GNU Make

A Program for Directing Recompilation

by Richard M. Stallman and Roland McGrath

Edition 0.28 Beta,

last updated 23 September 1991,

for `make`, Version 3.61 Beta.

This is Edition 0.28 Beta of the *GNU Make Manual*,
last updated 23 September 1991,
for `make` Version 3.61 Beta.

Published by the Free Software Foundation
675 Massachusetts Avenue,
Cambridge, MA 02139 USA
Printed copies are available for $15 each.

1 Overview of make

The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. This manual describes the GNU implementation of make, which was implemented by Richard Stallman and Roland McGrath.

Our examples show C programs, since they are most common, but you can use make with any programming language whose compiler can be run with a shell command. In fact, make is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change.

To prepare to use make, you must write a file called the *makefile* that describes the relationships among files in your program, and the states the commands for updating each file. In a program, typically the executable file is updated from object files, which are in turn made by compiling source files.

Once a suitable makefile exists, each time you change some source files, this simple shell command:

```
make
```

suffices to perform all necessary recompilations. The make program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

Command arguments to make can be used to control which files should be recompiled, or how. See Chapter 10 [Running], page 81.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

2 Problems and Bugs

If you have problems with GNU `make` or think you've found a bug, please report it to Roland McGrath; he doesn't promise to do anything but he might well want to fix it.

Before reporting a bug, make sure you've actually found a real bug. Carefully reread the documentation and see if it really says you can do what you're trying to do. If it's not clear whether you should be able to do something or not, report that too; it's a bug in the documentation!

Before reporting a bug or trying to fix it yourself, try to isolate it to the smallest possible makefile that reproduces the problem. Then send us the makefile and the exact results `make` gave you. Also say what you expected to occur; this will help us decide whether the problem was really in the documentation.

Once you've got a precise problem, send email to (Internet) '`bug-gnu-utils@prep.ai.mit.edu`' or (UUCP) '`mit-eddie!prep.ai.mit.edu!bug-gnu-utils`'. Please include the version number of `make` you are using. You can get this information with the command '`make -v -f /dev/null`'.

Non-bug suggestions are always welcome as well. If you have questions about things that are unclear in the documentation or are just obscure features, ask Roland McGrath; he'll be happy to help you out (but no promises). You can send him electronic mail at Internet address '`roland@prep.ai.mit.edu`' or UUCP path '`mit-eddie!prep.ai.mit.edu!roland`'.

3 Simple Example of make

Suppose we have a text editor consisting of eight C source files and three header files. We need a makefile to tell `make` how to compile and link the editor. Assume that all the C files include '`defs.h`', but only those defining editing commands include '`commands.h`' and only low level files that change the editor buffer include '`buffer.h`'.

To recompile the editor, each changed C source file must be recompiled. If a header file has changed, to be safe each C source file that includes the header file must be recompiled. Each compilation produces an object file corresponding to the source file. Finally, if any source file has been recompiled, all the object files, whether newly made or saved from previous compilations, must be linked together to produce the new executable editor.

Here is a straightforward makefile that describes these criteria and says how to compile and link when the time comes:

```
edit : main.o kbd.o commands.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o commands.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
commands.o : command.c defs.h command.h
        cc -c commands.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
```

We split each long line into two lines using a backslash-newline; this is like using one long line, but is easier to read.

Each file that is generated by a program—that is to say, each file except for source files—is the *target* of a *rule* (see Chapter 5 [Rules], page 25). (In this example, these are the object files such as '`main.o`', '`kbd.o`', etc., and the executable file '`edit`'.) The target appears at the beginning of a line, followed by a colon.

After the colon come the target's *dependencies*: all the files that are used as input when the target file is updated. A target file needs to be recompiled or relinked if any of its dependencies changes. In addition, any dependencies that are themselves automatically generated should be updated first. In this example, '`edit`' depends on each of the eight object files; the object file '`main.o`' depends on the source file '`main.c`' and on the header file '`defs.h`'.

By default, `make` starts with the first rule (not counting rules whose target names start with '`.`'). This is called the *default goal*. Therefore, we put the rule for the executable program '`edit`' first. The other rules are processed because their targets appear as dependencies of the goal.

After each line containing a target and dependencies come one or more lines of shell commands that say how to update the target file. These lines start with a tab to tell `make` that they are command lines. But `make` does not know anything about how the commands work. It is up to you to supply commands that will update the target file properly. All `make` does is execute the commands you have specified when the target file needs to be updated.

3.1 How `make` Processes This Makefile

After reading the makefile, `make` begins its real work by processing the first rule, the one for relinking '`edit`'; but before it can fully process this rule, it must process the rules for the files '`edit`' depends on: all the object files. Each of these files is processed according to its own rule. These rules say to update the '`.o`' file by compiling its source file. The recompilation must be done if the source file, or any of the header files named as dependencies, is more recent than the object file, or if the object file does not exist.

Before recompiling an object file, `make` considers updating its dependencies, the source file and header files. This makefile does not specify anything to be done for them—the '`.c`' and '`.h`' files are not the targets of any rules—so nothing needs to be done. But automatically generated C programs, such as made by Bison or Yacc, would be updated by their own rules at this time.

After recompiling whichever object files need it, `make` can now decide whether to relink '`edit`'. This must be done if the file '`edit`' does not exist, or if any of the object files are newer than it. If an object file was just recompiled, it is now newer than '`edit`', so '`edit`' will be relinked.

Thus, if we change the file '`insert.c`' and run `make`, `make` will compile that file to update 'insert.o', and then link 'edit'. If we change the file 'command.h' and run make, make will recompile the object files 'kbd.o', 'commands.o' and 'files.o' and then link file 'edit'.

3.2 Variables Make Makefiles Simpler

In our example, we had to list all the object files twice in the rule for 'edit' (repeated here):

```
edit : main.o kbd.o commands.o display.o \
              insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o commands.o display.o \
                   insert.o search.o files.o utils.o
```

Such duplication is error-prone; if a new object file is added to the system, we might add it to one list and forget the other. We can eliminate the risk and simplify the makefile by using a *variable*. Variables allow a text string to be defined once and substituted in multiple places later (see Chapter 7 [Variables], page 55).

It is standard practice for every makefile to have a variable named objects, OBJECTS, objs, OBJS, obj or OBJ which is a list of all object file names. We would define such a variable objects with a line like this in the makefile:

```
objects = main.o kbd.o commands.o display.o \
              insert.o search.o files.o utils.o
```

Then, each place we want to put a list of the object file names, we can substitute the variable's value by writing '$(objects)' (see Chapter 7 [Variables], page 55). Here is how the rule for edit looks as a result:

```
edit : $(objects)
        cc -o edit $(objects)
```

3.3 Letting make Deduce the Commands

It is not necessary to spell out the commands for compiling the individual C source files, because make can figure them out: it has an *implicit rule* for updating a '.o' file from a correspondingly named '.c' file using a 'cc -c' command. For example, it will use the command 'cc -c main.c -o `main.o`' to compile '`main.c`' into '`main.o`'. We can therefore omit the commands from the rules for the object files. See Chapter 11 [Implicit], page 89.

When a '`.c`' file is used automatically in this way, it is also automatically added to the list of dependencies. We can therefore omit the '`.c`' files from the dependencies, provided we omit the commands.

Here is the entire example, with both of these changes, and a variable `objects` as suggested above:

```
objects =  main.o kbd.o commands.o display.o \
 insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
commands.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h
```

This is how we would write the makefile in actual practice.

3.4 Another Style of Makefile

Since the rules for the object files specify only dependencies, no commands, one can alternatively combine them by dependency instead of by target. Here is what it looks like:

```
objects =  main.o kbd.o commands.o display.o \
 insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

$(objects) : defs.h
kbd.o commands.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

Here 'defs.h' is given as a dependency of all the object files; 'commands.h' and 'buffer.h' are dependencies of the specific object files listed for them.

Whether this is better is a matter of taste: it is more compact, but some people dislike it because they find it clearer to put all the information about each target in one place.

3.5 Rules for Cleaning the Directory

Compiling a program isn't the only thing you might want to write rules for. Makefiles commonly tell how to do a few other things besides compiling the program: for example, how to delete all the object files and executables so that the directory is "clean". Here is how we would write a make rule for cleaning our example editor:

```
clean:
        rm edit $(objects)
```

This rule would be added at the end of the makefile, because we don't want it to run by default! We want the rule for edit, which recompiles the editor, to remain the default goal.

Since clean is not a dependency of edit, this rule won't run at all if we give the command 'make' with no arguments. In order to make the rule run, we have to type 'make clean'.

4 Writing Makefiles

The information that tells `make` how to recompile a system comes from reading a data base called the *makefile*.

4.1 What Makefiles Contain

Makefiles contain four kinds of things: *rules*, *variable definitions*, *directives* and *comments*. Rules, variables and directives are described at length in later chapters.

- A rule says when and how to remake one or more files, called the rule's *targets*. It lists the other files that the targets *depend on*, and may also give commands to use to create or update the targets. See Chapter 5 [Rules], page 25.
- A variable definition is a line that specifies a text string value for a *variable* that can be substituted into the text later. The simple makefile example (see Section 3.5 [Simple], page 17) shows a variable definition for `objects` as a list of all object files. See Chapter 7 [Variables], page 55, for full details.
- A directive is a command for `make` to do something special while reading the makefile. These include:
  - Reading another makefile (see Section 4.3 [Include], page 20).
  - Deciding (based on the values of variables) whether to use or ignore a part of the makefile (see Chapter 8 [Conditionals], page 67).
  - Defining a variable from a verbatim string containing multiple lines (see Section 7.7 [Defining], page 63).
- '`#`' in a line of a makefile starts a comment. It and the rest of the line are ignored, except that a trailing backslash not escaped by another backslash will continue the comment across multiple lines. Comments may appear on any of the lines in the makefile, except within a `define` directive, and perhaps within commands (where the shell decides what is a comment). A line containing just a comment (with perhaps spaces before it) is effectively blank, and is ignored.

4.2 What Name to Give Your Makefile

By default, when `make` looks for the makefile, it tries the names '`GNUmakefile`', '`makefile`' and '`Makefile`', in that order.

Normally you should call your makefile either '`makefile`' or '`Makefile`'. (We recommend '`Makefile`' because it appears prominently near the beginning of a directory listing, right near other important files such as '`README`'.) The first name checked, '`GNUmakefile`', is not recommended for most makefiles. You should use this name if you have a makefile that is specific to GNU `make`, and will not be understood by other versions of `make`.

If `make` finds none of these names, it does not use any makefile. Then you must specify a goal with a command argument, and `make` will attempt to figure out how to remake it using only its built-in implicit rules. See Chapter 11 [Implicit], page 89.

If you want to use a nonstandard name for your makefile, you can specify the makefile name with the '`-f`' option. The arguments '`-f` *name*' tell `make` to read the file *name* as the makefile. If you use more than one '`-f`' option, you can specify several makefiles. All the makefiles are effectively concatenated in the order specified. The default makefile names '`GNUmakefile`', '`makefile`' and '`Makefile`' are not checked automatically if you specify '`-f`'.

4.3 Including Other Makefiles

The `include` directive tells `make` to suspend reading the current makefile and read another makefile before continuing. The directive is a line in the makefile that looks like this:

```
include filename
```

Extra spaces are allowed and ignored at the beginning of the line, but a tab is not allowed. (If the line begins with a tab, it will be considered a command line.) Whitespace is required between `include` and *filename*; extra whitespace is ignored there and at the end of the directive. A comment starting with '`#`' is allowed at the end of the line. If *filename* contains any variable or function references, they are expanded. (See Chapter 7 [Variables], page 55.)

When `make` processes an `include` directive, it suspends reading of the containing makefile and reads from *filename* instead. When that is finished, `make` resumes reading the makefile in which the directive appears.

One occasion for using `include` directives is when several programs, handled by individual makefiles in various directories, need to use a common set of variable definitions (see Section 7.5 [Setting], page 62) or pattern rules (see Section 11.5 [Pattern Rules], page 96).

Another such occasion is when you want to automatically generate dependencies from source files; the dependencies can be put in a file that is included by the main makefile. This practice is generally cleaner than that of somehow appending the dependencies to the end of the main makefile as has been traditionally done with other versions of `make`.

If the specified name does not start with a slash, and the file is not found in the current directory, several other directories are searched. First, any directories you have specified with the '`-I`' option are searched (see Section 10.7 [Options], page 86). Then the following directories (if they exist) are searched, in this order: '`/usr/gnu/include`', '`/usr/local/include`', '`/usr/include`'. If an included makefile cannot be found in any of these directories, a warning message is generated, but it is not a fatal error; processing of the makefile containing the `include` continues.

4.4 The Variable `MAKEFILES`

If the environment variable `MAKEFILES` is defined, `make` considers its value as a list of names (separated by whitespace) of additional makefiles to be read before the others. This works much like the `include` directive: various directories are searched for those files (see Section 4.3 [Include], page 20). In addition, the default goal is never taken from one of these makefiles and it is not an error if the files listed in `MAKEFILES` are not found.

The main use of `MAKEFILES` is in communication between recursive invocations of `make` (see Section 6.6 [Recursion], page 47). It usually isn't desirable to set the environment variable before a top-level invocation of `make`, because it is usually better not to mess with a makefile from outside. However, if you are running `make` without a specific makefile, a makefile in `MAKEFILES` can do useful things to help the built-in implicit rules work better, such as defining search paths.

Some users are tempted to set `MAKEFILES` in the environment automatically on login, and program makefiles to expect this to be done. This is a very bad idea, because such makefiles will fail to work if run by anyone else. It is much better to write explicit `include` directives in the makefiles.

4.5 How Makefiles Are Remade

Sometimes makefiles can be remade from other files, such as RCS or SCCS files. If a makefile can be remade from other files, you probably want `make` to get an up-to-date version of the makefile to read in.

To this end, after reading in all makefiles, `make` will consider each as a goal target and attempt to update it. If a makefile has a rule which says how to update it (found either in that very makefile or in another one) or if an implicit rule applies to it (see Chapter 11 [Implicit], page 89), it will be updated if necessary. After all makefiles have been checked, if any have actually been changed, `make` starts with a clean slate and reads all the makefiles over again. (It will also attempt to update each of them over again, but normally this will not change them again, since they are already up to date.)

If the makefiles specify commands to remake a file but no dependencies, the file will always be remade. In the case of makefiles, a makefile that has commands but no dependencies will be remade every time `make` is run, and then again after `make` starts over and reads the makefiles in again. This would cause an infinite loop; `make` would constantly remake the makefile, and never do anything else. So, to avoid this, `make` will *not* attempt to remake makefiles which are specified as targets but have no dependencies.

If you do not specify any makefiles to be read with '`-f`' options, `make` will try the default makefile names; see Section 4.2 [Makefile Names], page 19. Unlike makefiles explicitly requested with '`-f`' options, `make` is not certain that these makefiles should exist. However, if a default makefile does not exist but can be created by running `make` rules, you probably want the rules to be run so that the makefile can be used.

Therefore, if none of the default makefiles exists, `make` will try to make each of them in the same order in which they are searched for (see Section 4.2 [Makefile Names], page 19) until it succeeds in making one, or it runs out of names to try. Note that it is not an error if `make` cannot find or make any makefile; a makefile is not always necessary.

When you use the '`-t`' option (touch targets), you would not want to use an out-of-date makefile to decide which targets to touch. So the '`-t`' option has no effect on updating makefiles; they are really updated even if '`-t`' is specified. Likewise, '`-q`' and '`-n`' do not prevent updating of makefiles, because an out-of-date makefile would result in the wrong output for other targets. Thus, '`make -f mfile -n foo`' will update '`mfile`', read it in, and then print the commands to update '`foo`' and its dependencies without running them. The commands printed for '`foo`' will be those specified in the updated contents of '`mfile`'.

However, on occasion you might actually wish to prevent updating of even the makefiles. You can do this by specifying the makefiles as goals in the command line as well as specifying them as makefiles. When the makefile name is specified explicitly as a goal, the options '`-t`' and so on do apply to them.

Thus, '`make -f mfile -n mfile foo`' would read the makefile '`mfile`', print the commands needed to update it without actually running them, and then print the commands needed to update 'foo' without running them. The commands for 'foo' will be those specified by the existing contents of 'mfile'.

4.6 Overriding Part of One Makefile with Another Makefile

Sometimes it is useful to have a makefile that is mostly just like another makefile. You can often use the 'include' directive to include one in the other, and add more targets or variable definitions. However, if the two makefiles give different commands for the same target, make will not let you just do this. But there is another way.

In the containing makefile (the one that wants to include the other), you can use the .DEFAULT special target to say that to remake any target that cannot be made from the information in the containing makefile, make should look in another makefile. See Section 11.6 [Last Resort], page 103, for more information on .DEFAULT.

For example, if you have a makefile called 'Makefile' that says how to make the target 'foo' (and other targets), you can write a makefile called 'GNUmakefile' that contains:

```
foo:
        frobnicate > foo

.DEFAULT:
        @$(MAKE) -f Makefile $@
```

If you say 'make foo', make will find 'GNUmakefile', read it, and see that to make 'foo', it needs to run the command 'frobnicate > foo'. If you say 'make bar', make will find no way to make 'bar' in 'GNUmakefile', so it will use the commands from .DEFAULT: 'make -f Makefile bar'. If 'Makefile' provides a rule for updating 'bar', make will apply the rule. And likewise for any other target that 'GNUmakefile' does not say how to make.

5 Writing Rules

A *rule* appears in the makefile and says when and how to remake certain files, called the rule's *targets* (usually only one per rule). It lists the other files that are the *dependencies* of the target, and *commands* to use to create or update the target.

The order of rules is not significant, except for determining the *default goal*: the target for `make` to consider, if you do not otherwise specify one. The default goal is the target of the first rule in the first makefile, except that targets starting with a period do not count unless they contain slashes as well; also, a target that defines a pattern rule (see Section 11.5 [Pattern Rules], page 96) or a suffix rule (see Section 11.7 [Suffix Rules], page 104) has no effect on the default goal.

Therefore, we usually write the makefile so that the first rule is the one for compiling the entire program or all the programs described by the makefile. See Section 10.2 [Goals], page 81.

5.1 Rule Syntax

In general, a rule looks like this:

```
targets : dependencies
        command
        command
        ...
```

or like this:

