min_codes.o min_states.o reqdprec.o
evaluatef.o: fscale.o general.o num_constants.o reals.o reqdprec.o \
        supp_codes.o supp_defs.o supp_states.o true_false.o
factored.o: general.o h0v.o inner_product.o min_codes.o min_states.o reals.o \
        reqdprec.o
fscale.o: reals.o reqdprec.o supp_codes.o true_false.o
general.o: reals.o reqdprec.o strings.o
h0v.o: general.o min_codes.o min_states.o reals.o reqdprec.o true_false.o
initialize.o: general.o min_codes.o min_states.o myallocate.o reals.o \
        reqdprec.o
inner_product.o: num_constants.o reals.o reqdprec.o
interface77.o: reqdprec.o supp_codes.o
linesearch.o: cubic.o general.o min_codes.o min_states.o reals.o reqdprec.o \
        true_false.o
...
```

Figure 1 An abbreviated Makefile created by makemake [Buc94a, Buc94b].

`$scalar_variable`	(can be a number or a string)
`@list_or_array`	(indexed by integers)
`%associative_array`	(indexed by strings)
`sub function_or_subroutine`	(function/subroutine definition)
`$list_or_array[`*n*`]`	(list/array reference)
`$associative_array{`*string*`}`	(associative array reference)
`&function_or_subroutine`	(function/subroutine reference)

Table 1 Basic data type and subprogram Perl syntax.

output file Makefile. Some text is then directed into MAKEFILE with all variable references within "..." expanded fully before being written out. The list @srcs is defined next. It is set to the alphabetical list of all files in the current directory that end in the extensions .f90, .f, .F and .c, in that order. The *'s are used to perform wildcard filename matches. This list is then passed to the subroutine PrintWords.

More text is written into MAKEFILE. \n and \t are converted into the characters newline and tab, respectively. The list @srcs is copied into the new list @objs. The `foreach` statement is then used to modify each element in @objs by applying the sed style substitution rule within {...}. This requires a little explanation.

The `foreach` loop could have been written

```
foreach $obj (@objs) { $obj =~ s/\.[^.]+$/.o/ };
```

where $obj represents each element of @objs as the elements of the list are stepped through one by one. The `=~` operator replaces the string on its left side with the one produced using the transformation rule on the right. In this case, the transformation rule substitutes .o for the regular expression matched by `\.[^.]+$` (in words, a period followed by one or more non-period characters at the end of the string, i.e., the language extension). The temporary variable $obj is actually unnecessary and so is omitted in the corresponding loop in the program. Perl permits many shortcuts like this, although one has to be careful not to overstep the line into obscurity by trying to be too clever!

Regular expressions are an essential part of Perl. Table 2 presents examples of the basic types which are found in makemake. It is important to realize that matches are performed left to right and that a regular expression will match the longest sequence of characters it can. For example, `^.*$` and `.*` will both match an entire line, although the first pattern is perhaps somewhat clearer in its intention. A more complicated example is the second argument to the first call to the MakeDepends subroutine:

```
^\s*include\s+["\']([^"\']+)["\']
```

This pattern is searching for extended FORTRAN 77 `include` statements. The regular expression can be expressed in words as: starting at the beginning of the line, match zero or more whitespace characters, the literal string `include`, one or more whitespace characters, a " or ', one or more characters that are not a " or ' which will be stored in register 1 (this

`doremi`	matches the string doremi
`[doremi]`	matches one of d, e, i, m, o or r
`[^doremi]`	matches any character except d, e, i, m, o or r
`do|re|mi`	matches one of do, re or mi
`.`	matches any single character except a newline
`\.`	matches a period (similarly for `\'` and other metacharacters)
`\s`	matches a whitespace character (space, tab, etc.)
`x*`	matches zero or more x's
`x+`	matches one or more x's
`^`*pattern*	anchors match to the beginning of the string
pattern`$`	anchors match to the end of the string
`(`*pattern*`)`	saves match in a consecutively numbered register
`\1`	contents of register 1

Table 2 Examples of Perl regular expressions.

will be the name of the file being `included`), and another " or '. There may be additional characters at the end of the line, but they will be ignored by this pattern.

To understand more about how Perl works, consider the function LanguageCompiler. Its objective is to determine what is the toplevel language being used[4] so that the appropriate compiler will be invoked when the objects are linked into an executable (i.e., which of the Makefile macros CC, FC or F90 should be used). The arguments to a Perl subprogram are always passed in the list @_. LanguageCompiler picks off the first element of this list using the builtin `shift` function, after which the string is converted to all lower case and assigned to the local scalar variable $compiler. As a side effect, `shift` removes this element from @_. The shortened list is then copied into the local array @srcs in the next line.

The body of the function dispatches onto one of two 'case statements' depending on whether $compiler has a value or not. $compiler will have a value if a second optional argument is supplied to makemake specifying either the type of Fortran 90 compiler that will be used, or fc or f77 (for FORTRAN 77) or cc or c (for C). If $compiler does not have a value, Fortran 90 will be assumed if there is at least one source file with a .f90 extension, otherwise FORTRAN 77 if some file ends in .f or .F, else C if a .c file extension can be found.

The 'case statements' are constructed using two common Perl idioms. The boolean construction `x && y` (x and y) is evaluated purely for side effect. The second operand is evaluated only if the first operand is insufficient to determine the truth value of the overall statement. This means that y will only be evaluated if x is true (meaning that this construct acts basically like an `if` statement).[5] The other idiom used here is utilizing the builtin `grep` function to test a string for full or partial membership in a list of strings. `grep` actually counts the number of matches,[6] however, Perl will interpret zero as false and nonzero as

[4] This is the language in which the 'main' routine is written.

[5] Equivalently, for `x || y` (x or y), y will only be evaluated if x is false.

[6] Since it is being used in a scalar context—in an array context, the matching strings would be collected together in a list instead.

true in a boolean context. The function ends by evaluating $compiler whose value may have been changed in the function's body. Since this is the last statement, this will be the value returned by LanguageCompiler.

The heart of makemake is the two dependency maker subroutines. MakeDepends, which is used to generate dependency lists for FORTRAN 77 and C files, was really quite simple to write in Perl. The routine works by examining each source file with the appropriate extension (the `foreach` loop) line by line (the `while` loop) for matches to the `include` statement pattern that was supplied as its second argument (/*pattern*/i performs a case insensitive search). If an `include` statement is found, the name of the file being `included` will be stashed in register 1 as was described earlier. The contents of register 1 will then be appended to the end of the list @incs (initially empty) by `push(@incs, $1)`.[7] Once a file has been processed, a dependency statement will be generated if any `includes` were found.

MakeDependsf90 works in a similar manner except that now there must be some way to associate the name of the module with the file that contains it. This is where an associative array (an array with string indices) becomes an extremely handy device. A first pass is made through the Fortran 90 files looking for module declarations. Whenever one is found, an entry in %filename is made with the module's name[8] as the index and the corresponding name of the object file as the value. Later on, any references to that module can be replaced with the array reference `$filename{$module}`.

Once the standard dependency list has been output, if the Fortran 90 compiler has been specified (by the optional second argument to makemake) and is either cray or parasoft, then an explicit compile directive will be generated appropriate to the compiler. Note that in these sections of code, `push` is provided with one list and *two* scalar arguments. Also, just before these code sections, PrintWords is provided with two scalar and *two* list arguments (only one list argument is specified in PrintWords' definition). Since all the arguments passed to a subprogram are concatenated together into one long list, these actions are perfectly fine as long as the last argument to the subprogram is interpreted to be a list. This feature allows Perl functions and subroutines to be used in a fairly general way in many instances.[9]

Perl is a powerful language in which it is easy to develop powerful utilities. Another important and useful concept is Makefiles. Fortran 90 Makefiles add a special twist as module references require some analysis of the source files. Perl is an excellent choice for doing this processing and moreover, completing the task and completely generating the Makefile. This discussion gives a flavor of how one goes about doing this, but see the reference manual for more details and many additional examples.

I would like to thank John Prentice for the idea of doing this project, for many suggestions and testing makemake under battlefield conditions.

[7] The syntax $1 is used rather than \1 since the register is referred to in a statement completely separate from where the match occurred.

[8] The module names are converted into lower case throughout in order to avoid confusion. Fortran 90 is case insensitive and so there should be no distinction between modulename, ModuleName and MODULENAME, for example.

[9] Basically, there is no distinction between a list and a list of lists. This has both advantages and disadvantages. Perl is not the easiest language in which to code matrix operations, for instance. Such operations are rarely needed for string manipulation, however.

References

[Ada92] Jeanne C. Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith and Jerrold L. Wagener, *Fortran 90 Handbook*, McGraw-Hill Book Company, 1992.

[Buc94a] A. G. Buckley, "Conversion to Fortran 90: A Case Study", *Transactions on Mathematical Software*, Volume 20, Number 3, September 1994, 308–353.

[Buc94b] A. G. Buckley, "Algorithm 734: A Fortran 90 Code for Unconstrained Nonlinear Minimization", *Transactions on Mathematical Software*, Volume 20, Number 3, September 1994, 354–372.

[Tal89] Steve Talbott, *Managing Projects with Make*, O'Reilly & Associates, Inc., 1989.

[Wal90] Larry Wall and Randal L. Schwartz, *Programming perl*, O'Reilly & Associates, Inc., 1990.

