PART III
Appendixes

The final section of this book contains information that is not central to the goals ofthe book, but you might find it useful for unusual situations. Appendix A lists the options that the GNU `make` command accepts on the command line. Appendix B, which you will find amusing and possibly useful, stretches `make` to act as a general-purpose programming language with data structures and arithmetic operations. Appendix C contains the free license for the book.

APPENDIX A
Running make

GNU `make` has an impressive set of command-line options. Most command-line options include a short form and a long form. Short commands are indicated with a single dash followed by a single character, while long options begin with a double dash usually followed by whole words separated by dashes. The syntax of these commands is:

```
-o argument
--option-word=argument
```

The following are the most commonly used options to `make`. For a complete listing, see the GNU `make` manual or type `make --help`.

`--always-make`
`-B`
: Assume every target is out of date and update them all.

`--directory=`*`directory`*
`-C` *`directory`*
: Change to the given directory before searching for a *makefile* or performing any work. This also sets the variable `CURDIR` to *`directory`*.

`--environment-overrides`
`-e`
: Prefer environment variables to *makefile* variables when there is a choice. This command-line option can be overridden in the *makefile* for particular variables with the `override` directive.

`--file=`*`makefile`*
`-f` *`makefile`*
: Read the given file as the *makefile* rather than any of the default names (i.e., *makefile*, *Makefile*, or *GNUMakefile*).

`--help`
`-h`
: Print a brief summary of the command-line options.

`--include-dir=`*`directory`*
`-I` *`directory`*
: If an include file does not exist in the current directory, look in the indicated directories for include files before searching the compiled-in search path. Any number of `--include-dir` options can be given on the command line.

`--keep-going`
`-k`
: Do not terminate the `make` process if a command returns an error status. Instead, skip the remainder of the current target, and continue on with other targets.

`--just-print`
`-n`
: Display the set of commands that would be executed by `make`, but do not execute any commands from command scripts. This is very useful when you want to know what `make` will do before actually doing it. Be aware that this option does not prevent code in `shell` functions from executing, just commands in command scripts.

`--old-file=`*`file`*
`-o` *`file`*
: Treat *`file`* as if it were infinitely old, and perform the appropriate actions to update the goals. This can be very useful if a file has been accidentally touched or to determine the effect of one prerequisite on the dependency graph. This is the complement of `--new-file` (`-W`).

`--print-data-base`
`-p`
: Print `make`'s internal database.

`--touch`
`-t`
: Execute the `touch` program on each out-of-date target to update its timestamp. This can be useful in bringing the files in a dependency graph up to date. For instance, editing a comment in a central header file may cause `make` to unnecessarily recompile an immense amount of code. Instead of performing the compile and wasting machine cycles, you can use the `--touch` option to force all files to be up to date.

`--new-file=`*`file`*
`-W` *`file`*
: Assume *`file`* is newer than any target. This can be useful in forcing an update on targets without having to edit or touch a file. This is the complement of `--old-file`.

`--warn-undefined-variables`

Print a warning message if an undefined variable is expanded. This is a useful diagnostic tool since undefined variables quietly collapse into nothing. However, it is also common to include empty variables in *makefile*s for customization purposes. Any unset customization variables will be reported by this option as well.

APPENDIX B

The Outer Limits

As you've seen, GNU make can do some pretty incredible things, but I haven't seen very much that really pushes the limits of make 3.80 with its eval construct. In this exercise, we'll see if we can stretch it further than usual.

Data Structures

One of the limitations of make that occasionally chaffs when writing complex *makefile*s is make's lack of data structures. In a very limited way, you can simulate a data structure by defining variables with embedded periods (or even -> if you can stand it):

```
file.path = /foo/bar
file.type = unix
file.host = oscar
```

If pressed, you can even "pass" this file structure to a function by using computed variables:

```
define remote-file
  $(if $(filter unix,$($1.type)), \
    /net/$($1.host)/$($1.path),   \
    //$($1.host)/$($1.path))
endef
```

Nevertheless, this seems an unsatisfying solution for several reasons:

- You cannot easily allocate a instance of this "structure." Creating a new instance involves selecting a new variable name and assigning each element. This also means these pseudo-instances are not guaranteed to have the same fields (called *slots*).
- The structure exists only in the user's mind, and as a set of different make variables, rather than as a unified entity with its own name. And because the structure has no name, it is difficult to create a reference (or pointer) to a structure, so passing them as arguments or storing one in a variable is clumsy.