```
targets : dependencies ; command
        command
        command
        ...
```

The *targets* are file names, separated by spaces. Wild card characters may be used (see Section 5.2 [Wildcards], page 26) and a name of the form '`a(`*m*`)`' represents member *m* in archive file *a* (see Section 12.1 [Archive Members], page 109). Usually there is only one target per rule, but occasionally there is a reason to have more; See Section 5.8 [Multiple Targets], page 36.

The *command* lines start with a tab character. The first command may appear on the line after the dependencies, with a tab character, or may appear on the same line, with a semicolon. Either way, the effect is the same. See Chapter 6 [Commands], page 43.

Because dollar signs are used to start variable references, if you really want a dollar sign in the rule you must write two of them (‘`$$`’). See Chapter 7 [Variables], page 55. You may split a long line by inserting a backslash followed by a newline, but this is not required, as `make` places no limit on the length of a line in a makefile.

A rule tells `make` two things: when the targets are out of date, and how to update them when necessary.

The criterion for being out of date is specified in terms of the *dependencies*, which consist of file names separated by spaces. (Wildcards and archive members are allowed here too.) A target is out of date if it does not exist or if it is older than any of the dependencies (by comparison of last-modification times). The idea is that the contents of the target file are computed based on information in the dependencies, so if any of the dependencies changes, the contents of the existing target file are no longer necessarily valid.

How to update is specified by *commands*. These are lines to be executed by the shell (normally ‘`sh`’), but with some extra features (see Chapter 6 [Commands], page 43).

5.2 Using Wildcards Characters in File Names

A single file name can specify many files using *wildcard characters*. The wildcard characters in `make` are ‘`*`’, ‘`?`’ and ‘`[...]`’, the same as in the Bourne shell. For example, ‘`*.c`’ specifies a list of all the files (in the working directory) whose names end in ‘`.c`’.

The character ‘`~`’ at the beginning of a file name also has special significance. If alone, or followed by a slash, it represents your home directory. For example ‘`~/bin`’ expands to ‘`/home/you/bin`’. If the ‘`~`’ is followed by a word, the string represents the home directory of the user named by that word. For example ‘`~me/bin`’ expands to ‘`/home/me/bin`’.

Wildcard expansion happens automatically in targets, in dependencies, and in commands. In other contexts, wildcard expansion happens only if you request it explicitly with the `wildcard` function.

The special significance of a wildcard character can be turned off by preceding it with a backslash. Thus, ‘`foo\*bar`’ would refer to a specific file whose name consists of ‘`foo`’, an asterisk, and ‘`bar`’.

5.2.1 Wildcard Examples

Wildcards can be used in the commands of a rule. For example, here is a rule to delete all the object files:

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

Wildcards are also useful in the dependencies of a rule. With the following rule in the makefile, '`make print`' will print all the '`.c`' files that have changed since the last time you printed them:

```
print: *.c
        lpr -p $?
        touch print
```

This rule uses '`print`' as an empty target file; see Section 5.6 [Empty Targets], page 35.

Wildcard expansion does not happen when you define a variable. Thus, if you write this:

```
objects=*.o
```

then the value of the variable `objects` is the actual string '`*.o`'. However, if you use the value of `objects` in a target, dependency or command, wildcard expansion will take place at that time.

5.2.2 Pitfalls of Using Wildcards

Now here is an example of a naive way of using wildcard expansion, that does not do what you would intend. Suppose you would like to say that the executable file '`foo`' is made from all the object files in the directory, and you write this:

```
objects=*.o

foo : $(objects)
        cc -o foo $(CFLAGS) $(objects)
```

The value of `objects` is the actual string '`*.o`'. Wildcard expansion happens in the rule for '`foo`', so that each *existing* '`.o`' file becomes a dependency of '`foo`' and will be recompiled if necessary.

But what if you delete all the '.o' files? Then '*.o' will expand into *nothing*. The target 'foo' will have no dependencies and would be remade by linking no object files. This is not what you want!

Actually it is possible to obtain the desired result with wildcard expansion, but you need more sophisticated techniques, including the `wildcard` function and string substitution. These are described in the following section.

5.2.3 The Function wildcard

Wildcard expansion happens automatically in rules. But wildcard expansion does not normally take place when a variable is set, or inside the arguments of a function. If you want to do wildcard expansion in such places, you need to use the `wildcard` function, like this:

```
$(wildcard pattern)
```

This string, used anywhere in a makefile, is replaced by a space-separated list of names of existing files that match the pattern *pattern*.

One use of the `wildcard` function is to get a list of all the C source files in a directory, like this:

```
$(wildcard *.c)
```

We can change the list of C source files into a list of object files by substituting '.o' for '.c' in the result, like this:

```
$(subst .c,.o,$(wildcard *.c))
```

(Here we have used another function, `subst`. See Section 9.2 [Text Functions], page 72.)

Thus, a makefile to compile all C source files in the directory and then link them together could be written as follows:

```
objects:=$(subst .c,.o,$(wildcard *.c))

foo : $(objects)
        cc -o foo $(LDFLAGS) $(objects)
```

(This takes advantage of the implicit rule for compiling C programs, so there is no need to write explicit rules for compiling the files. See Section 7.2 [Flavors], page 56, for an explanation of '`:=`', which is a variant of '`=`'.)

5.3 Searching Directories for Dependencies

For large systems, it is often desirable to put sources in a separate directory from the binaries. The *directory search* features of `make` facilitate this by searching several directories automatically to find a dependency. When you redistribute the files among directories, you do not need to change the individual rules, just the search paths.

5.3.1 `VPATH`: Search Path for All Dependencies

The value of the `make` variable `VPATH` specifies a list of directories which `make` should search (in the order specified) for dependency files. The directory names are separated by colons. For example:

```
VPATH = src:../headers
```

specifies a path containing two directories, '`src`' and '`../headers`'.

Whenever a file listed as a dependency does not exist in the current directory, the directories listed in `VPATH` are searched for a file with that name. If a file is found in one of them, that file becomes the dependency. Rules may then specify the names of source files as if they all existed in the current directory.

Using the value of `VPATH` set in the previous example, a rule like this:

```
foo.o : foo.c
```

is interpreted as if it were written like this:

```
foo.o : src/foo.c
```

assuming the file '`foo.c`' does not exist in the current directory but is found in the directory '`src`'.

5.3.2 The `vpath` Directive

Similar to the `VPATH` variable but more selective is the `vpath` directive, which allows you to specify a search path for a particular class of filenames, those that match a particular pattern. Thus you can supply certain search directories for one class of filenames and other directories (or none) for other filenames.

There are three forms of the `vpath` directive:

`vpath` *pattern directories*

Specify the search path *directories* for filenames that match `pattern`. If another path was previously specified for the same pattern, the new path is effectively appended to the old path.

The search path, *directories*, is a colon-separated list of directories to be searched, just like the search path used in the `VPATH` variable.

`vpath` *pattern*

Clear out the search path associated with *pattern*.

`vpath`

Clear all search paths previously specified with `vpath` directives.

A `vpath` pattern is a string containing a '`%`' character. The string must match the filename of a dependency that is being searched for, the '`%`' character matching any sequence of zero or more characters (as in pattern rules; see Section 11.5 [Pattern Rules], page 96). (If there is no '`%`', the pattern must match the dependency, which is not useful very often.)

'`%`' characters in a `vpath` directive's pattern can be quoted with preceding backslashes ('`\`'). Backslashes that would otherwise quote '`%`' characters can be quoted with more backslashes. Backslashes that quote '`%`' characters or other backslashes are removed from the pattern before it is compared to file names. Backslashes that are not in danger of quoting '`%`' characters go unmolested.

When a dependency fails to exist in the current directory, if the *pattern* in a `vpath` directive matches the name of the dependency file, then the *directories* in that directive are searched just like (and before) the directories in the `VPATH` variable. For example,

```
vpath %.h ../headers
```

tells `make` to look for any dependency whose name ends in '`.h`' in the directory '`../headers`' if the file is not found in the current directory.

If several `vpath` patterns match the dependency file's name, then `make` processes each matching `vpath` directive one by one, searching all the directories mentioned in each directive. The `vpath` directives are processed in the order in which they appear in the makefiles.

5.3.3 Writing Shell Commands with Directory Search

When a dependency is found in another directory through directory search, this cannot change the commands of the rule; they will execute as written. Therefore, you must write the commands with care so that they will look for the dependency in the directory where `make` finds it.

This is done with the *automatic variables* such as '`$^`' (see Section 11.5.3 [Automatic], page 99). For instance, the value of '`$^`' is a list of all the dependencies of the rule, including the names of the directories in which they were found, and the value of '`$@`' is the target. Thus:

```
foo.o : foo.c
        cc -c $(CFLAGS) $^ -o $@
```

The variable `CFLAGS` exists so you can specify flags for C compilation by implicit rule; we use it here for consistency so it will affect all C compilations uniformly (see Section 11.3 [Implicit Variables], page 93).

Often the dependencies include header files as well, which you don't want to mention in the commands. The function `firstword` can be used to extract just the first dependency from the entire list, as shown here (see Section 9.3 [Filename Functions], page 74):

```
VPATH = src:../headers
foo.o : foo.c defs.h hack.h
        cc -c $(CFLAGS) $(firstword $^) -o $@
```

Here the value of '`$^`' would be something like '`src/foo.c ../headers/defs.h hack.h`', from which '`$(firstword $^)`' extracts just '`src/foo.c`'.

5.3.4 Directory Search and Implicit Rules

The search through the directories specified in `VPATH` or with `vpath` happens also during consideration of implicit rules (see Chapter 11 [Implicit], page 89).

For example, when a file '`foo.o`' has no explicit rule, `make` considers implicit rules, such as to compile '`foo.c`' if that file exists. If such a file is lacking in the current directory, the appropriate directories are searched for it. If '`foo.c`' exists (or is mentioned in the makefile) in any of the directories, the implicit rule for C compilation is applicable.

The commands of all the built-in implicit rules normally use automatic variables as a matter of necessity; consequently they will use the file names found by directory search with no extra effort.

5.3.5 Directory Search for Link Libraries

Directory search applies in a special way to libraries used with the linker. This special feature comes into play when you write a dependency whose name is of the form '`-l`*name*'. (You can tell something funny is going on here because the dependency is normally the name of a file, and the *file name* of the library looks like '`lib`*name*`.a`', not like '`-l`*name*'.)

When a dependency's name has the form '`-l`*name*', `make` handles it specially by searching for the file '`lib`*name*`.a`' in the directories '`/lib`' and '`/usr/lib`', and then using matching `vpath` search paths and the `VPATH` search path.

For example,

```
foo : foo.c -lcurses
        cc $^ -o $@
```

would cause the command '`cc foo.c -lcurses -o foo`' to be executed when '`foo`' is older than '`foo.c`' or than '`libcurses.a`' (which has probably been found by directory search in the file '`/usr/lib/libcurses.a`').

As shown by the example above, the file name found by directory search is used only for comparing the file time with the target file's time. It does not replace the file's name in later usage (such as in automatic variables like `$^`); the name remains unchanged, still starting with '`-l`'. This leads to the correct results because the linker will repeat the appropriate search when it processes this argument.

5.4 Phony Targets

A phony target is one that is not really the name of a file. It is just a name for some commands to be executed when you make an explicit request.

If you write a rule whose commands will not create the target file, the commands will be executed every time the target comes up for remaking. Here is an example:

```
clean:
        rm *.o temp
```

Because the `rm` command does not create a file named '`clean`', probably no such file will ever exist. Therefore, the `rm` command will be executed every time you say '`make clean`'.

The phony target will cease to work if anything ever does create a file named '`clean`' in this directory. Since it has no dependencies, the file '`clean`' would inevitably be considered up to date, and its commands would not be executed. To avoid this problem, you can explicitly declare the target to be phony, using the special target `.PHONY` (see Section 5.7 [Special Targets], page 35) as follows:

```
.PHONY : clean
```

Once this is done, '`make clean`' will run the commands regardless of whether there is a file named '`clean`'.

A phony target should not be a dependency of a real target file; strange things can result from that. As long as you don't do that, the phony target commands will be executed only when the phony target is a specified goal (see Section 10.2 [Goals], page 81).

Phony targets can have dependencies. When one directory contains multiple programs, it is most convenient to describe all of the programs in one makefile '`./Makefile`'. Since the target remade by default will be the first one in the makefile, it is common to make this a phony target named '`all`' and give it, as dependencies, all the individual programs. For example:

```
all : prog1 prog2 prog3
.PHONY : all

prog1 : prog1.o utils.o
        cc -o prog1 prog1.o utils.o

prog2 : prog2.o
        cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o
```

```
        cc -o prog3 prog3.o sort.o utils.o
```

Now you can say just '`make`' to remake all three programs, or specify as arguments the ones to remake (as in '`make prog1 prog3`').

When one phony target is a dependency of another, it serves as a subroutine of the other. For example, here '`make cleanall`' will delete the object files, the difference files, and the file '`program`':

```
cleanall : cleanobj cleandiff
        rm program

cleanobj :
        rm *.o

cleandiff :
        rm *.diff
```

5.5 Rules without Commands or Dependencies

If a rule has no dependencies or commands, and the target of the rule is a nonexistent file, then `make` imagines this target to have been updated whenever its rule is run. This implies that all targets depending on this one will always have their commands run.

An example will illustrate this:

```
clean: FORCE
        rm $(objects)
FORCE:
```

Here the target '`FORCE`' satisfies the special conditions, so the target '`clean`' that depends on it is forced to run its commands. There is nothing special about the name '`FORCE`', but that is one name commonly used this way.

As you can see, using '`FORCE`' this way has the same results as using '`.PHONY: clean`'. The latter is more explicit, but other versions of `make` do not support it; thus '`FORCE`' appears in many makefiles.

5.6 Empty Target Files to Record Events

The *empty target* is a variant of the phony target; it is used to hold commands for an action that you request explicitly from time to time. Unlike a phony target, this target file can really exist; but the file's contents do not matter, and usually are empty.

The purpose of the empty target file is to record, with its last-modification time, when the rule's commands were last executed. It does so because one of the commands is a `touch` command to update the target file.

The empty target file must have some dependencies. When you ask to remake the empty target, the commands are executed if any dependency is more recent than the target; in other words, if a dependency has changed since the last time you remade the target. Here is an example:

```
print: foo.c bar.c
        lpr -p $?
        touch print
```

With this rule, '`make print`' will execute the `lpr` command if either source file has changed since the last '`make print`'. The automatic variable '`$?`' is used to print only those files that have changed (see Section 11.5.3 [Automatic], page 99).

5.7 Special Built-in Target Names

Certain names have special meanings if they appear as targets.

`.PHONY` The dependencies of the special target `.PHONY` are considered to be phony targets. When it is time to consider such a target, `make` will run its commands unconditionally, regardless of whether a file with that name exists or what its last-modification time is. See Section 5.4 [Phony Targets], page 32.

`.SUFFIXES`
The dependencies of the special target `.SUFFIXES` are the list of suffixes to be used in checking for suffix rules. See Section 11.7 [Suffix Rules], page 104.

`.DEFAULT` The commands specified for `.DEFAULT` are used for any target for which no other commands are known (either explicitly or through an implicit rule). If `.DEFAULT` commands are specified, every nonexistent file mentioned as a dependency will have these commands executed on its behalf. See Section 11.8 [Search Algorithm], page 105.

`.PRECIOUS`
: The targets which `.PRECIOUS` depends on are given this special treatment: if `make` is killed or interrupted during the execution of their commands, the target is not deleted. See Section 6.5 [Interrupts], page 47.

`.IGNORE`
: Simply by being mentioned as a target, `.IGNORE` says to ignore errors in execution of commands. The dependencies and commands for `.IGNORE` are not meaningful.

  '`.IGNORE`' exists for historical compatibility. Since `.IGNORE` affects every command in the makefile, it is not very useful; we recommend you use the more selective ways to ignore errors in specific commands. See Section 6.4 [Errors], page 46.

`.SILENT`
: Simply by being mentioned as a target, `.SILENT` says not to print commands before executing them. The dependencies and commands for `.SILENT` are not meaningful.

  '`.SILENT`' exists for historical compatibility. We recommend you use the more selective ways to silence specific commands. See Section 6.1 [Echoing], page 43.

Any defined implicit rule suffix also counts as a special target if it appears as a target, and so does the concatenation of two suffixes, such as '`.c.o`'. These targets are suffix rules, an obsolete way of defining implicit rules (but a way still widely used). In principle, any target name could be special in this way if you break it in two and add both pieces to the suffix list. In practice, suffixes normally begin with '`.`', so these special target names also begin with '`.`'. See Section 11.7 [Suffix Rules], page 104.

5.8 Multiple Targets in a Rule

A rule with multiple targets is equivalent to writing many rules, each with one target, and all identical aside from that. The same commands apply to all the targets, but their effects may vary because you can substitute the actual target name into the command using '`$@`'. The rule contributes the same dependencies to all the targets also.

This is useful in two cases.

- You want just dependencies, no commands. For example:

  ```
  kbd.o commands.o files.o: command.h
  ```

  gives an additional dependency to each of the three object files mentioned.
- Similar commands work for all the targets. The commands do not need to be absolutely identical, since the automatic variable '`$@`' can be used to substitute the particular target to be remade into the commands (see Section 11.5.3 [Automatic], page 99). For example:

```
bigoutput littleoutput : text.g
        generate text.g -$(subst output,,$@) > $@
```

is equivalent to

```
bigoutput : text.g
        generate text.g -big > bigoutput
littleoutput : text.g
        generate text.g -little > littleoutput
```

Here we assume the hypothetical program `generate` makes two types of output, one if given '`-big`' and one if given '`-little`'.

5.9 Static Pattern Rules

Static pattern rules are rules which specify multiple targets and construct the dependency names for each target based on the target name. They are more general than ordinary rules with multiple targets because the targets don't have to have identical dependencies. Their dependencies must be *analogous*, but not necessarily *identical*.

5.9.1 Syntax of Static Pattern Rules

Here is the syntax of a static pattern rule:

targets: *target-pattern*: *dep-patterns* ...
        commands
        ...

The *targets* gives the list of targets that the rule applies to. The targets can contain wildcard characters, just like the targets of ordinary rules (see Section 5.2 [Wildcards], page 26).

The *target-pattern* and *dep-patterns* say how to compute the dependencies of each target. Each target is matched against the *target-pattern* to extract a part of the target name, called the *stem*. This stem is substituted into each of the *dep-patterns* to make the dependency names (one from each *dep-pattern*).

Each pattern normally contains the character '`%`' just once. When the *target-pattern* matches a target, the '`%`' can match any part of the target name; this part is called the *stem*. The rest of the pattern must match exactly. For example, the target '`foo.o`' matches the pattern '`%.o`', with '`foo`' as the stem. The targets '`foo.c`' and '`foo.out`' don't match that pattern.

The dependency names for each target are made by substituting the stem for the '%' in each dependency pattern. For example, if one dependency pattern is '%.c', then substitution of the stem 'foo' gives the dependency name 'foo.c'. It is legitimate to write a dependency pattern that doesn't contain '%'; then this dependency is the same for all targets.

'%' characters in pattern rules can be quoted with preceding backslashes ('\'). Backslashes that would otherwise quote '%' characters can be quoted with more backslashes. Backslashes that quote '%' characters or other backslashes are removed from the pattern before it is compared file names or has a stem substituted into it. Backslashes that are not in danger of quoting '%' characters go unmolested. For example, the pattern 'the\%weird\\%pattern\\' has 'the%weird\' preceding the operative '%' character, and 'pattern\\' following it. The final two backslashes are left alone because they can't affect any '%' character.

Here is an example, which compiles each of 'foo.o' and 'bar.o' from the corresponding '.c' file:

```
objects = foo.o bar.o

$(objects): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
```

Each target specified must match the target pattern; a warning is issued for each target that does not. If you have a list of files, only some of which will match the pattern, you can use the filter function to remove nonmatching filenames (see Section 9.2 [Text Functions], page 72):