```
#! /usr/local/bin/perl
#
# Usage: makemake {<program name> {<F90 compiler or fc or f77 or cc or c>}}
#
# Generate a Makefile from the sources in the current directory.  The source
# files may be in either C, FORTRAN 77, Fortran 90 or some combination of
# these languages.  If the F90 compiler specified is cray or parasoft, then
# the Makefile generated will conform to the conventions of these compilers.
# To run makemake, it will be necessary to modify the first line of this script
# to point to the actual location of Perl on your system.
#
# Written by Michael Wester <wester@math.unm.edu> February 16, 1995
# Cotopaxi (Consulting), Albuquerque, New Mexico
#
open(MAKEFILE, "> Makefile");
#
print MAKEFILE "PROG =\t$ARGV[0]\n\n";
#
# Source listing
#
print MAKEFILE "SRCS =\t";
@srcs = <*.f90 *.f *.F *.c>;
&PrintWords(8, 0, @srcs);
print MAKEFILE "\n\n";
#
# Object listing
#
print MAKEFILE "OBJS =\t";
@objs = @srcs;
foreach (@objs) { s/\.[^.]+$/.o/ };
&PrintWords(8, 0, @objs);
print MAKEFILE "\n\n";
#
# Define common macros
#
print MAKEFILE "LIBS =\t\n\n";
print MAKEFILE "CC = cc\n";
print MAKEFILE "CFLAGS = -O\n";
print MAKEFILE "FC = f77\n";
print MAKEFILE "FFLAGS = -O\n";
print MAKEFILE "F90 = f90\n";
print MAKEFILE "F90FLAGS = -O\n";
print MAKEFILE "LDFLAGS = -s\n\n";
#
# make
#
print MAKEFILE "all: \$(PROG)\n\n";
print MAKEFILE "\$(PROG): \$(OBJS)\n";
print MAKEFILE "\t\$(", &LanguageCompiler($ARGV[1], @srcs);
print MAKEFILE ") \$(LDFLAGS) -o \$@ \$(OBJS) \$(LIBS)\n\n";
#
# make clean
#
print MAKEFILE "clean:\n";
print MAKEFILE "\trm -f \$(PROG) \$(OBJS) *.mod\n\n";
#
# Make .f90 a valid suffix
#
print MAKEFILE ".SUFFIXES: \$(SUFFIXES) .f90\n\n";
#
# .f90 -> .o
#
print MAKEFILE ".f90.o:\n";
print MAKEFILE "\t\$(F90) \$(F90FLAGS) -c \$<\n\n";
#
# Dependency listings
#
&MakeDependsf90($ARGV[1]);
&MakeDepends("*.f *.F", '^\s*include\s+["\']([^"\']+)["\']');
&MakeDepends("*.c",     '^\s*#\s*include\s+["\']([^"\']+)["\']');

#
# &PrintWords(current output column, extra tab?, word list); --- print words
#    nicely
#
sub PrintWords {
   local($columns) = 78 - shift(@_);
   local($extratab) = shift(@_);
   local($wordlength);
   #
   print MAKEFILE @_[0];
   $columns -= length(shift(@_));
   foreach $word (@_) {
      $wordlength = length($word);
      if ($wordlength + 1 < $columns) {
         print MAKEFILE " $word";
         $columns -= $wordlength + 1;
         }
      else {
         #
         # Continue onto a new line
         #
         if ($extratab) {
            print MAKEFILE " \\\n\t\t$word";
            $columns = 62 - $wordlength;
            }
         else {
            print MAKEFILE " \\\n\t$word";
            $columns = 70 - $wordlength;
            }
         }
      }
   }