- There is no safe way to access a slot of the structure. Any typographical error in either part of the variable name yields the wrong value (or no value) with no warning from `make`.

But the `remote-file` function hints at a more comprehensive solution. Suppose we implement structure instances using computed variables. Early Lisp object systems (and even some today) used similar techniques. A structure, say `file-info`, can have instances represented by a symbolic name, such as `file_info_1`.

Another instance might be called `file_info_2`. The slots of this structure can be represented by computed variables:

```
file_info_1_path
file_info_1_type
file_info_1_host
```

Since the instance has a symbolic name, it can be saved in one or more variables (as usual, using recursive or simple variables is the choice of the programmer):

```
before_foo = file_info_1
another_foo = $(before_foo)
```

Elements of a `file-info` can be accessed using Lisp-like getters and setters:

```
path := $(call get-value,before_foo,path)
$(call set-value,before_foo,path,/usr/tmp/bar)
```

We can go further than this by creating a template for the `file-info` structure to allow the convenient allocation of new instances:

```
orig_foo := $(call new,file-info)
$(call set-value,orig_foo,path,/foo/bar)

tmp_foo := $(call new,file-info)
$(call set-value,tmp_foo,path,/tmp/bar)
```

Now, two distinct instances of <literal>file-info</literal> exist. As a final convenience, we can add the concept of default values for slots. To declare the `file-info` structure, we might use:

```
$(call defstruct,file-info,  \
  $(call defslot,path,),     \
  $(call defslot,type,unix), \
  $(call defslot,host,oscar))
```

Here, the first argument to the `defstruct` function is the structure name, followed by a list of `defslot` calls. Each `defslot` contains a single (*`name`*, *`default value`*) pair. Example B-1 shows the implementation of `defstruct` and its supporting code.

Example B-1. Structure definition in make

```
# $(next-id) - return a unique number
next_id_counter :=
define next-id
$(words $(next_id_counter))$(eval next_id_counter += 1)
endef
```

Example B-1. Structure definition in make (continued)

```
# all_structs - a list of the defined structure names
all_structs :=

value_sep := XxSepxX

# $(call defstruct, struct_name, $(call defslot, slot_name, value), ...)
define defstruct
  $(eval all_structs += $1)                                             \
  $(eval $1_def_slotnames :=)                                           \
  $(foreach v, $2 $3 $4 $5 $6 $7 $8 $9 $(10) $(11),                     \
    $(if $($v_name),                                                    \
      $(eval $1_def_slotnames          += $($v_name))                   \
      $(eval $1_def_$($v_name)_default := $($v_value))))
endef

# $(call defslot,slot_name,slot_value)
define defslot
  $(eval tmp_id := $(next_id))
  $(eval $1_$(tmp_id)_name := $1)
  $(eval $1_$(tmp_id)_value := $2)
  $1_$(tmp_id)
endef

# all_instances - a list of all the instances of any structure
all_instances :=

# $(call new, struct_name)
define new
$(strip                                                         \
  $(if $(filter $1,$(all_structs)),,                            \
    $(error new on unknown struct '$(strip $1)'))               \
  $(eval instance := $1@$(next-id))                             \
  $(eval all_instances += $(instance))                          \
  $(foreach v, $($(strip $1)_def_slotnames),                    \
    $(eval $(instance)_$v := $($(strip $1)_def_$v_default)))    \
  $(instance))
endef

# $(call delete, variable)
define delete
$(strip                                                                  \
  $(if $(filter $($(strip $1)),$(all_instances)),,                       \
    $(error Invalid instance '$($(strip $1))'))                          \
  $(eval all_instances := $(filter-out $($(strip $1)),$(all_instances))) \
  $(foreach v, $($(strip $1)_def_slotnames),                             \
    $(eval $(instance)_$v := )))
endef

# $(call struct-name, instance_id)
define struct-name
$(firstword $(subst @, ,$($(strip $1))))
endef
```