```
files = foo.elc bar.o lose.o

$(filter %.o,$(files)): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)): %.elc: %.el
        emacs -f batch-byte-compile $<
```

Here the result of '$(filter %.o,$(files))' is 'bar.o lose.o', and the first static pattern rule causes each of these object files to be updated by compiling the corresponding C source file. The result of '$(filter %.elc,$(files))' is 'foo.elc', so that file is made from 'foo.el'.

5.9.2 Static Pattern Rules versus Implicit Rules

A static pattern rule has much in common with an implicit rule defined as a pattern rule (see Section 11.5 [Pattern Rules], page 96). Both have a pattern for the target and patterns for constructing the names of dependencies. The difference is in how `make` decides *when* the rule applies.

An implicit rule *can* apply to any target that matches its pattern, but it *does* apply only when the target has no commands otherwise specified, and only when the dependencies can be found. If more than one implicit rule appears applicable, only one applies; the choice depends on the order of rules.

By contrast, a static pattern rule applies to the precise list of targets that you specify in the rule. It cannot apply to any other target and it invariably does apply to each of the targets specified. If two conflicting rules apply, and both have commands, that's an error.

The static pattern rule can be better than an implicit rule for these reasons:

- You may wish to override the usual implicit rule for a few files whose names cannot be categorized syntactically but can be given in an explicit list.
- If you cannot be sure of the precise contents of the directories you are using, you may not be sure which other irrelevant files might lead `make` to use the wrong implicit rule. The choice might depend on the order in which the implicit rule search is done. With static pattern rules, there is no uncertainty: each rule applies to precisely the targets specified.

5.10 Multiple Rules for One Target

One file can be the target of several rules. All the dependencies mentioned in all the rules are merged into one list of dependencies for the target. If the target is older than any dependency from any rule, the commands are executed.

There can only be one set of commands to be executed for a file. If more than one rule gives commands for the same file, the last `make` uses the last set given and prints an error message. (As a special case, if the file's name begins with a dot, no error message is printed. This odd behavior is only for compatibility with other `make`s.) There is no reason to write your makefiles this way; that is why `make` gives you an error message.

An extra rule with just dependencies can be used to give a few extra dependencies to many files at once. For example, one usually has a variable named `objects` containing a list of all the compiler output files in the system being made. An easy way to say that all of them must be recompiled if '`config.h`' changes is to write

```
objects = foo.o bar.o
foo.o : defs.h
bar.o : defs.h test.h
$(objects) : config.h
```

This could be inserted or taken out without changing the rules that really say how to make the object files, making it a convenient form to use if you wish to add the additional dependency intermittently.

Another wrinkle is that the additional dependencies could be specified with a variable that you could set with a command argument to `make` (see Section 10.5 [Overriding], page 85). For example,

```
extradeps=
$(objects) : $(extradeps)
```

means that the command '`make extradeps=foo.h`' will consider '`foo.h`' as a dependency of each object file, but plain '`make`' will not.

If none of the explicit rules for a target has commands, then `make` searches for an applicable implicit rule to find some commands. See Chapter 11 [Implicit], page 89.

5.11 Double-Colon Rules

Double-colon rules are rules written with '`::`' instead of '`:`' after the target names. They are handled differently from ordinary rules when the same target appears in more than one rule.

When a target appears in multiple rules, all the rules must be the same type: all ordinary, or all double-colon. If they are double-colon, each of them is independent of the others. Each double-colon rule's commands are executed if the target is older than any dependencies of that rule. This can result in executing none, any or all of the double-colon rules.

Double-colon rules with the same target are in fact completely separate from one another. Each double-colon rule is processed individually, just as rules with different targets are processed.

The double-colon rules for a target are executed in the order they appear in the makefile. However, the cases where double-colon rules really make sense are those where the order of executing the commands would not matter.

Double-colon rules are somewhat obscure and not often very useful; they provide a mechanism for cases in which the method used to update a target differs depending on which dependency files caused the update, and such cases are rare.

Each double-colon rule should specify commands; if it does not, an implicit rule will be used if one applies. See Chapter 11 [Implicit], page 89.

6 Writing the Commands in Rules

The commands of a rule consist of shell command lines to be executed one by one. Each command line must start with a tab, except that the first command line may be attached to the target-and-dependencies line with a semicolon in between. Blank lines and lines of just comments may appear among the command lines; they are ignored.

Users use many different shell programs, but commands in makefiles are always interpreted by '`/bin/sh`' unless the makefile specifies otherwise.

Whether comments can be written on command lines, and what syntax they use, is under the control of the shell that is in use. If it is '`/bin/sh`', a '`#`' at the start of a word starts a comment.

6.1 Command Echoing

Normally `make` prints each command line before it is executed. We call this *echoing* because it gives the appearance that you are typing the commands yourself.

When a line starts with '`@`', the echoing of that line is suppressed. The '`@`' is discarded before the command is passed to the shell. Typically you would use this for a command whose only effect is to print something, such as an `echo` command to indicate progress through the makefile:

```
@echo About to make distribution files
```

When `make` is given the flag '`-n`', echoing is all that happens, no execution. See Section 10.7 [Options], page 86. In this case and only this case, even the commands starting with '`@`' are printed. This flag is useful for finding out which commands `make` thinks are necessary without actually doing them.

The '`-s`' flag to `make` prevents all echoing, as if all commands started with '`@`'. A rule in the makefile for the special target `.SILENT` has the same effect (see Section 5.7 [Special Targets], page 35). `.SILENT` is essentially obsolete since '`@`' is more flexible.

6.2 Command Execution

When it is time to execute commands to update a target, they are executed by making a new subshell for each line. (In practice, `make` may take shortcuts that do not affect the results.)

This implies that shell commands such as `cd` that set variables local to each process will not affect the following command lines. If you want to use `cd` to affect the next command, put the two on a single line with a semicolon between them. Then `make` will consider them a single command and pass them, together, to a shell which will execute them in sequence. For example:

```
foo : bar/lose
        cd bar; gobble lose > ../foo
```

If you would like to split a single shell command into multiple lines of text, you must use a backslash at the end of all but the last subline. Such a sequence of lines is combined into a single line, by deleting the backslash-newline sequences, before passing it to the shell. Thus, the following is equivalent to the preceding example:

```
foo : bar/lose
        cd bar;  \
        gobble lose > ../foo
```

The program used as the shell is taken from the variable `SHELL`. By default, the program '`/bin/sh`' is used.

Unlike most variables, the variable `SHELL` will not be set from the environment, except in a recursive `make`. This is because the environment variable `SHELL` is used to specify your personal choice of shell program for interactive use. It would be very bad for personal choices like this to affect the functioning of makefiles. See Section 7.8 [Environment], page 64.

6.3 Parallel Execution

GNU `make` knows how to execute several commands at once. Normally, `make` will execute only one command at a time, waiting for it to finish before executing the next. However, the '`-j`' option tells `make` to execute many commands simultaneously.

If the '`-j`' option is followed by an integer, this is the number of commands to execute at once; this is called the number of *job slots*. If there is nothing looking like an integer after the '`-j`' option, there is no limit on the number of job slots. The default number of job slots is one, which means serial execution (one thing at a time).

One unpleasant consequence of running several commands simultaneously is that output from all of the commands comes when the commands send it, so messages from different commands may be interspersed.

Another problem is that two processes cannot both take input from the same device; so to make sure that only one command tries to take input from the terminal at once, `make` will invalidate the standard input streams of all but one running command. This means that attempting to read from standard input, for most child processes if there are several, will usually be a fatal error (a '`Broken pipe`' signal).

It is unpredictable which command will have a valid standard input stream (which will come from the terminal, or wherever you redirect the standard input of `make`). The first command run will always get it first, and the first command started after that one finishes will get it next, and so on.

We will change how this aspect of `make` works if we find a better alternative. In the mean time, you should not rely on any command using standard input at all if you are using the parallel execution feature; but if you are not using this feature, then standard input works normally in all commands.

If a command fails (is killed by a signal or exits with a nonzero status), and errors are not ignored for that command (see Section 6.4 [Errors], page 46), the remaining command lines to remake the same target will not be run. If a command fails and the '`-k`' option was not given (see Section 10.7 [Options], page 86), `make` aborts execution. If make terminates for any reason (including a signal) with child processes running, it waits for them to finish before actually exiting.

When the system is heavily loaded, you will probably want to run fewer jobs than when it is lightly loaded. You can use the '`-l`' option to tell `make` to limit the number of jobs to run at once, based on the load average. The '`-l`' option is followed by a floating-point number. For example,

```
-l 2.5
```

will not let `make` start more than one job if the load average is above 2.5. The '`-l`' option with no following number removes the load limit, if one was given with a previous '`-l`' option.

More precisely, when `make` goes to start up a job, and it already has at least one job running, it checks the current load average; if it is not lower than the limit given with '-l', make waits until the load average goes below that limit, or until all the other jobs finish.

By default, there is no load limit.

6.4 Errors in Commands

After each shell command returns, make looks at its exit status. If the command completed successfully, the next command line is executed in a new shell, or after the last command line is executed, the rule is finished.

If there is an error (the exit status is nonzero), make gives up on the current rule, and perhaps on all rules.

Sometimes the failure of a certain command does not indicate a problem. For example, you may use the mkdir command to insure that a directory exists. If the directory already exists, mkdir will report an error, but you probably want make to continue regardless.

To ignore errors in a command line, write a '-' at the beginning of the line's text (after the initial tab). The '-' is discarded before the command is passed to the shell for execution. For example,

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

When make is run with the '-i' flag, errors are ignored in all commands of all rules. A rule in the makefile for the special target .IGNORE has the same effect. These ways of ignoring errors are obsolete because '-' is more flexible.

When errors are to be ignored, because of either a '-' or the '-i' flag, make treats an error return just like success, except that it prints out a message telling you the status code the command exited with and saying that the error has been ignored.

When an error happens that make has not been told to ignore, it implies that the current target cannot be correctly remade, and neither can any other that depends on it either directly or indirectly. No further commands will be executed for these targets, since their preconditions have not been achieved.

Normally `make` gives up immediately in this circumstance, returning a nonzero status. However, if the '`-k`' flag is specified, `make` continues to consider the other dependencies of the pending targets, remaking them if necessary, before it gives up and returns nonzero status. For example, after an error in compiling one object file, '`make -k`' will continue compiling other object files even though it already knows that linking them will be impossible. See Section 10.7 [Options], page 86.

The usual behavior assumes that your purpose is to get the specified targets up to date; once `make` learns that this is impossible, it might as well report the failure immediately. The '`-k`' option says that the real purpose is to test as much as possible of the changes made in the program, perhaps to find several independent problems so that you can correct them all before the next attempt to compile. This is why Emacs's `compile` command passes the '`-k`' flag by default.

6.5 Interrupting or Killing `make`

If `make` gets a fatal signal while a command is executing, it may delete the target file that the command was supposed to update. This is done if the target file's last-modification time has changed since `make` first checked it.

The purpose of deleting the target is to make sure that it is remade from scratch when `make` is next run. Why is this? Suppose you type `Ctrl-c` while a compiler is running, and it has begun to write an object file '`foo.o`'. The `Ctrl-c` kills the compiler, resulting in an incomplete file whose last-modification time is newer than the source file '`foo.c`'. But `make` also receives the `Ctrl-c` signal and deletes this incomplete file. If `make` did not do this, the next invocation of `make` would think that '`foo.o`' did not require updating—resulting in a strange error message from the linker when it tries to link an object file half of which is missing.

You can prevent the deletion of a target file in this way by making the special target `.PRECIOUS` depend on it. Before remaking a target, `make` checks to see whether it appears on the dependencies of `.PRECIOUS`, and thereby decides whether the target should be deleted if a signal happens. Some reasons why you might do this are that the target is updated in some atomic fashion, or exists only to record a modification-time (its contents do not matter), or must exist at all times to prevent other sorts of trouble.

6.6 Recursive Use of `make`

Recursive use of `make` means using `make` as a command in a makefile. This technique is useful when you want separate makefiles for various subsystems that compose a larger system. For example, suppose you have a subdirectory 'subdir' which has its own makefile, and you would like the containing directory's makefile to run make on the subdirectory. You can do it by writing this:

```
subsystem:
        cd subdir; $(MAKE)
```

or, equivalently, this (see Section 10.7 [Options], page 86):

```
subsystem:
        $(MAKE) -C subdir
```

You can write recursive make commands just by copying this example, but there are many things to know about how they work and why, and about how the sub-make relates to the top-level make.

6.6.1 How the MAKE Variable Works

Recursive make commands should always use the variable MAKE, not the explicit command name 'make', as shown here:

```
subsystem:
        cd subdir; $(MAKE)
```

The value of this variable is the file name with which make was invoked. If this file name was '/bin/make', then the command executed is 'cd subdir; /bin/make'. If you use a special version of make to run the top-level makefile, the same special version will be executed for recursive invocations.

Also, any arguments that define variable values are added to MAKE, so the sub-make gets them too. Thus, if you do 'make CFLAGS=-O', so that all C compilations will be optimized, the sub-make is run with 'cd subdir; /bin/make CFLAGS=-O'.

As a special feature, using the variable MAKE in the commands of a rule alters the effects of the '-t', '-n' or '-q' option. (See Section 10.3 [Instead of Execution], page 83.)

Consider the command 'make -t' in the above example. Following the usual definition of '-t', this would create a file named 'subsystem' and do nothing else. What you really want it to do is run '`cd subdir; make -t`'; but that would require executing the command, and '`-t`' says not to execute commands.

The special feature makes this do what you want: whenever a rule's commands use the variable `MAKE`, the flags '`-t`', '`-n`' or '`-q`' do not apply to that rule. The commands of that rule are executed normally despite the presence of a flag that causes most commands not to be run. The usual `MAKEFLAGS` mechanism passes the flags to the sub-`make` (see Section 6.6.3 [Options/Recursion], page 50), so your request to touch the files, or print the commands, is propagated to the subsystem.

6.6.2 Communicating Variables to a Sub-`make`

Most variable values of the top-level `make` are passed to the sub-`make` through the environment. These variables are defined in the sub-`make` as defaults, but do not override what is specified in the sub-`make`'s makefile.

Variables are passed down if their names consist only of letters, numbers and underscores. Some shells cannot cope with environment variable names consisting of characters other than letters, numbers, and underscores.

Variable are *not* passed down if they were created by default by `make` (see Section 11.3 [Implicit Variables], page 93). The sub-`make` will define these for itself.

The way this works is that `make` adds each variable and its value to the environment for running each command. The sub-`make`, in turn, uses the environment to initialize its table of variable values. See Section 7.8 [Environment], page 64.

As a special feature, the variable `MAKELEVEL` is changed when it is passed down from level to level. This variable's value is a string which is the depth of the level as a decimal number. The value is '`0`' for the top-level `make`; '`1`' for a sub-`make`, '`2`' for a sub-sub-`make`, and so on. The incrementation happens when `make` sets up the environment for a command.

The main use of `MAKELEVEL` is to test it in a conditional directive (see Chapter 8 [Conditionals], page 67); this way you can write a makefile that behaves one way if run recursively and another way if run directly by you.

You can use the variable `MAKEFILES` to cause all sub-`make` commands to use additional makefiles. The value of `MAKEFILES` is a whitespace-separated list of filenames. This variable, if defined in the outer-level makefile, is passed down through the environment as usual; then it serves as a list of extra makefiles for the sub-`make` to read before the usual or specified ones. See Section 4.4 [MAKEFILES Variable], page 21.

6.6.3 Communicating Options to a Sub-make

Flags such as '`-s`' and '`-k`' are passed automatically to the sub-`make` through the variable `MAKEFLAGS`. This variable is set up automatically by `make` to contain the flag letters that `make` received. Thus, if you do '`make -ks`' then `MAKEFLAGS` gets the value '`ks`'.

As a consequence, every sub-`make` gets a value for `MAKEFLAGS` in its environment. In response, it takes the flags from that value and processes them as if they had been given as arguments. See Section 10.7 [Options], page 86.

The options '`-C`', '`-f`', '`-I`', '`-o`', and '`-W`' are not put into `MAKEFLAGS`; these options are not passed down.

The '`-j`' (see Section 6.3 [Parallel], page 44) option is a special case. If you set it to some numeric value, '`-j 1`' is always put into `MAKEFLAGS` instead of the value you specified. This is because if the '`-j`' option were passed down to sub-`make`s, you would get many more jobs running in parallel than you asked for. If you give '`-j`' with no numeric argument, meaning to run as many jobs as possible in parallel, this is passed down, since multiple infinities are no more than one.

If you don't want to pass the other flags down, you must change the value of `MAKEFLAGS`, like this:

```
MAKEFLAGS=
subsystem:
        cd subdir; $(MAKE)
```

or like this:

```
subsystem:
        cd subdir; $(MAKE) MAKEFLAGS=
```

A similar variable `MFLAGS` exists also, for historical compatibility. It has the same value as `MAKEFLAGS` except that a hyphen is added at the beginning if it is not empty. `MFLAGS` was traditionally used explicitly in the recursive `make` command, like this:

```
subsystem:
        cd subdir; $(MAKE) $(MFLAGS)
```

but now `MAKEFLAGS` makes this usage redundant.

The `MAKEFLAGS` and `MFLAGS` variables can also be useful if you want to have certain options, such as '`-k`' (see Section 10.7 [Options], page 86) set each time you run `make`. Just put '`MAKEFLAGS=k`' or '`MFLAGS=-k`' in your environment. These variables may also be set in makefiles, so a makefile can specify additional flags that should also be in effect for that makefile.

If you do put `MAKEFLAGS` or `MFLAGS` in your environment, you should be sure not to include any options that will drastically affect the actions of `make` and undermine the purpose of makefiles and of `make` itself. For instance, the '`-t`', '`-n`', and '`-q`' options, if put in one of these variables, could have disastrous consequences and would certainly have at least surprising and probably annoying effects.

6.6.4 The '-w' Option

If you use several levels of recursive `make` invocations, the '`-w`' option can make the output a lot easier to understand by showing each directory as `make` starts processing it and as `make` finishes processing it. For example, if '`make -w`' is run in the directory '`/u/gnu/make`', `make` will print a line of the form:

```
make: Entering directory `/u/gnu/make'.
```

before doing anything else, and a line of the form:

```
make: Leaving directory `/u/gnu/make'.
```

when processing is completed.

6.7 Defining Canned Command Sequences

When the same sequence of commands is useful in making various targets, you can define it as a canned sequence with the `define` directive, and refer to the canned sequence from the rules for those targets. The canned sequence is actually a variable, so the name must not conflict with other variable names.

Here is an example of defining a canned sequence of commands:

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef
```

Here `run-yacc` is the name of the variable being defined; `endef` marks the end of the definition; the lines in between are the commands. The `define` directive does not expand variable references and function calls in the canned sequence; the '`$`' characters, parentheses, variable names, and so on, all become part of the value of the variable you are defining. See Section 7.7 [Defining], page 63, for a complete explanation of `define`.

The first command in this example runs Yacc on the first dependency (of whichever rule uses the canned sequence). The output file from Yacc is always named '`y.tab.c`'. The second command moves the output to the rule's target file name.

To use the canned sequence, substitute the variable into the commands of a rule. You can substitute it like any other variable (see Section 7.1 [Reference], page 55). Because variables defined by `define` are recursively expanded variables, all the variable references you wrote inside the `define` are expanded now. For example:

```
foo.c : foo.y
        $(run-yacc)
```

'`foo.y`' will substituted for the variable '`$^`' when it occurs in `run-yacc`'s value, and '`foo.c`' for '`$@`'.

This is a realistic example, but this particular one is not needed in practice because `make` has an implicit rule to figure out these commands based on the file names involved. See Chapter 11 [Implicit], page 89.

6.8 Defining Empty Commands

It is sometimes useful to define commands which do nothing. This is done simply by giving a command that consists of nothing but whitespace. For example:

```
target:;
```

defines an empty command string for '`target`'. You could also use a line beginning with a tab character to define an empty command string, but this would be confusing because such a line looks empty.

You may be wondering why you would want to define a command string that does nothing. The only reason this is useful is to prevent a target from getting implicit commands (from implicit rules or the `.DEFAULT` special target; see Chapter 11 [Implicit], page 89 and see Section 11.6 [Last Resort], page 103).