#
# &LanguageCompiler(compiler, sources); --- determine the correct language
#    compiler
#
sub LanguageCompiler {
   local($compiler) = &toLower(shift(@_));
   local(@srcs) = @_;
   #
   if (length($compiler) > 0) {
      CASE: {
         grep(/^$compiler$/, ("fc", "f77")) &&
            do { $compiler = "FC"; last CASE; };
         grep(/^$compiler$/, ("cc", "c"))   &&
            do { $compiler = "CC"; last CASE; };
         $compiler = "F90";
         }
      }
   else {
      CASE: {
         grep(/\.f90$/, @srcs)   && do { $compiler = "F90"; last CASE; };
         grep(/\.(f|F)$/, @srcs) && do { $compiler = "FC";  last CASE; };
         grep(/\.c$/, @srcs)     && do { $compiler = "CC";  last CASE; };
         $compiler = "???";
         }
      }
   $compiler;
   }

#
# &toLower(string); --- convert string into lower case
#
sub toLower {
   local($string) = @_[0];
   $string =~ tr/A-Z/a-z/;
   $string;
   }

#
# &uniq(sorted word list); --- remove adjacent duplicate words
#
sub uniq {
   local(@words);
   foreach $word (@_) {
      if ($word ne $words[$#words]) {
         push(@words, $word);
         }
      }
   @words;
   }

#
# &MakeDepends(language pattern, include file sed pattern); --- dependency
```

```
#    maker
#
sub MakeDepends {
   local(@incs);
   local($lang) = @_[0];
   local($pattern) = @_[1];
   #
   foreach $file (<${lang}>) {
      open(FILE, $file) || warn "Cannot open $file: $!\n";
      while (<FILE>) {
         /$pattern/i && push(@incs, $1);
         }
      if (defined @incs) {
         $file =~ s/\.[^.]+$/.o/;
         print MAKEFILE "$file: ";
         &PrintWords(length($file) + 2, 0, @incs);
         print MAKEFILE "\n";
         undef @incs;
         }
      }
   }

#
# &MakeDependsf90(f90 compiler); --- FORTRAN 90 dependency maker
#
sub MakeDependsf90 {
   local($compiler) = &toLower(@_[0]);
   local(@dependencies);
   local(%filename);
   local(@incs);
   local(@modules);
   local($objfile);
   #
   # Associate each module with the name of the file that contains it
   #
   foreach $file (<*.f90>) {
      open(FILE, $file) || warn "Cannot open $file: $!\n";
      while (<FILE>) {
         /^\s*module\s+([^\s!]+)/i &&
            ($filename{&toLower($1)} = $file) =~ s/\.f90$/.o/;
         }
      }
   #
   # Print the dependencies of each file that has one or more include's or
   # references one or more modules
   #
   foreach $file (<*.f90>) {
      open(FILE, $file);
      while (<FILE>) {
         /^\s*include\s+["\']([^"\']+)["\']/i && push(@incs, $1);
         /^\s*use\s+([^\s,!]+)/i && push(@modules, &toLower($1));
         }
      if (defined @incs || defined @modules) {
         ($objfile = $file) =~ s/\.f90$/.o/;
         print MAKEFILE "$objfile: ";
         undef @dependencies;
         foreach $module (@modules) {
            push(@dependencies, $filename{$module});
            }
         @dependencies = &uniq(sort(@dependencies));
         &PrintWords(length($objfile) + 2, 0,
                     @dependencies, &uniq(sort(@incs)));
         print MAKEFILE "\n";
         undef @incs;
         undef @modules;
         #
         # Cray F90 compiler
         #
         if ($compiler eq "cray") {
            print MAKEFILE "\t\$(F90) \$(F90FLAGS) -c ";
            foreach $depend (@dependencies) {
               push(@modules, "-p", $depend);
               }
            push(@modules, $file);
            &PrintWords(30, 1, @modules);
            print MAKEFILE "\n";
            undef @modules;
            }
         #
         # ParaSoft F90 compiler
         #
         if ($compiler eq "parasoft") {
            print MAKEFILE "\t\$(F90) \$(F90FLAGS) -c ";
            foreach $depend (@dependencies) {
               $depend =~ s/\.o$/.f90/;
               push(@modules, "-module", $depend);
               }
            push(@modules, $file);
            &PrintWords(30, 1, @modules);
            print MAKEFILE "\n";
            undef @modules;
            }
         }
      }
   }
```