Example B-1. Structure definition in make (continued)

```
# $(call check-params, instance_id, slot_name)
define check-params
  $(if $(filter $($(strip $1)),$(all_instances)),,              \
    $(error Invalid instance '$(strip $1)'))                    \
  $(if $(filter $2,$($(call struct-name,$1)_def_slotnames)),,   \
    $(error Instance '$($(strip $1))' does not have slot '$(strip $2)'))
endef

# $(call get-value, instance_id, slot_name)
define get-value
$(strip                         \
  $(call check-params,$1,$2)    \
  $($($(strip $1))_$(strip $2)))
endef

# $(call set-value, instance_id, slot_name, value)
define set-value
  $(call check-params,$1,$2) \
  $(eval $($(strip $1))_$(strip $2) := $3)
endef

# $(call dump-struct, struct_name)
define dump-struct
{ $(strip $1)_def_slotnames "$($(strip $1)_def_slotnames)"      \
  $(foreach s,                                                  \
    $($(strip $1)_def_slotnames),$(strip                        \
    $(strip $1)_def_$s_default "$($(strip $1)_def_$s_default)")) }
endef

# $(call print-struct, struct_name)
define print-struct
{ $(foreach s,                                  \
    $($(strip $1)_def_slotnames),$(strip        \
    { "$s" "$($(strip $1)_def_$s_default)" })) }
endef

# $(call dump-instance, instance_id)
define dump-instance
{ $(eval tmp_name := $(call struct-name,$1))    \
  $(foreach s,                                  \
    $($(tmp_name)_def_slotnames),$(strip        \
    { $($(strip $1))_$s "$($($(strip $1))_$s)" })) }
endef

# $(call print-instance, instance_id)
define print-instance
{ $(foreach s,                                          \
    $($(call struct-name,$1)_def_slotnames),"$(strip    \
    $(call get-value,$1,$s))") }
endef
```

Examining this code one clause at a time, you can see that it starts by defining the function next-id. This is a simple counter:

```
# $(next-id) - return a unique number
next_id_counter :=
define next-id
$(words $(next_id_counter))$(eval next_id_counter += 1)
endef
```

It is often said that you cannot perform arithmetic in make, because the language is too limited. In general, this is true, but for limited cases like this you can often compute what you need. This function uses eval to redefine the value of a simple variable. The function contains two expressions: the first expression returns the number of words in next_id_counter; the second expression appends another word to the variable. It isn't very efficient, but for numbers in the small thousands it is fine.

The next section defines the defstruct function itself and creates the supporting data structures.

```
# all_structs - a list of the defined structure names
all_structs :=

value_sep := XxSepxX

# $(call defstruct, struct_name, $(call defslot, slot_name, value), ...)
define defstruct
  $(eval all_structs += $1)                               \
  $(eval $1_def_slotnames :=)                             \
  $(foreach v, $2 $3 $4 $5 $6 $7 $8 $9 $(10) $(11),      \
    $(if $($v_name),                                      \
      $(eval $1_def_slotnames          += $($v_name))    \
      $(eval $1_def_$($v_name)_default := $($v_value))))
endef

# $(call defslot,slot_name,slot_value)
define defslot
  $(eval tmp_id := $(next_id))
  $(eval $1_$(tmp_id)_name := $1)
  $(eval $1_$(tmp_id)_value := $2)
  $1_$(tmp_id)
endef
```

The variable all_structs is a list of all known structures defined with defstruct. This list allows the new function to perform type checking on the structures it allocates. For each structure, *S*, the defstruct function defines a set of variables:

```
S_def_slotnames
S_def_slotn_default
```

The first variable defines the set of slots for a structure. The other variables define the default value for each slot. The first two lines of the defstruct function append to all_structs and initialize the slot names list, respectively. The remainder of the function iterates through the slots, building the slot list and saving the default value.