You may be inclined to define empty command strings for targets that are not actual files, but only exist so that their dependencies can be remade. However, this is not the best way to do that, because if the target file actually does exist, its dependencies may not be remade. See Section 5.4 [Phony Targets], page 32, for a better way to do this.

7 How to Use Variables

A *variable* is a name defined within `make` to represent a string of text, called the variable's *value*. These values can be substituted by explicit request into targets, dependencies, commands and other parts of the makefile.

Variables can represent lists of file names, options to pass to compilers, programs to run, directories to look in for source files, directories to write output in, or anything else you can imagine.

A variable name may be any sequence characters not containing '`:`', '`#`', '`=`', or leading or trailing whitespace. However, variable names containing characters other than letters, numbers and underscores should be avoided, as they may be given special meanings in the future, and they are not passed through the environment to a sub-`make` (see Section 6.6.2 [Variables/Recursion], page 49).

It is traditional to use upper case letters in variable names, but we recommend using lower case letters for variable names that serve internal purposes in the makefile, and reserving upper case for parameters that control implicit rules or for parameters that the user should override with command options (see Section 10.5 [Overriding], page 85).

7.1 Basics of Variable References

To substitute a variable's value, write a dollar sign followed by the name of the variable in parentheses or braces: either '`$(foo)`' or '`${foo}`' is a valid reference to the variable `foo`. This special significance of '`$`' is why you must write '`$$`' to have the effect of a single dollar sign in a file name or command.

Variable references can be used in any context: targets, dependencies, commands, most directives, and new variable values. Here is a common kind of example, where a variable holds the names of all the object files in a program:

```
objects = program.o foo.o utils.o
program : $(objects)
        cc -o program $(objects)

$(objects) : defs.h
```

Variable references work by strict textual substitution. Thus, the rule

```
foo = c
prog.o : prog.c
        $(foo)$(foo) prog.c
```

could be used to compile a C program '`prog.c`'. Since spaces around the variable value are ignored in variable assignments, the value of `foo` is precisely '`c`'. (Don't actually write your makefiles this way!)

A dollar sign followed by a character other than a dollar sign, open-parenthesis or open-brace treats that single character as the variable name. Thus, you could reference the variable `x` with '`$x`'. However, this practice is strongly discouraged, except in the case of the automatic variables (see Section 11.5.3 [Automatic], page 99).

7.2 The Two Flavors of Variables

There are two ways that a variables in GNU `make` can have a value; we call them two *flavors* of variables. The two flavors are distinguished in how they are defined and in what they do when expanded.

The first flavor of variable is a *recursively expanded* variable. Variables of this sort are defined by lines using '`=`' (see Section 7.5 [Setting], page 62). The value you specify is installed verbatim; if it contains references to other variables, these references are expanded whenever this variable is substituted (in the course of expanding some other string). When this happens, it is called *recursive expansion*.

For example,

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:;echo $(foo)
```

will echo '`Huh?`': '`$(foo)`' expands to '`$(bar)`' which expands to '`$(ugh)`' which finally expands to '`Huh?`'.

This flavor of variable is the only sort supported by other versions of `make`. It has its advantages and its disadvantages. An advantage (most would say) is that:

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

will do what was intended: when '`CFLAGS`' is expanded in a command, it will expand to '`-Ifoo -Ibar -O`'. A major disadvantage is that you can't append something on the end of a variable, as in

```
CFLAGS = $(CFLAGS) -O
```

because it will cause an infinite loop in the variable expansion. (Actually **make** detects the infinite loop and reports an error.)

Another disadvantage is that any functions (see Chapter 9 [Functions], page 71) referenced in the definition will be executed every time the variable is expanded. This makes **make** run slower; worse, it causes the `wildcard` and `shell` functions to give unpredictable results because you cannot easily control when they are called, or even how many times.

To avoid all the problems and inconveniences of recursively expanded variables, there is another flavor: *simply expanded* variables.

Simply expanded variables are defined by lines using '`:=`' (see Section 7.5 [Setting], page 62). The value of a simply expanded variable is scanned once and for all, expanding any references to other variables and functions, when the variable is defined. The actual value of the simply expanded variable is the result of expanding the text that you write. It does not contain any references to other variables; it contains their values *as of the time this variable was defined*. Therefore,

```
x := foo
y := $(x) bar
x := later
```

is equivalent to

```
y := foo bar
x := later
```

When a simply expanded variable is referenced, its value is substituted verbatim.

Simply expanded variables generally make complicated makefile programming more predictable because they work like variables in most programming languages. They allow you to redefine a variable using its own value (or its value processed in some way by one of the expansion functions) and to use the expansion functions much more efficiently (see Chapter 9 [Functions], page 71).

You can also use them to introduce controlled leading or trailing spaces into variable values. Such spaces are discarded from your input before substitution of variable references and function calls; this means you can include leading or trailing spaces in a variable value by protecting them with variable references, like this:

```
nullstring :=
space := $(nullstring) $(nullstring)
```

Here the value of the variable `space` is precisely one space.

7.3 Advanced Features for Reference to Variables

This section describes some advanced features you can use to reference variables in more flexible ways.

7.3.1 Substitution References

A *substitution reference* substitutes the value of a variable with alterations that you specify. It has the form '`$(`*var*`:`*a*`=`*b*`)`' (or '`${`*var*`:`*a*`=`*b*`}`') and its meaning is to take the value of the variable *var*, replace every *a* at the end of a word with *b* in that value, and substitute the resulting string.

When we say "at the end of a word", we mean that *a* must appear either followed by whitespace or at the end of the value in order to be replaced; other occurrences of *a* in the value are unaltered. For example:

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

sets '`bar`' to '`a.c b.c c.c`'. See Section 7.5 [Setting], page 62.

A substitution reference is actually an abbreviation for use of the `patsubst` expansion function (see Section 9.2 [Text Functions], page 72). We provide substitution references as well as `patsubst` for compatibility with other implementations of `make`.

Another type of substitution reference lets you use the full power of the `patsubst` function. It has the same form '`$(`*var*`:`*a*`=`*b*`)`' described above, except that now *a* must contain a single '`%`' character. This case is equivalent to '`$(patsubst` *a*`,`*b*`,$(`*var*`))`'. See Section 9.2 [Text Functions], page 72, for a description of the `patsubst` function. For example:

```
foo := a.o b.o c.o
bar := $(foo:%.o=%.c)
```

sets '`bar`' to '`a.c b.c c.c`'.

7.3.2 Computed Variable Names

Computed variable names are a complicated concept needed only for sophisticated makefile programming. For most purposes you need not consider about them, except to know that making a variable with a dollar sign in its name might have strange results. However, if you are the type that wants to understand everything, or you are actually interested in what they do, read on.

Variables may be referenced inside the name of a variable. This is called a *computed variable name* or a *nested variable reference*. For example,

```
x = y
y = z
a := $($(x))
```

defines `a` as '`z`': the '`$(x)`' inside '`$($(x))`' expands to '`y`', so '`$($(x))`' expands to '`$(y)`' which in turn expands to '`z`'. Here the name of the variable to reference is not stated explicitly; it is computed by expansion of '`$(x)`'. The reference '`$(x)`' here is nested within the outer variable reference.

The previous example shows two levels of nesting, but any number of levels is possible. For example, here are three levels:

```
x = y
y = z
z = u
a := $($($(x)))
```

Here the innermost '`$(x)`' expands to '`y`', so '`$($(x))`' expands to '`$(y)`' which in turn expands to '`z`'; now we have '`$(z)`', which becomes '`u`'.

References to recursively-expanded variables within a variable name are reexpanded in the usual fashion. For example:

```
x = $(y)
y = z
z = Hello
a := $($(x))
```

defines `a` as '`Hello`': '`$($(x))`' becomes '`$($(y))`' which becomes '`$(z)`' which becomes '`Hello`'.

Nested variable references can also contain modified references and function invocations (see Chapter 9 [Functions], page 71), just like any other reference. For example, using the `subst` function (see Section 9.2 [Text Functions], page 72):

```
x = variable1
variable2 := Hello
y = $(subst 1,2,$(x))
z = y
a := $($($(z)))
```

eventually defines `a` as '`Hello`'. It is doubtful that anyone would ever want to write a nested reference as convoluted as this one, but it works: '`$($($(z)))`' expands to '`$($(y))`' which becomes '`$($(subst 1,2,$(x)))`'. This gets the value '`variable1`' from `x` and changes it by substitution to '`variable2`', so that the entire string becomes '`$(variable2)`', a simple variable reference whose value is '`Hello`'.

A computed variable name need not consist entirely of a single variable reference. It can contain several variable references, as well as some invariant text. For example,

```
a_dirs := dira dirb
1_dirs := dir1 dir2

a_files := filea fileb
1_files := file1 file2

ifeq "$(use_a)" "yes"
a1 := a
else
a1 := 1
endif

ifeq "$(use_dirs)" "yes"
df := dirs
```

```
else
df := files
endif

dirs := $($(a1)_$(df))
```

will give `dirs` the same value as `a_dirs`, `1_dirs`, `a_files` or `1_files` depending on the settings of `use_a` and `use_dirs`.

Computed variable names can also be used in substitution references:

```
a_objects := a.o b.o c.o
1_objects := 1.o 2.o 3.o

sources := $($(a1)_object:.o=.c)
```

defines `sources` as either '`a.c b.c c.c`' or '`1.c 2.c 3.c`', depending on the value of `a1`.

The only restriction on this sort of use of nested variable references is that they cannot specify part of the name of a function to be called. This is because the test for a recognized function name is done before the expansion of nested references. For example,

```
ifdef do_sort
func := sort
else
func := strip
endif

bar := a d b g q c

foo := $($(func) $(bar))
```

attempts to give '`foo`' the value of the variable '`sort a d b g q c`' or '`strip a d b g q c`', rather than giving '`a d b g q c`' as the argument to either the `sort` or the `strip` function. This restriction could be removed in the future if that change is shown to be a good idea.

Note that *nested variable references* are quite different from *recursively expanded variables* (see Section 7.2 [Flavors], page 56), though both are used together in complex ways when doing makefile programming.

7.4 How Variables Get Their Values

Variables can get values in several different ways:

- You can specify an overriding value when you run `make`. See Section 10.5 [Overriding], page 85.
- You can specify a value in the makefile, either with an assignment (see Section 7.5 [Setting], page 62) or with a verbatim definition (see Section 7.7 [Defining], page 63).
- Values are inherited from the environment. See Section 7.8 [Environment], page 64.
- Several *automatic* variables are given new values for each rule. Each of these has a single conventional use. See Section 11.5.3 [Automatic], page 99.
- Several variables have constant initial values. See Section 11.3 [Implicit Variables], page 93.

7.5 Setting Variables

To set a variable from the makefile, write a line starting with the variable name followed by '`=`' or '`:=`'. Whatever follows the '`=`' or '`:=`' on the line becomes the value. For example,

```
objects = main.o foo.o bar.o utils.o
```

defines a variable named `objects`. Whitespace around the variable name and immediately after the '`=`' is ignored.

Variables defined with '`=`' are *recursively expanded* variables. Variables defined with '`:=`' are *simply expanded* variables; these definitions can contain variable references which will be expanded before the definition is made. See Section 7.2 [Flavors], page 56.

There is no limit on the length of the value of a variable except the amount of swapping space on the computer. When a variable definition is long, it is a good idea to break it into several lines by inserting backslash-newline at convenient places in the definition. This will not affect the functioning of `make`, but it will make the makefile easier to read.

Most variable names are considered to have the empty string as a value if you have never set them. Several variables have built-in initial values that are not empty, but can be set by you in the usual ways (see Section 11.3 [Implicit Variables], page 93). Several special variables are set automatically to a new value for each rule; these are called the *automatic* variables (see Section 11.5.3 [Automatic], page 99).

7.6 The `override` Directive

If a variable has been set with a command argument (see Section 10.5 [Overriding], page 85), then ordinary assignments in the makefile are ignored. If you want to set the variable in the makefile even though it was set with a command argument, you can use an `override` directive, which is a line that looks like this:

```
override variable = value
```

or

```
override variable := value
```

The `override` directive was not invented for escalation in the war between makefiles and command arguments. It was invented so you can alter and add to values that the user specifies with command arguments.

For example, suppose you always want the '`-g`' switch when you run the C compiler, but you would like to allow the user to specify the other switches with a command argument just as usual. You could use this `override` directive:

```
override CFLAGS := $(CFLAGS) -g
```

You can also use `override` directives with `define` directives. This is done as you might expect:

```
override define foo
bar
endef
```

See the next section.

7.7 Defining Variables Verbatim

Another way to set the value of a variable is to use the `define` directive. This directive has a different syntax which allows newline characters to be included in the value, which is convenient for defining canned sequences of commands (see Section 6.7 [Sequences], page 51).

The `define` directive is followed on the same line by the name of the variable and nothing more. The value to give the variable appears on the following lines. The end of the value is marked by a line containing just the word `endef`. Aside from this difference in syntax, `define` works just like '`=`'; it creates a recursively-expanded variable (see Section 7.2 [Flavors], page 56).

```
define two-lines
echo foo
echo $(bar)
endef
```

The value in an ordinary assignment cannot contain a newline; but the newlines that separate the lines of the value in a `define` become part of the variable's value (except for the final newline which precedes the `endef` and is not considered part of the value).

The previous example is functionally equivalent to this:

```
two-lines = echo foo; echo $(bar)
```

since the shell will interpret the semicolon and the newline identically.

If you want variable definitions made with `define` to take precedence over command-line variable definitions, the `override` directive can be used together with `define`:

```
override define two-lines
foo
$(bar)
endef
```

See Section 7.6 [Override Directive], page 63.

7.8 Variables from the Environment

Variables in `make` can come from the environment with which `make` is run. Every environment variable that `make` sees when it starts up is transformed into a `make` variable with the same name and value. But an explicit assignment in the makefile, or with a command argument, overrides the environment. (If the '`-e`' flag is specified, then values from the environment override assignments in the makefile. See Section 10.7 [Options], page 86. But this is not recommended practice.)

Thus, by setting the variable `CFLAGS` in your environment, you can cause all C compilations in most makefiles to use the compiler switches you prefer. This is safe for variables with standard or conventional meanings because you know that no makefile will use them for other things. (But this is not totally reliable; some makefiles set `CFLAGS` explicitly and therefore are not affected by the value in the environment.)

When `make` is invoked recursively, variables defined in the outer invocation are automatically passed to inner invocations through the environment (see Section 6.6 [Recursion], page 47). This is the main purpose of turning environment variables into `make` variables, and it requires no attention from you.

Other use of variables from the environment is not recommended. It is not wise for makefiles to depend for their functioning on environment variables set up outside their control, since this would cause different users to get different results from the same makefile. This is against the whole purpose of most makefiles.

Such problems would be especially likely with the variable `SHELL`, which is normally present in the environment to specify the user's choice of interactive shell. It would be very undesirable for this choice to affect `make`. So `make` ignores the environment value of `SHELL` if the value of `MAKELEVEL` is zero (which is normally true except in recursive invocations of `make`).

8 Conditional Parts of Makefiles

A *conditional* causes part of a makefile to be obeyed or ignored depending on the values of variables. Conditionals can compare the value of one variable with another, or the value of a variable with a constant string. Conditionals control what `make` actually "sees" in the makefile, so they *cannot* be used to control shell commands at the time of execution.

8.1 Example of a Conditional

This conditional tells `make` to use one set of libraries if the `CC` variable is '`gcc`', and a different set of libraries otherwise. It works by controlling which of two command lines will be used as the command for a rule. The result is that '`CC=gcc`' as an argument to `make` changes not only which compiler is used but also which libraries are linked.

```
libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

This conditional uses three directives: one `ifeq`, one `else` and one `endif`.

The `ifeq` directive begins the conditional, and specifies the condition. It contains two arguments, separated by a comma and surrounded by parentheses. Variable substitution is performed on both arguments and then they are compared. The lines of the makefile following the `ifeq` are obeyed if the two arguments match; otherwise they are ignored.

The `else` directive causes the following lines to be obeyed if the previous conditional failed. In the example above, this means that the second alternative linking command is used whenever the first alternative is not used. It is optional to have an `else` in a conditional.

The `endif` directive ends the conditional. Every conditional must end with an `endif`. Unconditional makefile text follows.

Conditionals work at the textual level: the lines of the conditional are treated as part of the makefile, or ignored, according to the condition. This is why the larger syntactic units of the makefile, such as rules, may cross the beginning or the end of the conditional.

When the variable `CC` has the value '`gcc`', the above example has this effect:

```
foo: $(objects)
        $(CC) -o foo $(objects) $(libs_for_gcc)
```

When the variable `CC` has any other value, the effect is this:

```
foo: $(objects)
        $(CC) -o foo $(objects) $(normal_libs)
```

Equivalent results can be obtained in another way by conditionalizing a variable assignment and then using the variable unconditionally:

```
libs_for_gcc = -lgnu
normal_libs =

ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif

foo: $(objects)
        $(CC) -o foo $(objects) $(libs)
```

8.2 Syntax of Conditionals

The syntax of a simple conditional with no `else` is as follows:

conditional-directive
text-if-true
`endif`

The *text-if-true* may be any lines of text, to be considered as part of the makefile if the condition is true. If the condition is false, no text is used instead.

The syntax of a complex conditional is as follows:

conditional-directive
text-if-true
`else`
text-if-false
`endif`

If the condition is true, *text-if-true* is used; otherwise, *text-if-false* is used instead. The *text-if-false* can be any number of lines of text.

The syntax of the *conditional-directive* is the same whether the conditional is simple or complex. There are four different directives that test different conditions. Here is a table of them:

`ifeq (`*arg1*`, `*arg2*`)`
`ifeq '`*arg1*`' '`*arg2*`'`
`ifeq "`*arg1*`" "`*arg2*`"`
`ifeq "`*arg1*`" '`*arg2*`'`
`ifeq '`*arg1*`' "`*arg2*`"`

: Expand all variable references in *arg1* and *arg2* and compare them. If they are identical, the *text-if-true* is effective; otherwise, the *text-if-false*, if any, is effective.

`ifneq (`*arg1*`, `*arg2*`)`
`ifneq '`*arg1*`' '`*arg2*`'`
`ifneq "`*arg1*`" "`*arg2*`"`
`ifneq "`*arg1*`" '`*arg2*`'`
`ifneq '`*arg1*`' "`*arg2*`"`

: Expand all variable references in *arg1* and *arg2* and compare them. If they are different, the *text-if-true* is effective; otherwise, the *text-if-false*, if any, is effective.

`ifdef` *variable-name*

: If the variable *variable-name* has a non-empty value, the *text-if-true* is effective; otherwise, the *text-if-false*, if any, is effective. Variables that have never been defined have an empty value.

`ifndef` *variable-name*

: If the variable *variable-name* has an empty value, the *text-if-true* is effective; otherwise, the *text-if-false*, if any, is effective.

Extra spaces are allowed and ignored at the beginning of the conditional directive line, but a tab is not allowed. (If the line begins with a tab, it will be considered a command for a rule.) Aside from this, extra spaces or tabs may be inserted with no effect anywhere except within the directive name or within an argument. A comment starting with '`#`' may appear at the end of the line.

The other two directives that play a part in a conditional are `else` and `endif`. Each of these directives is written as one word, with no arguments. Extra spaces are allowed and ignored at the beginning of the line, and spaces or tabs at the end. A comment starting with '`#`' may appear at the end of the line.

Conditionals work at the textual level. The lines of the *text-if-true* are read as part of the makefile if the condition is true; if the condition is false, those lines are ignored completely. It follows that syntactic units of the makefile, such as rules, may safely be split across the beginning or the end of the conditional.

To prevent intolerable confusion, it is not permitted to start a conditional in one makefile and end it in another. However, you may write an `include` directive within a conditional, provided you do not attempt to terminate the conditional inside the included file.

8.3 Conditionals that Test Flags

You can write a conditional that tests `make` command flags such as '`-t`' by using the variable `MAKEFLAGS` together with the `findstring` function. This is useful when `touch` is not enough to make a file appear up to date.

The `findstring` function determines whether one string appears as a substring of another. If you want to test for the '`-t`' flag, use '`t`' as the first string and the value of `MAKEFLAGS` as the other.

For example, here is how to arrange to use '`ranlib -t`' to finish marking an archive file up to date:

```
archive.a: ...
ifneq (,$(findstring t,$(MAKEFLAGS)))
        @echo $(MAKE) > /dev/null
        touch archive.a
        ranlib -t archive.a
else
        ranlib archive.a
endif
```

The `echo` command does nothing when executed; but its presence, with a reference to the variable `MAKE`, marks the rule as "recursive" so that its commands will be executed despite use of the '`-t`' flag. See Section 6.6 [Recursion], page 47.

9 Functions for Transforming Text

Functions allow you to do text processing in the makefile to compute the files to operate on or the commands to use. You use a function in a *function call*, where you give the name of the function and some text (the *arguments*) for the function to operate on. The result of the function's processing is substituted into the makefile at the point of the call, just as a variable might be substituted.

9.1 Function Call Syntax

A function call resembles a variable reference. It looks like this:

`$(`*function arguments*`)`

or like this:

`${`*function arguments*`}`

Here *function* is a function name; one of a short list of names that are part of `make`. There is no provision for defining new functions.

The *arguments* are the arguments of the function. They are separated from the function name by one or more spaces and/or tabs, and if there is more than one argument they are separated by commas. Such whitespace and commas are not part of any argument's value. The delimiters which you use to surround the function call, whether parentheses or braces, can appear in an argument only in matching pairs; the other kind of delimiters may appear singly. If the arguments themselves contain other function calls or variable references, it is wisest to use the same kind of delimiters for all the references; in other words, write '`$(subst a,b,$(x))`', not '`$(subst a,b,${x})`'. This is both because it is clearer, and because only one type of delimiters is matched to find the end of the reference. Thus in '`$(subst a,b,${subst c,d,${x}})`' doesn't work because the second `subst` function invocation ends at the first '`}`', not the second.

The text written for each argument is processed by substitution of variables and function calls to produce the argument value, which is the text on which the function acts. The substitution is done in the order in which the arguments appear.

Commas and unmatched parentheses or braces cannot appear in the text of an argument as written; leading spaces cannot appear in the text of the first argument as written. These characters can be put into the argument value by variable substitution. First define variables `comma` and `space` whose values are isolated comma and space characters, then substitute those variables where such characters are wanted, like this:

```
comma:= ,
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
# bar is now 'a,b,c'.
```

Here the `subst` function replaces each space with a comma, through the value of `foo`, and substitutes the result.

9.2 Functions for String Substitution and Analysis

Here are some functions that operate on strings:

`$(subst` *from*`,`*to*`,`*text*`)`

Performs a textual replacement on the text *text*: each occurrence of *from* is replaced by *to*. The result is substituted for the function call. For example,

```
$(subst ee,EE,feet on the street)
```

substitutes the string '`fEEt on the strEEt`'.

`$(patsubst` *pattern*`,`*replacement*`,`*text*`)`

Finds whitespace-separated words in *text* that match *pattern* and replaces them with *replacement*. Here *pattern* may contain a '`%`' which acts as a wildcard, matching any number of any characters within a word. If *replacement* also contains a '`%`', the '`%`' is replaced by the text that matched the '`%`' in *pattern*.

'`%`' characters in `patsubst` function invocations can be quoted with preceding backslashes ('`\`'). Backslashes that would otherwise quote '`%`' characters can be quoted with more backslashes. Backslashes that quote '`%`' characters or other backslashes are removed from the pattern before it is compared file names or has a stem substituted into it. Backslashes that are not in danger of quoting '`%`' characters go unmolested. For example, the pattern '`the\%weird\\%pattern\\`' has '`the%weird\`' preceding the operative '`%`' character, and '`pattern\\`' following it. The final two backslashes are left alone because they can't affect any '`%`' character.

Whitespace between words is folded into single space characters; leading and trailing whitespace is discarded.

For example,

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

produces the value '`x.c.o bar.o`'.

`$(strip` *string*`)`
: Removes leading and trailing whitespace from *string* and replaces each internal sequence of one or more whitespace characters with a single space. Thus, '`$(strip a b c )`' results in '`a b c`'.

`$(findstring` *find*`,`*in*`)`
: Searches *in* for an occurrence of *find*. If it occurs, the value is *find*; otherwise, the value is empty. You can use this function in a conditional to test for the presence of a specific substring in a given string. Thus, the two examples,

```
$(findstring a,a b c)
$(findstring a,b c)
```

produce the values '`a`' and '', respectively. See Section 8.3 [Testing Flags], page 70, for a practical application of `findstring`.

`$(filter` *pattern*`,`*text*`)`
: Removes all whitespace-separated words in *text* that do *not* match *pattern*, returning only matching words. The pattern is written using '`%`', just like the patterns used in `patsubst` function above.

The `filter` function can be used to separate out different types of strings (such as filenames) in a variable. For example:

```
sources := foo.c bar.c ugh.h
foo: $(sources)
        cc $(filter %.c,$(sources)) -o foo
```

says that '`foo`' depends of '`foo.c`', '`bar.c`' and '`ugh.h`' but only '`foo.c`' and '`bar.c`' should be specified in the command to the compiler.

`$(filter-out` *pattern*`,`*text*`)`
: Removes all whitespace-separated words in *text* that *do* match *pattern*, returning only the words that match. This is the exact opposite of the `filter` function.

`$(sort` *list*`)`
: Sorts the words of *list* in lexical order, removing duplicate words. The output is a list of words separated by single spaces. Thus,

```
$(sort foo bar lose)
```

returns the value '`bar foo lose`'.

Here is a realistic example of the use of `subst` and `patsubst`. Suppose that a makefile uses the `VPATH` variable to specify a list of directories that `make` should search for dependency files. This example shows how to tell the C compiler to search for header files in the same list of directories.

The value of VPATH is a list of directories separated by colons, such as 'src:../headers'. First, the subst function is used to change the colons to spaces:

```
$(subst :, ,$(VPATH))
```

This produces 'src ../headers'. Then patsubst is used to turn each directory name into a '-I' flag. These can be added to the value of the variable CFLAGS, which is passed automatically to the C compiler, like this:

```
override CFLAGS:= $(CFLAGS) $(patsubst %,-I%,$(subst :, ,$(VPATH)))
```

The effect is to append the text '-Isrc -I../headers' to the previously given value of CFLAGS. The override directive is used so that the new value is assigned even if the previous value of CFLAGS was specified with a command argument (see Section 7.6 [Override Directive], page 63).

The function strip can be very useful when used in conjunction with conditionals. When comparing something with the null string '""' using ifeq or ifneq, you usually want a string of just whitespace to match the null string. Thus,

```
.PHONY: all
ifneq   "$(needs_made)" ""
all: $(needs_made)
else
all:;@echo 'Nothing to make!'
endif
```

might fail to have the desired results. Replacing the variable reference '"$(needs_made)"' with the function call '"$(strip $(needs_made))"' in the ifneq directive would make it more robust.

9.3 Functions for File Names

Several of the built-in expansion functions relate specifically to taking apart file names or lists of file names.

Each of the following functions performs a specific transformation on a file name. The argument of the function is regarded as a series of file names, separated by whitespace. (Leading and trailing whitespace is ignored.) Each file name in the series is transformed in the same way and the results are concatenated with single spaces between them.

`$(dir names)`

Extracts the directory-part of each file name in *names*. The directory-part of the file name is everything up through (and including) the last slash in it. If the file name contains no slash, the directory part is the string '`./`'. For example,

```
$(dir src/foo.c hacks)
```

produces the result '`src/ ./`'.

`$(notdir names)`

Extracts all but the directory-part of each file name in *names*. If the file name contains no slash, it is left unchanged. Otherwise, everything through the last slash is removed from it.

A file name that ends with a slash becomes an empty string. This is unfortunate, because it means that the result does not always have the same number of whitespace-separated file names as the argument had; but we do not see any other valid alternative.

For example,

```
$(notdir src/foo.c hacks)
```

produces the result '`foo.c hacks`'.

`$(suffix names)`

Extracts the suffix of each file name in *names*. If the file name contains a period, the suffix is everything starting with the last period. Otherwise, the suffix is the empty string. This frequently means that the result will be empty when *names* is not, and if *names* contains multiple file names, the result may contain fewer file names.

For example,

```
$(suffix src/foo.c hacks)
```

produces the result '`.c`'.

`$(basename names)`

Extracts all but the suffix of each file name in *names*. If the file name contains a period, the basename is everything starting up to (and not including) the last period. Otherwise, the basename is the entire file name. For example,

```
$(basename src/foo.c hacks)
```

produces the result '`src/foo hacks`'.

`$(addsuffix suffix,names)`

The argument *names* is regarded as a series of names, separated by whitespace; *suffix* is used as a unit. The value of *suffix* is appended to the end of each individual name and the resulting larger names are concatenated with single spaces between them. For example,

```
$(addsuffix .c,foo bar)
```

produces the result '`foo.c bar.c`'.

`$(addprefix` *prefix*`,`*names*`)`

The argument *names* is regarded as a series of names, separated by whitespace; *prefix* is used as a unit. The value of *prefix* is appended to the front of each individual name and the resulting larger names are concatenated with single spaces between them. For example,

```
$(addprefix src/,foo bar)
```

produces the result '`src/foo src/bar`'.

`$(join` *list1*`,`*list2*`)`

Concatenates the two arguments word by word: the two first words (one from each argument) concatenated form the first word of the result, the two second words form the second word of the result, and so on. So the *n*th word of the result comes from the *n*th word of each argument. If one argument has more words that the other, the extra words are copied unchanged into the result.

For example, '`$(join a b,.c .o)`' produces '`a.c b.o`'.

Whitespace between the words in the lists is not preserved; it is replaced with a single space.

This function can merge the results of the `dir` and `notdir` functions, to produce the original list of files which was given to those two functions.

`$(word` *n*`,`*text*`)`

Returns the *n*th word of *text*. The legitimate values of *n* start from 1. If *n* is bigger than the number of words in *text*, the value is empty. For example,

```
$(word 2, foo bar baz)
```

returns '`bar`'.

`$(words` *text*`)`

Returns the number of words in *text*. Thus, `$(word $(words` *text*`),`*text*`)` is the last word of *text*.

`$(firstword` *names*`)`

The argument *names* is regarded as a series of names, separated by whitespace. The value is the first name in the series. The rest of the names are ignored. For example,

```
$(firstword foo bar)
```

produces the result '`foo`'. Although `$(firstword` *text*`)` is the same as `$(word 1,`*text*`)`, the `firstword` function is retained for its simplicity.

`$(wildcard` *pattern*`)`

The argument *pattern* is a file name pattern, typically containing wildcard characters. The result of `wildcard` is a space-separated list of the names of existing files that match the pattern.

Wildcards are expanded automatically in rules. See Section 5.2 [Wildcards], page 26. But they are not normally expanded when a variable is set, or inside the arguments of other functions. Those occasions are when the `wildcard` function is useful.

9.4 The `foreach` Function

The `foreach` function is very different from other functions. It causes one piece of text to be used repeatedly, each time with a different substitution performed on it. It resembles the `for` command in the shell `sh` and the `foreach` command in the C-shell `csh`.

The syntax of the `foreach` function is:

`$(foreach` *var*`,`*list*`,`*text*`)`

The first two arguments, *var* and *list*, are expanded before anything else is done; note that the last argument, *text*, is *not* expanded at the same time. Then for each word of the expanded value of *list*, the variable named by the expanded value of *var* is set to that word, and *text* is expanded. Presumably *text* contains references to that variable, so its expansion will be different each time.

The result is that *text* is expanded as many times as there are whitespace-separated words in *list*. The multiple expansions of *text* are concatenated, with spaces between them, to make the result of `foreach`.

This simple example sets the variable '`files`' to the list of all files in the directories in the list '`dirs`':

```
dirs := a b c d
files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

Here *text* is '`$(wildcard $(dir)/*)`'. The first repetition finds the value '`a`' for `dir`, so it produces the same result as '`$(wildcard a/*)`'; the second repetition produces the result of '`$(wildcard b/*)`'; and the third, that of '`$(wildcard c/*)`'.

This example has the same result (except for setting '`find_files`', '`dirs`' and '`dir`') as the following example:

```
files := $(wildcard a/* b/* c/* d/*)
```

When *text* is complicated, you can improve readability by giving it a name, with an additional variable:

```
find_files = $(wildcard $(dir)/*)
dirs := a b c d
files := $(foreach dir,$(dirs),$(find_files))
```

Here we use the variable `find_files` this way. We use plain '`=`' to define a recursively-expanding variable, so that its value contains an actual function call to be reexpanded under the control of `foreach`; a simply-expanded variable would not do, since `wildcard` would be called only once at the time of defining `find_files`.

The `foreach` function has no permanent effect on the variable *var*; its value and flavor after the `foreach` function call are the same as they were beforehand. The other values which are taken from *list* are in effect only temporarily, during the execution of `foreach`. The variable *var* is a simply-expanded variable during the execution of `foreach`. If *var* was undefined before the `foreach` function call, it is undefined after the call. See Section 7.2 [Flavors], page 56.

You must take care when using complex variable expressions that result in variable names because many strange things are valid variable names, but are probably not what you intended. For example,

```
files := $(foreach Es escrito en espanol!,b c ch,$(find_files))
```

might be useful if the value of `find_files` references the variable whose name is '`Es escrito en espanol!`' (es un nombre bastante largo, que no?), but it is more likely to be a mistake.

9.5 The `origin` Function

The `origin` function is unlike most other functions in that it does not operate on the values of variables; it tells you something *about* a variable. Specifically, it tells you where it came from.

The syntax of the `origin` function is:

```
$(origin variable)
```

Note that *variable* is the *name* of a variable to inquire about; not a *reference* to that variable. Therefore you would not normally use a '`$`' or parentheses when writing it. (You can, however, use a variable reference in the name if you want the name not to be a constant.)

The result of this function is a string telling you how the variable *variable* was defined:

'`undefined`'
: if *variable* was never defined.

'`default`'
: if *variable* has a default definition, as is usual with `CC` and so on. See Section 11.3 [Implicit Variables], page 93. Note that if you have redefined a default variable, the `origin` function will return the origin of the later definition.

'`environment`'
: if *variable* was defined as an environment variable and the '`-e`' option is *not* turned on (see Section 10.7 [Options], page 86).

'`environment override`'
: if *variable* was defined as an environment variable and the '`-e`' option *is* turned on (see Section 10.7 [Options], page 86).

'`file`'
: if *variable* was defined in a makefile.

'`command line`'
: if *variable* was defined on the command line.

'`override`'
: if *variable* was defined with an `override` directive in a makefile (see Section 7.6 [Override Directive], page 63).

'`automatic`'
: if *variable* is an automatic variable defined for the execution of the commands for each rule.

This information is primarily useful (other than for your curiosity) to determine if you want to believe the value of a variable. For example, suppose you have a makefile '`foo`' that includes another makefile '`bar`'. You want a variable `bletch` to be defined in '`bar`' if you run the command '`make -f bar`', even if the environment contains a definition of `bletch`. However, if '`foo`' defined `bletch` before including '`bar`', you don't want to override that definition. This could be done by using an `override` directive in '`foo`', giving that definition precedence over the later definition in '`bar`'; unfortunately, the `override` directive would also override any command line definitions. So, '`bar`' could include:

```
ifdef bletch
ifeq "$(origin bletch)" "environment"
bletch = barf, gag, etc.
endif
endif
```

If `bletch` has been defined from the environment, this will redefine it.

If you want to override a previous definition of `bletch` if it came from the environment, even under '`-e`', you could instead write:

```
ifneq "$(findstring environment,$(origin bletch))" ""
bletch = barf, gag, etc.
endif
```

Here the redefinition takes place if '`$(origin bletch)`' returns either '`environment`' or '`environment override`'.

9.6 The `shell` Function

The `shell` function is unlike any other function except the `wildcard` function (see Section 5.2.3 [Wildcard Function], page 28) in that it communicates with the world outside of `make`.

The `shell` function performs the same function that backquotes ('`' ') perform in most shells: it does *command expansion*. This means that it takes an argument that is a shell command and returns the output of the command. The only processing `make` does on the result, before substituting it into the surrounding text, is to convert newlines to spaces.

The commands run by calls to the `shell` function are run when the function calls are expanded. In most cases, this is when the makefile is read in. The exception is that function calls in the commands of the rules are expanded when the commands are run, and this applies to `shell` function calls like all others.

Here are some examples of the use of the `shell` function:

```
contents := $(shell cat foo)
```

sets `contents` to the contents of the file '`foo`', with a space (rather than a newline) separating each line.

```
files := $(shell echo *.c)
```

sets `files` to the expansion of '`*.c`'. Unless `make` is using a very strange shell, this has the same result as '`$(wildcard *.c)`'.

10 How to Run `make`

A makefile that says how to recompile a program can be used in more than one way. The simplest use is to recompile every file that is out of date. This is what `make` will do if run with no arguments.

But you might want to update only some of the files; you might want to use a different compiler or different compiler options; you might want just to find out which files are out of date without changing them.

By specifying arguments when you run `make`, you can do any of these things or many others.

10.1 Arguments to Specify the Makefile

The way to specify the name of the makefile is with the '`-f`' option. For example, '`-f altmake`' says to use the file '`altmake`' as the makefile.

If you use the '`-f`' flag several times (each time with a following argument), all the specified files are used jointly as makefiles.

If you do not use the '`-f`' flag, the default is to try '`GNUmakefile`', '`makefile`', or '`Makefile`', in that order, and use the first of these three which exists. See Chapter 4 [Makefiles], page 19.

10.2 Arguments to Specify the Goals

The *goals* are the targets that `make` should strive ultimately to update. Other targets are updated as well if they appear as dependencies of goals, or dependencies of dependencies of goals, etc.

By default, the goal is the first target in the makefile (not counting targets that start with a period). Therefore, makefiles are usually written so that the first target is for compiling the entire program or programs they describe.

You can specify a different goal or goals with arguments to `make`. Use the name of the goal as an argument. If you specify several goals, `make` processes each of them in turn, in the order you name them.

Any target in the makefile may be specified as a goal (unless it starts with '`-`' or contains an '`=`'). Even targets not in the makefile may be specified, if `make` can find implicit rules that say how to make them.

One use of specifying a goal is if you want to compile only a part of the program, or only one of several programs. Specify as a goal each file that you wish to remake. For example, consider a directory containing several programs, with a makefile that starts like this:

```
.PHONY: all
all: size nm ld ar as
```

If you are working on the program `size`, you might want to say '`make size`' so that only the files of that program are recompiled.

Another use of specifying a goal is to make files that aren't normally made. For example, there may be a file of debugging output, or a version of the program that is compiled specially for testing, which has a rule in the makefile but isn't a dependency of the default goal.

Another use of specifying a goal is to run the commands associated with a phony target (see Section 5.4 [Phony Targets], page 32) or empty target (see Section 5.6 [Empty Targets], page 35). Many makefiles contain a phony target named '`clean`' which deletes everything except source files. Naturally, this is done only if you request it explicitly with '`make clean`'. Here is a list of typical phony and empty target names:

'`all`' Make all the top-level targets the makefile knows about.

'`clean`' Delete all files that are normally created by running `make`.

'`distclean`'
'`realclean`'
'`clobber`' Any of these three might be defined to delete everything that would not be part of a standard distribution. For example, this would delete configuration files or links that you would normally create as preparation for compilation, even if the makefile itself cannot create these files.

'`install`' Copy the executable file into a directory that users typically search for commands; copy any auxiliary files that the executable uses into the directories where it will look for them.

'`print`' Print listings of the source files that have changed.

'`tar`' Create a tar file of the source files.

‘`shar`’ Create a shell archive (shar file) of the source files.

‘`dist`’ Create a distribution file of the source files. This might be a tar file, or a shar file, or a compressed version of one of the above, or even more than one of the above.

10.3 Instead of Executing the Commands

The makefile tells `make` how to tell whether a target is up to date, and how to update each target. But updating the targets is not always what you want. Certain options specify other activities for `make`.

‘`-t`’ “Touch”. The activity is to mark the targets as up to date without actually changing them. In other words, `make` pretends to compile the targets but does not really change their contents.

‘`-n`’ “No-op”. The activity is to print what commands would be used to make the targets up to date, but not actually execute them.

‘`-q`’ “Question”. The activity is to find out silently whether the targets are up to date already; but execute no commands in either case. In other words, neither compilation nor output will occur.

‘`-W`’ “What if”. Each ‘`-W`’ flag is followed by a file name. The given files’ modification times are recorded by `make` as being the present time, although the actual modification times remain the same. When used in conjunction with the ‘`-n`’ flag, the ‘`-W`’ flag provides a way to see what would happen if you were to modify specific files.

With the ‘`-n`’ flag, `make` prints without execution the commands that it would normally execute.

With the ‘`-t`’ flag, `make` ignores the commands in the rules and uses (in effect) the command `touch` for each target that needs to be remade. The `touch` command is also printed, unless ‘`-s`’ or `.SILENT` is used. For speed, `make` does not actually invoke the program `touch`. It does the work directly.

With the ‘`-q`’ flag, `make` prints nothing and executes no commands, but the exit status code it returns is zero if and only if the targets to be considered are already up to date.

It is an error to use more than one of these three flags in the same invocation of `make`.

The ‘`-n`’, ‘`-t`’, and ‘`-q`’ options do not affect command lines that begin with ‘`+`’ characters or contain the strings ‘`$(MAKE)`’ or ‘`${MAKE}`’. Note that only the line containing the ‘`+`’ character or the strings '`$(MAKE)`' or '`${MAKE}`' is run regardless of these options. Other lines in the same rule are not run unless they too begin with '`+`' or contain '`$(MAKE)`' or '`${MAKE}`'.

The '`-W`' flag provides two features:

- If you also use the '`-n`' or '`-q`' flag, you can see what `make` would do if you were to modify some files.
- Without the '`-n`' or '`-q`' flag, when `make` is actually executing commands, the '`-W`' flag can direct `make` to act as if some files had been modified, without actually modifying the files.

Note that the options '`-p`' and '`-v`' allow you to obtain other information about `make` or about the makefiles in use. See Section 10.7 [Options], page 86.

10.4 Avoiding Recompilation of Some Files

Sometimes you may have changed a source file but you don't want to recompile all the files that depend on it. For example, suppose you add a macro or a declaration to a header file that many other files depend on. Being conservative, `make` assumes that any change in the header file requires recompilation of all dependent files, but you know that they don't need to be recompiled and you would rather not waste the time waiting for them to compile.

If you anticipate the problem before changing the header file, you can use the '`-t`' flag. This flag tells `make` not to run the commands in the rules, but rather to mark the target up to date by changing its last-modification date. You would follow this procedure:

1. Use the command '`make`' to recompile the source files that really need recompilation.
2. Make the changes in the header files.
3. Use the command '`make -t`' to mark all the object files as up to date. The next time you run `make`, the changes in the header files will not cause any recompilation.

If you have already changed the header file at a time when some files do need recompilation, it is too late to do this. Instead, you can use the '`-o` *file*' flag, which marks a specified file as "old" (see Section 10.7 [Options], page 86). This means that the file itself won't be remade, and nothing else will be remade on its account. Follow this procedure:

1. Recompile the source files that need compilation for reasons independent of the particular header file, with '`make -o` *headerfile*'. If several header files are involved, use a separate '`-o`'

option for each header file.

2. Touch all the object files with '`make -t`'.

10.5 Overriding Variables

An argument that contains '`=`' specifies the value of a variable: '*v*`=`*x*' sets the value of the variable *v* to *x*. If you specify a value in this way, all ordinary assignments of the same variable in the makefile are ignored; we say they have been *overridden* by the command line argument.

The most common way to use this facility is to pass extra flags to compilers. For example, in a properly written makefile, the variable `CFLAGS` is included in each command that runs the C compiler, so a file '`foo.c`' would be compiled something like this:

```
cc -c $(CFLAGS) foo.c
```

Thus, whatever value you set for `CFLAGS` affects each compilation that occurs. The makefile probably specifies the usual value for `CFLAGS`, like this:

```
CFLAGS=-g
```

Each time you run `make`, you can override this value if you wish. For example, if you say '`make CFLAGS='-g -O'`', each C compilation will be done with '`cc -c -g -O`'. (This illustrates how you can enclose spaces and other special characters in the value of a variable when you override it.)

The variable `CFLAGS` is only one of many standard variables that exist just so that you can change them this way. See Section 11.3 [Implicit Variables], page 93, for a complete list.

You can also program the makefile to look at additional variables of your own, giving the user the ability to control other aspects of how the makefile works by changing the variables.

When you override a variable with a command argument, you can define either a recursively-expanded variable or a simply-expanded variable. The examples shown above make a recursively-expanded variable; to make a simply-expanded variable, write '`:=`' instead of '`=`'. But, unless you want to include a variable reference or function call in the *value* that you specify, it makes no difference which kind of variable you create.

There is one way that the makefile can change a variable that you have overridden. This is to use the `override` directive, which is a line that looks like this: '`override` *variable* `=` *value*'. See Section 7.6 [Override Directive], page 63.

10.6 Testing the Compilation of a Program

Normally, when an error happens in executing a shell command, `make` gives up immediately, returning a nonzero status. No further commands are executed for any target. The error implies that the goal cannot be correctly remade, and `make` reports this as soon as it knows.

When you are compiling a program that you have just changed, this is not what you want. Instead, you would rather that `make` try compiling every file that can be tried, to show you as many compilation errors as possible.

On these occasions, you should use the '`-k`' flag. This tells `make` to continue to consider the other dependencies of the pending targets, remaking them if necessary, before it gives up and returns nonzero status. For example, after an error in compiling one object file, '`make -k`' will continue compiling other object files even though it already knows that linking them will be impossible. In addition to continuing after failed shell commands, '`make -k`' will continue as much as possible after discovering that it doesn't know how to make a target or dependency file. This will always cause an error message, but without '`-k`', it is a fatal error. See Section 10.7 [Options], page 86.

The usual behavior of `make` assumes that your purpose is to get the goals up to date; once `make` learns that this is impossible, it might as well report the failure immediately. The '`-k`' flag says that the real purpose is to test as much as possible of the changes made in the program, perhaps to find several independent problems so that you can correct them all before the next attempt to compile. This is why Emacs's `M-x compile` command passes the '`-k`' flag by default.

10.7 Summary of Options

Here is a table of all the options `make` understands:

'`-b`'
'`-m`'
: These options are ignored for compatibility with other versions of `make`.

'`-C` *dir*'
: Change to directory *dir* before reading the makefiles. If multiple '`-C`' options are specified, each is interpreted relative to the previous one: '`-C / -C etc`' is equivalent

to '-C /etc'. This is typically used with recursive invocations of make (see Section 6.6 [Recursion], page 47).

'-d' Print debugging information in addition to normal processing. The debugging information says which files are being considered for remaking, which file-times are being compared and with what results, which files actually need to be remade, which implicit rules are considered and which are applied—everything interesting about how make decides what to do.

'-e' Give variables taken from the environment precedence over variables from makefiles. See Section 7.8 [Environment], page 64.

'-f *file*' Use file *file* as a makefile. See Chapter 4 [Makefiles], page 19.

'-i' Ignore all errors in commands executed to remake files. See Section 6.4 [Errors], page 46.

'-I *dir*' Specifies a directory *dir* to search for included makefiles. See Section 4.3 [Include], page 20. If several '-I' options are used to specify several directories, the directories are searched in the order specified. Unlike the arguments to other flags of make, directories given with '-I' flags may come directly after the flag: '-I*dir*' is allowed, as well as '-I *dir*'. This syntax is allowed for compatibility with the C preprocessor's '-I' flag.

'-j *jobs*' Specifies the number of jobs (commands) to run simultaneously. If there is more than one '-j' option, the last one is effective. See Section 6.2 [Execution], page 44, for more information on how commands are run.

'-k' Continue as much as possible after an error. While the target that failed, and those that depend on it, cannot be remade, the other dependencies of these targets can be processed all the same. See Section 10.6 [Testing], page 86.

'-l *load*'

'-l' Specifies that no new jobs (commands) should be started if there are others jobs running and the load average is at least *load* (a floating-point number). With no argument, removes a previous load limit. See Section 6.3 [Parallel], page 44.

'-n' Print the commands that would be executed, but do not execute them. See Section 10.3 [Instead of Execution], page 83.

'-o *file*' Do not remake the file *file* even if it is older than its dependencies, and do not remake anything on account of changes in *file*. Essentially the file is treated as very old and its rules are ignored. See Section 10.4 [Avoiding Compilation], page 84.

'-p' Print the data base (rules and variable values) that results from reading the makefiles; then execute as usual or as otherwise specified. This also prints the version information given by the '-v' switch (see below). To print the data base without trying to remake any files, use 'make -p -f /dev/null'.

'-q' "Question mode". Do not run any commands, or print anything; just return an exit status that is zero if the specified targets are already up to date, nonzero otherwise. See Section 10.3 [Instead of Execution], page 83.

'-r' Eliminate use of the built-in implicit rules (see Chapter 11 [Implicit], page 89). Also clear out the default list of suffixes for suffix rules (see Section 11.7 [Suffix Rules], page 104).

'-s' Silent operation; do not print the commands as they are executed. See Section 6.1 [Echoing], page 43.

'-S' Cancel the effect of the '-k' option. This is never necessary except in a recursive `make` where '-k' might be inherited from the top-level `make` via `MAKEFLAGS` (see Section 6.6 [Recursion], page 47) or if you set '-k' in `MAKEFLAGS` in your environment.

'-t' Touch files (mark them up to date without really changing them) instead of running their commands. This is used to pretend that the commands were done, in order to fool future invocations of `make`. See Section 10.3 [Instead of Execution], page 83.

'-v' Print the version of the `make` program plus a copyright, a list of authors and a notice that there is no warranty. After this information is printed, processing continues normally. To get this information without doing anything else, use '`make -v -f /dev/null`'.

'-w' Print a message containing the working directory both before and after executing the makefile. This may be useful for tracking down errors from complicated nests of recursive `make` commands. See Section 6.6 [Recursion], page 47.

'-W *file*' Pretend that the target *file* has just been modified. When used with the '-n' flag, this shows you what would happen if you were to modify that file. Without '-n', it is almost the same as running a `touch` command on the given file before running `make`, except that the modification time is changed only in the imagination of `make`.

11 Using Implicit Rules

Certain standard ways of remaking target files are used very often. For example, one customary way to make an object file is from a C source file using the C compiler, `cc`.

Implicit rules tell `make` how to use customary techniques so that you don't have to specify them in detail when you want to use them. For example, there is an implicit rule for C compilation. Implicit rules work based on file names. For example, C compilation typically takes a '`.c`' file and makes a '`.o`' file. So `make` applies the implicit rule for C compilation when it sees this combination of file-name endings.

A chain of implicit rules can apply in sequence; for example, `make` will remake a '`.o`' file from a '`.y`' file by way of a '`.c`' file. See Section 11.4 [Chained Rules], page 95.

The built-in implicit rules use several variables in their commands so that, by changing the values of the variables, you can change the way the implicit rule works. For example, the variable `CFLAGS` controls the flags given to the C compiler by the implicit rule for C compilation. See Section 11.3 [Implicit Variables], page 93.

You can define your own implicit rules by writing *pattern rules*. See Section 11.5 [Pattern Rules], page 96.

11.1 Using Implicit Rules

To allow `make` to find a customary method for updating a target file, all you have to do is refrain from specifying commands yourself. Either write a rule with no command lines, or don't write a rule at all. Then `make` will figure out which implicit rule to use based on which kind of source file exists.

For example, suppose the makefile looks like this:

```
foo : foo.o bar.o
        cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

Because you mention '`foo.o`' but do not give a rule for it, `make` will automatically look for an implicit rule that tells how to update it. This happens whether or not the file '`foo.o`' currently exists.

If an implicit rule is found, it supplies both commands and one or more dependencies (the source files). You would want to write a rule for '`foo.o`' with no command lines if you need to specify additional dependencies, such as header files, that the implicit rule cannot supply.

Each implicit rule has a target pattern and dependency patterns. There may be many implicit rules with the same target pattern. For example, numerous rules make '`.o`' files: one, from a '`.c`' file with the C compiler; another, from a '`.p`' file with the Pascal compiler; and so on. The rule that actually applies is the one whose dependencies exist or can be made. So, if you have a file '`foo.c`', `make` will run the C compiler; otherwise, if you have a file '`foo.p`', `make` will run the Pascal compiler; and so on.

Of course, when you write the makefile, you know which implicit rule you want `make` to use, and you know it will choose that one because you know which possible dependency files are supposed to exist. See Section 11.2 [Catalogue of Rules], page 90, for a catalogue of all the predefined implicit rules.

Above, we said an implicit rule applies if the required dependencies "exist or can be made". A file "can be made" if it is mentioned explicitly in the makefile as a target or a dependency, or if an implicit rule can be recursively found for how to make it. When an implicit dependency is the result of another implicit rule, we say that *chaining* is occurring. See Section 11.4 [Chained Rules], page 95.

In general, `make` searches for an implicit rule for each target, and for each double-colon rule, that has no commands. A file that is mentioned only as a dependency is considered a target whose rule specifies nothing, so implicit rule search happens for it. See Section 11.8 [Search Algorithm], page 105, for the details of how the search is done.

If you don't want an implicit rule to be used for a target that has no commands, you can give that target empty commands by writing a semicolon. See Section 6.8 [Empty Commands], page 52.

11.2 Catalogue of Implicit Rules

Here is a catalogue of predefined implicit rules which are always available unless the makefile explicitly overrides or cancels them. See Section 11.5.6 [Canceling Rules], page 102, for information on canceling or overriding an implicit rule. The '`-r`' option cancels all predefined rules.

Not all of these rules will always be defined, even when the '`-r`' option is not given. Many of the predefined implicit rules are implemented in `make` as suffix rules, so which ones will be defined depends on the *suffix list* (the list of dependencies of the special target `.SUFFIXES`). See Section 11.7 [Suffix Rules], page 104. The default suffix list is: '`.out`', '`.a`', '`.o`', '`.c`', '`.cc`', '`.C`', '`.p`', '`.f`', '`.F`', '`.r`', '`.e`', '`.y`', '`.ye`', '`.yr`', '`.l`', '`.s`', '`.S`', '`.h`', '`.info`', '`.dvi`', '`.tex`', '`.texinfo`', '`.cweb`', '`.web`', '`.sh`', '`.elc`', '`.el`'. All of the implicit rules described below whose dependencies have one of these suffixes are actually suffix rules. If you modify the suffix list, the only predefined suffix rules in effect will be those named by one or two of the suffixes that are on the list you specify; rules whose suffixes fail to be on the list are disabled.

Compiling C programs
: '*n*`.o`' will be made automatically from '*n*`.c`' with a command of the form '`$(CC) -c $(CPPFLAGS) $(CFLAGS)`'.

Compiling C++ programs
: '*n*`.o`' will be made automatically from '*n*`.cc`' or '*n*`.C`' with a command of the form '`$(C++) -c $(CPPFLAGS) $(C++FLAGS)`'. We encourage you to use the suffix '`.cc`' for C++ source files instead of '`.C`'.

Compiling Pascal programs
: '*n*`.o`' will be made automatically from '*n*`.p`' with the command '`$(PC) -c $(PFLAGS)`'.

Compiling Fortran and Ratfor programs
: '*n*`.o`' will be made automatically from '*n*`.r`', '*n*`.F`' or '*n*`.f`' by running the Fortran compiler. The precise command used is as follows:

  '`.f`' '`$(FC) -c $(FFLAGS)`'.

  '`.F`' '`$(FC) -c $(FFLAGS) $(CPPFLAGS)`'.

  '`.r`' '`$(FC) -c $(FFLAGS) $(RFLAGS)`'.

Preprocessing Fortran and Ratfor programs
: '*n*`.f`' will be made automatically from '*n*`.r`' or '*n*`.F`'. This rule runs just the preprocessor to convert a Ratfor or preprocessable Fortran program into a strict Fortran program. The precise command used is as follows:

  '`.F`' '`$(FC) -F $(CPPFLAGS) $(FFLAGS)`'.

  '`.r`' '`$(FC) -F $(FFLAGS) $(RFLAGS)`'.

Compiling Modula-2 programs
: '*n*`.sym`' will be made from '*n*`.def`' with a command of the form '`$(M2C) $(M2FLAGS) $(DEFFLAGS)`'. '*n*`.o`' will be made from '*n*`.mod`' with a command of the form '`$(M2C) $(M2FLAGS) $(MODFLAGS)`'.

Assembling and preprocessing assembler programs
: '*n*`.o`' will be made automatically from '*n*`.s`' by running the assembler `as`. The precise command used is '`$(AS) $(ASFLAGS)`'.

  '*n*`.s`' will be made automatically from '*n*`.S`' by running the C preprocessor `cpp`. The precise command used is '`$(CPP) $(CPPFLAGS)`'.

Linking a single object file

'*n*' will be made automatically from '*n*.o' by running the linker `ld` via the C compiler. The precise command used is '`$(CC) $(LDFLAGS)` *n*.o `$(LOADLIBES)`'.

This rule does the right thing for a simple program with only one source file. It will also do the right thing if there are multiple object files (presumably coming from various other source files), the first of which has a name matching that of the executable file. Thus,

```
x: y.o z.o
```

when '`x.c`', '`y.c`' and '`z.c`' all exist will execute:

```
cc -c x.c -o x.o
cc -c y.c -o y.o
cc -c z.c -o z.o
cc x.o y.o z.o -o x
rm -f x.o
rm -f y.o
rm -f z.o
```

In more complicated cases, such as when there is no object file whose name derives from the executable file name, you must write an explicit command for linking.

Each kind of file automatically made into `.o` object files will be automatically linked by using the compiler ('`$(CC)`', '`$(FC)`' or '`$(PC)`'; the C compiler '`$(CC)`' is used to assemble `.s` files) without the '`-c`' option. This could be done by using the `.o` object files as intermediates, but it is faster to do the compiling and linking in one step, so that's how it's done.

Yacc for C programs

'*n*.c' will be made automatically from '*n*.y' by running Yacc with the command '`$(YACC) $(YFLAGS)`'.

Lex for C programs

'*n*.c' will be made automatically from '*n*.l' by by running Lex. The actual command is '`$(LEX) $(LFLAGS)`'.

Lex for Ratfor programs

'*n*.r' will be made automatically from '*n*.l' by by running Lex. The actual command is '`$(LEX) $(LFLAGS)`'.

The convention of using the same suffix '`.l`' for all Lex files regardless of whether they produce C code or Ratfor code makes it impossible for `make` to determine automatically which of the two languages you are using in any particular case. If `make` is called upon to remake an object file from a '`.l`' file, it must guess which compiler to use. It will guess the C compiler, because that is more common. If you are using Ratfor, make sure `make` knows this by mentioning '*n*.r' in the makefile. Or, if you are using Ratfor exclusively, with no C files, remove `.c` from the list of implicit rule suffixes with:

```
.SUFFIXES:
```

```
.SUFFIXES: .r .f .l ...
```

Making Lint Libraries from C, Yacc, or Lex programs
: '*n*.ln' will be made from '*n*.c' with a command of the form '$(LINT) $(LINTFLAGS) $(CPPFLAGS) -i'. The same command will be used on the C code produced from '*n*.y' or '*n*.l'.

TeX and Web
: '*n*.dvi' will be made from '*n*.tex' with the command '$(TEX)'. '*n*.tex' will be made from '*n*.web' with '$(WEAVE)', or from '*n*.cweb' with '$(CWEAVE)'. '*n*.p' will be made from '*n*.web' with '$(TANGLE)' and '*n*.c' will be made from '*n*.cweb' with '$(CTANGLE)'.

Texinfo and Info
: '*n*.dvi' will be made from '*n*.texinfo' using the '$(TEX)' and '$(TEXINDEX)' commands. The actual command sequence contains many shell conditionals to avoid unnecessarily running TeX twice and to create the proper sorted index files. '*n*.info' will be made from '*n*.texinfo' with the command '$(MAKEINFO)'.

RCS
: Any file '*n*' will be extracted if necessary from an RCS file named either '*n*,v' or 'RCS/*n*,v'. The precise command used is '$(CO) $(COFLAGS)'. '*n*' will not be extracted from RCS if it already exists, even if the RCS file is newer.

SCCS
: Any file '*n*' will be extracted if necessary from an SCCS file named either 's.*n*' or 'SCCS/s.*n*'. The precise command used is '$(GET) $(GFLAGS)'.

  For the benefit of SCCS, a file '*n*' will be copied from '*n*.sh' and made executable (by everyone). This is for shell scripts that are checked into SCCS. Since RCS preserves the execution permission of a file, you don't need to use this feature with RCS.

  We recommend that you avoid the use of SCCS. RCS is widely held to be superior, and is also free. By choosing free software in place of comparable (or inferior) proprietary software, you support the free software movement.

11.3 Variables Used by Implicit Rules

The commands in built-in implicit rules make liberal use of certain predefined variables. You can alter these variables, either in the makefile or with arguments to `make`, to alter how the implicit rules work without redefining the rules themselves.

For example, the command used to compile a C source file actually says '$(CC) -c $(CFLAGS) $(CPPFLAGS)'. The default values of the variables used are 'cc' and nothing, resulting in the command 'cc -c'. By redefining '$(CC)' to 'ncc', you could cause 'ncc' to be used for all C compilations performed by the implicit rule. By redefining '$(CFLAGS)' to be '-g', you could pass the '-g' option to each compilation. *All* implicit rules that do C compilation use '$(CC)' to get the program name for the compiler and *all* include '`$(CFLAGS)`' among the arguments given to the compiler.

The variables used in implicit rules fall into two classes: those that are names of programs (like `CC`) and those that contain arguments for the programs (like `CFLAGS`). (The "name of a program" may also contain some command arguments, but it must start with an actual executable program name.) If a variable value contains more than one argument, separate them with spaces.

Here is a table of variables used as names of programs:

`AR` Archive-maintaing program; default '`ar`'.

`AS` Program for doing assembly; default '`as`'.

`CC` Program for compiling C programs; default '`cc`'.

`C++` Program for compiling C++ programs; default '`g++`'.

`CO` Program for extracting a file from RCS; default '`co`'.

`CPP` Program for running the C preprocessor, with results to standard output; default '`$(CC) -E`'.

`FC` Program for compiling or preprocessing Fortran, Ratfor, and EFL programs; default '`f77`'.

`GET` Program for extracting a file from SCCS; default '`get`'.

`LEX` Program to use to turn Lex grammars into C programs or Ratfor programs; default '`lex`'.

`PC` Program for compiling Pascal programs; default '`pc`'.

`FC`
`EC`
`RC` Programs for compiling Fortran, EFL, and Ratfor programs, respectively; these all default to '`f77`'.

`YACC` Program to use to turn Yacc grammars into C programs; default '`yacc`'.

`YACCR` Program to use to turn Yacc grammars into Ratfor programs; default '`yacc -r`'.

`YACCE` Program to use to turn Yacc grammars into EFL programs; default '`yacc -e`'.

`MAKEINFO` Program to make Info files from Texinfo source; default '`makeinfo`'.

`TEX` Program to make TeX DVI files from TeX or Texinfo source; default '`tex`'.

`TEXINDEX` The `texindex` program distributed with Emacs. This is used in the process to make TeX DVI files from Texinfo source.

`WEAVE` Program to translate Web into TeX; default '`weave`'.

`CWEAVE` Program to translate C Web into TeX; default '`cweave`'.

`TANGLE` Program to translate Web into Pascal; default '`tangle`'.

`CTANGLE` Program to translate C Web into C; default '`ctangle`'.

`RM` Command to remove a file; default '`rm -f`'.

Here is a table of variables whose values are additional arguments for the programs above. The default values for all of these is the empty string, unless otherwise noted.

`ARFLAGS` Flags to give the archive- maintaining program; default '`rv`'.

`ASFLAGS` Extra flags to give to the assembler (when explicitly invoked on a '`.s`' file).

`CFLAGS` Extra flags to give to the C compiler.

`C++FLAGS` Extra flags to give to the C++ compiler.

`COFLAGS` Extra flags to give to the RCS `co` program.

`CPPFLAGS` Extra flags to give to the C preprocessor and programs that use it (the C and Fortran compilers).

`EFLAGS` Extra flags to give to the Fortran compiler for EFL programs.

`FFLAGS` Extra flags to give to the Fortran compiler.

`GFLAGS` Extra flags to give to the SCCS `get` program.

`LDFLAGS` Extra flags to give to compilers when they are supposed to invoke the linker, '`ld`'.

`LFLAGS` Extra flags to give to Lex.

`PFLAGS` Extra flags to give to the Pascal compiler.

`RFLAGS` Extra flags to give to the Fortran compiler for Ratfor programs.

`YFLAGS` Extra flags to give to Yacc.

11.4 Chains of Implicit Rules

Sometimes a file can be made by a sequence of implicit rules. For example, a file '*n*`.o`' could be made from '*n*`.y`' by running first Yacc and then `cc`. Such a sequence is called a *chain*.

If the file '*n*`.c`' exists, or is mentioned in the makefile, no special searching is required: `make` finds that the object file can be made by C compilation from '*n*`.c`'; later on, when considering how to make '*n*`.c`', the rule for running Yacc will be used. Ultimately both '*n*`.c`' and '*n*`.o`' are updated.

However, even if '*n*`.c`' does not exist and is not mentioned, `make` knows how to envision it as the missing link between '*n*`.o`' and '*n*`.y`'! In this case, '*n*`.c`' is called an *intermediate file*. Once `make` has decided to use the intermediate file, it is entered in the data base as if it had been mentioned in the makefile, along with the implicit rule that says how to create it.

Intermediate files are remade using their rules just like all other files. The difference is that the intermediate file is deleted when `make` is finished. Therefore, the intermediate file which did not exist before `make` also does not exist after `make`. The deletion is reported to you by printing a '`rm -f`' command that shows what `make` is doing. (You can optionally define an implicit rule so as to preserve certain intermediate files. You can also list the target pattern of an implicit rule (such as '`%.o`') as a dependency file of the special target `.PRECIOUS` to preserve intermediate files whose target patterns match that file's name.)

A chain can involve more than two implicit rules. For example, it is possible to make a file '`foo`' from '`RCS/foo.y,v`' by running RCS, Yacc and `cc`. Then both '`foo.y`' and '`foo.c`' are intermediate files that are deleted at the end.

No single implicit rule can appear more than once in a chain. This means that `make` will not even consider such a ridiculous thing as making '`foo`' from '`foo.o.o`' by running the linker twice. This constraint has the added benefit of preventing any infinite loop in the search for an implicit rule chain.

There are some special implicit rules to optimize certain cases that would otherwise by handled by rule chains. For example, making '`foo`' from '`foo.c`' could be handled by compiling and linking with separate chained rules, using '`foo.o`' as an intermediate file. But what actually happens is that a special rule for this case does the compilation and linking with a single `cc` command. The optimized rule is used in preference to the step-by-step chain because it comes earlier in the ordering of rules.

11.5 Defining and Redefining Pattern Rules

You define an implicit rule by writing a *pattern rule*. A pattern rule looks like an ordinary rule, except that its target contains the character '`%`' (exactly one of them). The target is considered a pattern for matching file names; the '`%`' can match any nonempty substring, while other characters match only themselves. The dependencies likewise use '`%`' to show how their names relate to the target name.

Thus, a pattern rule '`%.o : %.c`' says how to make any file '*stem*`.o`' from another file '*stem*`.c`'.

11.5.1 Introduction to Pattern Rules

You define an implicit rule by writing a *pattern rule*. A pattern rule looks like an ordinary rule, except that its target contains the character '%' (exactly one of them). The target is considered a pattern for matching file names; the '%' can match any nonempty substring, while other characters match only themselves.

For example, '`%.c`' as a pattern matches any file name that ends in '`.c`'. '`s.%.c`' as a pattern matches any file name that starts with '`s.`', ends in '`.c`' and is at least five characters long. (There must be at least one character to match the '%'.) The substring that the '%' matches is called the *stem*.

'%' in a dependency of a pattern rule stands for the same stem that was matched by the '%' in the target. In order for the pattern rule to apply, its target pattern must match the file name under consideration, and its dependency patterns must name files that exist or can be made. These files become dependencies of the target.

Thus, a rule of the form

```
%.o : %.c
```

would specify how to make any file '*n*`.o`', with another file '*n*`.c`' as its dependency, provided that the other file exists or can be made.

There may also be dependencies that do not use '%'; such a dependency attaches to every file made by this pattern rule. These unvarying dependencies are useful occasionally.

It is allowed for a pattern rule to have no dependencies that contain '%' or to have no dependencies at all. This is effectively a general wildcard. It provides a way to make any file that matches the target pattern.

Pattern rules may have more than one target. Unlike normal rules, this does not act as many different rules with the same dependencies and commands. If a pattern rule has multiple targets, `make` knows that the rule's commands are responsible for making all of the targets. The commands are executed only once to make all of the targets. When searching for a pattern rule to match a target, the target patterns of a rule other than the one that matches the target in need of a rule are incidental: `make` worries only about giving commands and dependencies to the file presently in question. However, when this file's commands are run, the other targets are marked as having been updated themselves.

The order in which pattern rules appear in the makefile is important because the rules are considered in that order. Of equally applicable rules, the first one found is used. The rules you write take precedence over those that are built in. Note, however, that a rule whose dependencies actually exist or are mentioned always takes priority over a rule with dependencies that must be made by chaining other implicit rules.

11.5.2 Pattern Rule Examples

Here are some examples of pattern rules actually predefined in `make`. First, the rule that compiles '.c' files into '.o' files:

```
%.o : %.c
        $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

defines a rule that can make any file '*x*.o' from '*x*.c'. The command uses the automatic variables '$@' and '$<' to substitute the names of the target file and the source file in each case where the rule applies (see Section 11.5.3 [Automatic], page 99).

Here is a second built-in rule:

```
% :: RCS/%,v
        $(CO) $(COFLAGS) $<
```

defines a rule that can make any file '*x*' whatever from a corresponding file '*x*,v' in the subdirectory 'RCS'. Since the target is '%', this rule will apply to any file whatever, provided the appropriate dependency file exists. The double colon makes the rule *terminal*, which means that its dependency may not be an intermediate file (see Section 11.5.5 [Match-Anything Rules], page 101).

This pattern rule has two targets:

```
%.tab.c %.tab.h: %.y
        bison -d $<
```

This tells `make` that the command 'bison -d *x*.y' will make both '*x*.tab.c' and '*x*.tab.h'. If the file 'foo' depends on the files 'parse.tab.o' and 'scan.o' and 'scan.o' depends on 'parse.tab.h', when 'parse.y' is changed, the command 'bison -d parse.y' will be executed only once, and the dependencies of both 'parse.tab.o' and 'scan.o' will be satisfied. (Presumably, 'parse.tab.o' will be recompiled from 'parse.tab.c' and 'scan.o' from 'scan.c', and 'foo' will be linked from 'parse.tab.o', 'scan.o', and its other dependencies, and it will execute happily ever after.)

11.5.3 Automatic Variables

Suppose you are writing a pattern rule to compile a '.c' file into a '.o' file: how do you write the 'cc' command so that it operates on the right source file name? You can't write the name in the command, because the name is different each time the implicit rule is applied.

What you do is use a special feature of `make`, the *automatic variables*. These variables have values computed afresh for each rule that is executed, based on the target and dependencies of the rule. In this example, you would use '$@' for the object file name and '$<' for the source file name.

Here is a table of automatic variables:

`$@`
: The file name of the target of the rule. If the target is an archive member, then '$@' is the name of the archive file.

`$%`
: The target member name, when the target is an archive member. For example, if the target is 'foo.a(bar.o)' then '$%' is 'bar.o' and '$@' is 'foo.a'. '$%' is empty when the target is not an archive member.

`$<`
: The name of the first dependency.

`$?`
: The names of all the dependencies that are newer than the target, with spaces between them.

`$^`
: The names of all the dependencies, with spaces between them.

`$*`
: The stem with which an implicit rule matches (see Section 11.5.4 [Pattern Match], page 100). If the target is 'dir/a.foo.b' and the target pattern is 'a.%.b' then the stem is 'dir/foo'. The stem is useful for constructing names of related files.

  In an explicit rule, there is no stem; so '$*' cannot be determined in that way. Instead, if the target name ends with a recognized suffix (see Section 11.7 [Suffix Rules], page 104), '$*' is set to the target name minus the suffix. For example, if the target name is 'foo.c', then '$*' is set to 'foo', since '.c' is a suffix.

  If the target name in an explicit rule does not end with a recognized suffix, '$*' is set to the empty string for that rule.

'$?' is useful even in explicit rules when you wish to operate on only the dependencies that have changed. For example, suppose that an archive named 'lib' is supposed to contain copies of several object files. This rule copies just the changed object files into the archive:

```
lib: foo.o bar.o lose.o win.o
        ar r lib $?
```

Of the variables listed above, four have values that are single file names, and two have values that are lists of file names. These six have variants that get just the file's directory name or just the file name within the directory. The variant variables' names are formed by appending '`D`' or '`F`', respectively. These variants are semi-obsolete in GNU `make` since the functions `dir` and `notdir` can be used to get an equivalent effect (see Section 9.3 [Filename Functions], page 74). Here is a table of the variants:

'`$(@D)`'
: The directory part of the file name of the target. If the value of '`$@`' is '`dir/foo.o`' then '`$(@D)`' is '`dir/`'. This value is '`./`' if '`$@`' does not contain a slash. '`$(@D)`' is equivalent to '`$(dir $@)`'.

'`$(@F)`'
: The file-within-directory part of the file name of the target. If the value of '`$@`' is '`dir/foo.o`' then '`$(@F)`' is '`foo.o`'. '`$(@F)`' is equivalent to '`$(notdir $@)`'.

'`$(*D)`'
'`$(*F)`'
: The directory part and the file-within-directory part of the stem; '`dir/`' and '`foo`' in this example.

'`$(%D)`'
'`$(%F)`'
: The directory part and the file-within-directory part of the target archive member name. This makes sense only for archive member targets of the form '*archive*`(`*member*`)`' and useful only when *member* may contain a directory name. (See Section 12.1 [Archive Members], page 109.)

'`$(<D)`'
'`$(<F)`'
: The directory part and the file-within-directory part of the first dependency.

'`$(^D)`'
'`$(^F)`'
: Lists of the directory parts and the file-within-directory parts of all dependencies.

'`$(?D)`'
'`$(?F)`'
: Lists of the directory parts and the file-within-directory parts of all dependencies that are out of date with respect to the target.

Note that we use a special stylistic convention when we talk about these automatic variables; we write "the value of '`$<`'", rather than "the variable `<`" as we would write for ordinary variables such as `objects` and `CFLAGS`. We think this convention looks more natural in this special case. Please don't assume it has a deep significance; '`$<`' refers to the variable named `<` just as '`$(CFLAGS)`' refers to the variable named `CFLAGS`.

11.5.4 How Patterns Match

A target pattern is composed of a '`%`' between a prefix and a suffix, either or both of which may be empty. The pattern matches a file name only if the file name starts with the prefix and ends with the suffix, without overlap. The text between the prefix and the suffix is called the *stem*. Thus, when the pattern '`%.o`' matches the file name '`test.o`', the stem is '`test`'. The pattern rule dependencies are turned into actual file names by substituting the stem for the character '`%`'. Thus, if in the same example one of the dependencies is written as '`%.c`', it expands to '`test.c`'.

When the target pattern does not contain a slash (and usually it does not), directory names in the file names are removed from the file name before it is compared with the target prefix and suffix. The directory names, along with the slash that ends them, are added back to the stem. Thus, '`e%t`' does match the file name '`src/eat`', with '`src/a`' as the stem. When dependencies are turned into file names, the directories from the stem are added at the front, while the rest of the stem is substituted for the '`%`'. The stem '`src/a`' with a dependency pattern '`c%r`' gives the file name '`src/car`'.

11.5.5 Match-Anything Pattern Rules

When a pattern rule's target is just '`%`', it matches any filename whatever. We call these rules *match-anything* rules. They are very useful, but it can take a lot of time for `make` to think about them, because it must consider every such rule for each file name listed either as a target or as a dependency.

Suppose the makefile mentions '`foo.c`'. For this target, `make` would have to consider making it by linking an object file '`foo.c.o`', or by C compilation-and-linking in one step from '`foo.c.c`', or by Pascal compilation-and-linking from '`foo.c.p`', and many other possibilities.

We know these possibilities are ridiculous since '`foo.c`' is a C source file, not an executable. If `make` did consider these possibilities, it would ultimately reject them, because files such as '`foo.c.o`', '`foo.c.p`', etc. would not exist. But these possibilities are so numerous that `make` would run very slowly if it had to consider them.

To gain speed, we have put various constraints on the way `make` considers match-anything rules. There are two different constraints that can be applied, and each time you define a match-anything rule you must choose one or the other for that rule.

One choice is to mark the match-anything rule as *terminal* by defining it with a double colon. When a rule is terminal, it does not apply unless its dependencies actually exist. Dependencies that could be made with other implicit rules are not good enough. In other words, no further chaining is allowed beyond a terminal rule.

For example, the built-in implicit rules for extracting sources from RCS and SCCS files are terminal; as a result, if the file '`foo.c,v`' does not exist, `make` will not even consider trying to make it as an intermediate file from '`foo.c,v.o`' or from '`RCS/SCCS/s.foo.c,v`'. RCS and SCCS files are generally ultimate source files, which should not be remade from any other files; therefore, `make` can save time by not looking for ways to remake them.

If you do not mark the match-anything rule as terminal, then it is nonterminal. A nonterminal match-anything rule cannot apply to a file name that indicates a specific type of data. A file name indicates a specific type of data if some non-match-anything implicit rule target matches it.

For example, the file name '`foo.c`' matches the target for the pattern rule '`%.c : %.y`' (the rule to run Yacc). Regardless of whether this rule is actually applicable (which happens only if there is a file '`foo.y`'), the fact that its target matches is enough to prevent consideration of any nonterminal match-anything rules for the file '`foo.c`'. Thus, `make` will not even consider trying to make '`foo.c`' as an executable file from '`foo.c.o`', '`foo.c.c`', '`foo.c.p`', etc.

The motivation for this constraint is that nonterminal match-anything rules are used for making files containing specific types of data (such as executable files) and a file name with a recognized suffix indicates some other specific type of data (such as a C source file).

Special built-in dummy pattern rules are provided solely to recognize certain file names so that nonterminal match-anything rules won't be considered. These dummy rules have no dependencies and no commands, and they are ignored for all other purposes. For example, the built-in implicit rule

```
%.p :
```

exists to make sure that Pascal source files such as '`foo.p`' match a specific target pattern and thereby prevent time from being wasted looking for '`foo.p.o`' or '`foo.p.c`'.

Dummy pattern rules such as the one for '`%.p`' are made for every suffix listed as valid for use in suffix rules. See Section 11.7 [Suffix Rules], page 104.

11.5.6 Canceling Implicit Rules

You can override a built-in implicit rule by defining a new pattern rule with the same target and dependencies, but different commands. When the new rule is defined, the built-in one is replaced.

The new rule's position in the sequence of implicit rules is determined by where you write the new rule.

You can cancel a built-in implicit rule by defining a pattern rule with the same target and dependencies, but no commands. For example, the following would cancel the rule that runs the assembler:

```
%.o : %.s
```

11.6 Defining Last-Resort Default Rules

You can define a last-resort implicit rule by writing a rule for the target `.DEFAULT`. Such a rule's commands are used for all targets and dependencies that have no commands of their own and for which no other implicit rule applies. Naturally, there is no `.DEFAULT` rule unless you write one.

For example, when testing a makefile, you might not care if the source files contain real data, only that they exist. Then you might do this:

```
.DEFAULT:
        touch $@
```

to cause all the source files needed (as dependencies) to be created automatically.

If you give `.DEFAULT` with no commands or dependencies:

```
.DEFAULT:
```

the commands previously stored for `.DEFAULT` are cleared. Then `make` acts as if you had never defined `.DEFAULT` at all.

If you want a target not to get the commands from `.DEFAULT`, but nor do you want any commands to be run for the target, you can give it empty commands. See Section 6.8 [Empty Commands], page 52.

11.7 Old-Fashioned Suffix Rules

Suffix rules are the old-fashioned way of defining implicit rules for `make`. Suffix rules are obsolete because pattern rules are more general and clearer. They are supported in GNU `make` for compatibility with old makefiles. They come in two kinds: *double-suffix* and *single-suffix*.

A double-suffix rule is defined by a pair of suffixes: the target suffix and the source suffix. It matches any file whose name ends with the target suffix. The corresponding implicit dependency is to the file name made by replacing the target suffix with the source suffix. A two-suffix rule whose target and source suffixes are '`.o`' and '`.c`' is equivalent to the pattern rule '`%.o : %.c`'.

A single-suffix rule is defined by a single suffix, which is the source suffix. It matches any file name, and the corresponding implicit dependency name is made by appending the source suffix. A single-suffix rule whose source suffix is '`.c`' is equivalent to the pattern rule '`% : %.c`'.

Suffix rule definitions are recognized by comparing each rule's target against a defined list of known suffixes. When `make` sees a rule whose target is a known suffix, this rule is considered a single-suffix rule. When `make` sees a rule whose target is two known suffixes concatenated, this rule is taken as a double-suffix rule.

For example, '`.c`' and '`.o`' are both on the default list of known suffixes. Therefore, if you define a rule whose target is '`.c.o`', `make` takes it to be a double-suffix rule with source suffix '`.c`' and target suffix '`.o`'. For example, here is the old fashioned way to define the rule for compiling a C source:

```
.c.o:
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

Suffix rules cannot have any dependencies of their own. If they have any, they are treated as normal files with funny names, not as suffix rules. Thus, the rule:

```
.c.o: foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

tells how to make the file '`.c.o`' from the dependency file '`foo.h`', and is not at all like the pattern rule:

```
%.o: %.c foo.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

which tells how to make `.o` files from `.c` files, and makes all `.o` files using this pattern rule also depend on '`foo.h`'.

Suffix rules with no commands are also meaningless. They do not remove previous rules as do pattern rules with no commands (see Section 11.5.6 [Canceling Rules], page 102). They simply enter the suffix or pair of suffixes concatenated as a target in the data base.

The known suffixes are simply the names of the dependencies of the special target `.SUFFIXES`. You can add your own suffixes by writing a rule for `.SUFFIXES` that adds more dependencies, as in:

```
.SUFFIXES: .hack .win
```

which adds '`.hack`' and '`.win`' to the end of the list of suffixes.

If you wish to eliminate the default known suffixes instead of just adding to them, write a rule for `.SUFFIXES` with no dependencies. By special dispensation, this eliminates all existing dependencies of `.SUFFIXES`. You can then write another rule to add the suffixes you want. For example,

```
.SUFFIXES:            # Delete the default suffixes
.SUFFIXES: .c .o .h   # Define our suffix list
```

The '`-r`' flag causes the default list of suffixes to be empty.

The variable `SUFFIXES` is defined to the default list of suffixes before `make` reads any makefiles. You can change the list of suffixes with a rule for the special target `.SUFFIXES`, but that does not alter this variable.

11.8 Implicit Rule Search Algorithm

Here is the procedure `make` uses for searching for an implicit rule for a target *t*. This procedure is followed for each double-colon rule with no commands, for each target of ordinary rules none of which have commands, and for each dependency that is not the target of any rule. It is also followed recursively for dependencies that come from implicit rules, in the search for a chain of rules.

Suffix rules are not mentioned in this algorithm because suffix rules are converted to equivalent pattern rules once the makefiles have been read in.

For an archive member target of the form '*archive*(*member*)', the following algorithm is run twice, first using '(*member*)' as the target *t*, and second using the entire target if the first run found no rule.

1. Split *t* into a directory part, called *d*, and the rest, called *n*. For example, if *t* is '`src/foo.o`', then *d* is '`src/`' and *n* is '`foo.o`'.
2. Make a list of all the pattern rules one of whose targets matches *t* or *n*. If the target pattern contains a slash, it is matched against *t*; otherwise, against *n*.
3. If any rule in that list is *not* a match-anything rule, then remove all nonterminal match-anything rules from the list.
4. Remove from the list all rules with no commands.
5. For each pattern rule in the list:
    1. Find the stem *s*, which is the nonempty part of *t* or *n* matched by the '`%`' in the target pattern.
    2. Compute the dependency names by substituting *s* for '`%`'; if the target pattern does not contain a slash, append *d* to the front of each dependency name.
    3. Test whether all the dependencies exist or ought to exist. (If a file name is mentioned in the makefile as a target or as an explicit dependency then we say it ought to exist.)

       If all dependencies exist or ought to exist, or there are no dependencies, then this rule applies.
6. If no pattern rule has been found so far, try harder. For each pattern rule in the list:
    1. If the rule is terminal, ignore it and go on to the next rule.
    2. Compute the dependency names as before.
    3. Test whether all the dependencies exist or ought to exist.
    4. For each dependency that does not exist, follow this algorithm recursively to see if the dependency can be made by an implicit rule.
    5. If all dependencies exist, ought to exist, or can be made by implicit rules, then this rule applies.
7. If no implicit rule applies, the rule for `.DEFAULT`, if any, applies. In that case, give *t* the same commands that `.DEFAULT` has. Otherwise, there are no commands for *t*.

Once a rule that applies has been found, for each target pattern of the rule other than the one that matched *t* or *n*, the '`%`' in the pattern is replaced with *s* and the resultant file name is stored until the commands to remake the target file *t* are executed. After these commands are executed, each of these stored file names are entered into the data base and marked as having been updated and having the same update status as the file *t*.

When the commands of a pattern rule are executed for t, the automatic variables are set corresponding to the target and dependencies. See Section 11.5.3 [Automatic], page 99.

12 Using make to Update Archive Files

Archive files are files containing named subfiles called *members*; they are maintained with the program `ar` and their main use is as subroutine libraries for linking.

12.1 Archive Members as Targets

An individual member of an archive file can be used as a target or dependency in `make`. The archive file must already exist, but the member need not exist. You specify the member named *member* in archive file *archive* as follows:

```
archive(member)
```

This construct is available only in targets and dependencies, not in commands! Most programs that you might use in commands do not support this syntax and cannot act directly on archive members. Only `ar` and other programs specifically designed to operate on archives can do so. Therefore, valid commands to update an archive member target probably must use `ar`. For example, this rule says to create a member '`hack.o`' in archive '`foolib`' by copying the file '`hack.o`':

```
foolib(hack.o) : hack.o
        ar r foolib hack.o
```

In fact, nearly all archive member targets are updated in just this way and there is an implicit rule to do it for you.

12.2 Implicit Rule for Archive Member Targets

Recall that a target that looks like '`a(m)`' stands for the member named *m* in the archive file a.

When `make` looks for an implicit rule for such a target, as a special feature it considers implicit rules that match '`(m)`', as well as those that match the actual target '`a(m)`'.

This causes one special rule whose target is '`(%)`' to match. This rule updates the target '`a(m)`' by copying the file *m* into the archive. For example, it will update the archive member target '`foo.a(bar.o)`' by copying the *file* '`bar.o`' into the archive '`foo.a`' as a *member* named '`bar.o`'.

When this rule is chained with others, the result is very powerful. Thus, '`make "foo.a(bar.o)"`' in the presence of a file '`bar.c`' is enough to cause the following commands to be run, even without a makefile:

```
cc -c bar.c -o bar.o
ar r foo.a bar.o
rm -f bar.o
```

Here `make` has envisioned the file '`bar.o`' as an intermediate file.

Implicit rules such as this one are written using the automatic variable '`$%`'. See Section 11.5.3 [Automatic], page 99.

An archive member name in an archive cannot contain a directory name, but it may be useful in a makefile to pretend that it does. If you write an archive member target '`foo.a(dir/file.o)`', `make` will perform automatic updating with this command:

```
ar r foo.a dir/file.o
```

which has the effect of copying the file '`dir/foo.o`' into a member named '`foo.o`'. In connection with such usage, the automatic variables `%D` and `%F` may be useful.

12.2.1 Updating Archive Symbol Directories

An archive file that is used as a library usually contains a special member named '`__.SYMDEF`' that contains a directory of the external symbol names defined by all the other members. After you update any other members, you need to update '`__.SYMDEF`' so that it will summarize the other members properly. This is done by running the `ranlib` program:

`ranlib` *archivefile*

Normally you would put this command in the rule for the archive file, and make all the members of the archive file dependents of that rule. For example,

```
libfoo.a: libfoo.a(x.o) libfoo.a(y.o) ...
        ranlib libfoo.a
```

The effect of this is to update archive members '`x.o`', '`y.o`', etc., and then update the symbol directory member '__.SYMDEF' by running ranlib. The rules for updating the members are not shown here; most likely you can omit them and use the implicit rule which copies files into the archive, as described in the preceding section.

This is not necessary when using the GNU ar program, which updates the '__.SYMDEF' member automatically.

13 Features of GNU make

Here is a summary of the features of GNU make, for comparison with and credit to other versions of make. We consider the features of make in BSD 4.2 systems as a baseline.

Many features come from the version of make in System V.

- The VPATH variable and its special meaning. See Section 5.3 [Directory Search], page 29. This feature exists in System V make, but is undocumented. It is documented in 4.3 BSD make (which says it mimics System V's VPATH feature).
- Included makefiles. See Section 4.3 [Include], page 20.
- Variables are read from and communicated via the environment. See Section 7.8 [Environment], page 64.
- Options passed through the variable MAKEFLAGS to recursive invocations of make. See Section 6.6.3 [Options/Recursion], page 50.
- The automatic variable $% is set to the member name in an archive reference. See Section 11.5.3 [Automatic], page 99.
- The automatic variables $@, $*, $< and $% have corresponding forms like $(@F) and $(@D). See Section 11.5.3 [Automatic], page 99.
- Substitution variable references. See Section 7.1 [Reference], page 55.
- The command-line options '-b' and '-m', accepted and ignored.
- Execution of recursive commands to run make via the variable MAKE even if '-n', '-q' or '-t' is specified. See Section 6.6 [Recursion], page 47.
- Support for suffix '.a' in suffix rules. In GNU make, this is actually implemented by chaining with one pattern rule for installing members in an archive. See Section 11.4 [Chained Rules], page 95.
- The arrangement of lines and backslash-newline combinations in commands is retained when the commands are printed, so they appear as they do in the makefile, except for the stripping of initial whitespace.

The following features were inspired by various other versions of make. In some cases it is unclear exactly which versions inspired which others.

- Pattern rules using '%'. This has been implemented in several versions of make. We're not sure who invented it first, but it's been spread around a bit. See Section 11.5 [Pattern Rules], page 96.

- Rule chaining and implicit intermediate files. This was implemented by Stu Feldman in his version of `make` for AT&T Eighth Edition Research Unix, and later by Andrew Hume of AT&T Bell Labs in his `mk` program. We don't really know if we got this from either of them or thought it up ourselves at the same time. See Section 11.4 [Chained Rules], page 95.
- The automatic variable `$^` containing a list of all dependencies of the current target. We didn't invent this, but we have no idea who did. See Section 11.5.3 [Automatic], page 99.
- The "what if" flag ('`-W`' in GNU `make`) was (as far as we know) invented by Andrew Hume in `mk`. See Section 10.3 [Instead of Execution], page 83.
- The concept of doing several things at once (parallelism) exists in many incarnations of `make` and similar programs, though not in the System V or BSD implementations. See Section 6.2 [Execution], page 44.
- Modified variable references using pattern substitution come from SunOS 4.0. See Section 7.1 [Reference], page 55. This functionality was provided in GNU `make` by the `patsubst` function before the alternate syntax was implemented for compatibility with SunOS 4.0. It is not altogether clear who inspired whom, since GNU `make` had `patsubst` before SunOS 4.0 was released.
- The special significance of '`+`' characters preceding command lines (see Section 10.3 [Instead of Execution], page 83) is mandated by draft 8 of IEEE Std 1003.2 (POSIX).

The remaining features are inventions new in GNU `make`:

- The '`-v`' option to print version and copyright information.
- Simply-expanded variables. See Section 7.2 [Flavors], page 56.
- Passing command-line variable assignments automatically through the variable `MAKE` to recursive `make` invocations. See Section 6.6 [Recursion], page 47.
- The '`-C`' command option to change directory. See Section 10.7 [Options], page 86.
- Verbatim variable definitions made with `define`. See Section 7.7 [Defining], page 63.
- Phony targets declared with the special target `.PHONY`. A similar feature with a different syntax was implemented by Andrew Hume of AT&T Bell Labs in his `mk` program. This seems to be a case of parallel discovery. See Section 5.4 [Phony Targets], page 32.
- Text manipulation by calling functions. See Chapter 9 [Functions], page 71.
- The '`-o`' option to pretend a file's modification-time is old. See Section 10.4 [Avoiding Compilation], page 84.
- Conditional execution. This has been implemented numerous times in various versions of `make`; it seems a natural extension derived from the features of the C preprocessor and similar macro languages and is not a revolutionary concept. See Chapter 8 [Conditionals], page 67.
- The included makefile search path. See Section 4.3 [Include], page 20.
- Specifying extra makefiles to read. See Section 4.4 [MAKEFILES Variable], page 21.

- Stripping leading sequences of '`./`' from file names, so that '`./`*file*' and '*file*' are considered to be the same file.
- Special search method for library dependencies written in the form '`-l`*name*'. See Section 5.3.5 [Libraries/Search], page 32.
- Allowing suffixes for suffix rules (see Section 11.7 [Suffix Rules], page 104) to contain any characters. In other version of `make`, they must begin with '`.`' and not contain any '`/`' characters.
- The variable `MAKELEVEL` which keeps track of the current level of `make` recursion. See Section 6.6 [Recursion], page 47.
- Static pattern rules. See Section 5.9 [Static Pattern], page 37.
- Selective `vpath` search. See Section 5.3 [Directory Search], page 29.
- Recursive variable references. See Section 7.1 [Reference], page 55.
- Updated makefiles. See Section 4.5 [Remaking Makefiles], page 21. System V `make` has a very, very limited form of this functionality in that it will check out SCCS files for makefiles.
- Several new built-in implicit rules. See Section 11.2 [Catalogue of Rules], page 90.

14 Missing Features in GNU make

The make programs in various other systems support a few features that are not implemented in GNU make. Draft 11.1 of the POSIX.2 standard which specifies make does not require any of these features.

- A target of the form '*file*((*entry*))' stands for a member of archive file *file*. The member is chosen, not by name, but by being an object file which defines the linker symbol *entry*.

  This feature was not put into GNU make because of the nonmodularity of putting knowledge into make of the internal format of archive file symbol directories. See Section 12.2.1 [Archive Symbols], page 110.

- Suffixes (used in suffix rules) that end with the character '~' have a special meaning; they refer to the SCCS file that corresponds to the file one would get without the '~'. For example, the suffix rule '.c~.o' would make the file '*n*.o' file from the SCCS file 's.*n*.c'. For complete coverage, a whole series of such suffix rules is required. See Section 11.7 [Suffix Rules], page 104.

  In GNU make, this entire series of cases is handled by two pattern rules for extraction from SCCS, in combination with the general feature of rule chaining. See Section 11.4 [Chained Rules], page 95.

- In System V make, the string '$$@' has the strange meaning that, in the dependencies of a rule with multiple targets, it stands for the particular target that is being processed.

  This is not defined in GNU make because '$$' should always stand for an ordinary '$'.

  It is possible to get this functionality through the use of static pattern rules (see Section 5.9 [Static Pattern], page 37). The System V make rule:

  ```
  $(targets): $$@.o lib.a
  ```

  can be replaced with the GNU make static pattern rule:

  ```
  $(targets): %: %.o lib.a
  ```

- In System V and 4.3 BSD make, files found by VPATH search (see Section 5.3 [Directory Search], page 29) have their names changed inside command strings. We feel it is much cleaner to always use automatic variables and thus make this feature obsolete.

- In some Unix makes, implicit rule search (see Chapter 11 [Implicit], page 89) is apparently done for *all* targets, not just those without commands. This means you can do:

  ```
  foo.o:
          cc -c foo.c
  ```

  and Unix make will intuit that 'foo.o' depends on 'foo.c'.

  We feel that such usage is broken. The dependency properties of make are well-defined (for GNU make, at least), and doing such a thing simply does not fit the model.

Index of Concepts

J

M

N

O

P

Q

R

S

T

U

V

W

Index of Functions, Variables, and Directives

Short Contents

Table of Contents