Each slot definition is handled by defslot. The function allocates an id, saves the slot name and value in two variables, and returns the prefix. Returning the prefix allows the argument list of defstruct to be a simple list of symbols, each of which provides access to a slot definition. If more attributes are added to slots later, incorporating them into defslot is straightforward. This technique also allows default values to have a wider range of values (including spaces) than simpler, alternative implementations.

The foreach loop in defstruct determines the maximum number of allowable slots. This version allows for 10 slots. The body of the foreach processes each argument by appending the slot name to S_def_slotnames and assigning the default value to a variable. For example, our file-info structure would define:

```
file-info_def_slotnames := path type host
file-info_def_path_default :=
file-info_def_type_default := unix
file-info_def_host_default := oscar
```

This completes the definition of a structure.

Now that we can define structures, we need to be able to instantiate one. The new function performs this operation:

```
# $(call new, struct_name)
define new
$(strip                                                    \
  $(if $(filter $1,$(all_structs)),,                       \
    $(error new on unknown struct '$(strip $1)'))          \
  $(eval instance := $1@$(next-id))                        \
  $(eval all_instances += $(instance))                     \
  $(foreach v, $($(strip $1)_def_slotnames),               \
    $(eval $(instance)_$v := $($(strip $1)_def_$v_default)))    \
  $(instance))
endef
```

The first if in the function checks that the name refers to a known structure. If the structure isn't found in all_structs, an error is signaled. Next, we construct a unique id for the new instance by concatenating the structure name with a unique integer suffix. We use an at sign to separate the structure name from the suffix so we can easily separate the two later. The new function then records the new instance name for type checking by accessors later. Then the slots of the structure are initialized with their default values. The initialization code is interesting:

```
$(foreach v, $($(strip $1)_def_slotnames),               \
  $(eval $(instance)_$v := $($(strip $1)_def_$v_default)))
```

The foreach loop iterates over the slot names of the structure. Using strip around on the structure name allows the user to add spaces after commas in the call to new. Recall that each slot is represented by concatenating the instance name and the slot name (for instance, file_info@1_path). The righthand side is the default value computed from the structure name and slot name. Finally, the instance name is returned by the function.

Note that I call these constructs functions, but they are actually macros. That is, the symbol `new` is recursively expanded to yield a new piece of text that is inserted into the *makefile* for reparsing. The reason the `defstruct` macro does what we want is because all the work is eventually embedded within `eval` calls, which collapse to nothing. Similarly, the `new` macro performs its significant work within `eval` calls. It can reasonably be termed a function, because the expansion of the macro conceptually yields a single value, the symbol representing the new instance.

Next, we need to be able to get and set values within our structures. To do this, we define two new functions:

```
# $(call get-value, instance_id, slot_name)
define get-value
$(strip                         \
  $(call check-params,$1,$2)    \
  $($($(strip $1))_$(strip $2)))
endef

# $(call set-value, instance_id, slot_name, value)
define set-value
  $(call check-params,$1,$2) \
  $(eval $($(strip $1))_$(strip $2) := $3)
endef
```

To get the value of a slot, we simply need to compute the slot variable name from the instance id and the slot name. We can improve safety by first checking that the instance and slot name are valid strings with the `check-params` function. To allow more aesthetic formating and to ensure that extraneous spaces do not corrupt the slot value, we wrap most of these parameters in `strip` calls.

The set function also checks parameters before setting the value. Again, we strip the two function arguments to allow users the freedom to add spaces in the argument list. Note that we do not strip the slot value, because the user might actually need the spaces.

```
# $(call check-params, instance_id, slot_name)
define check-params
  $(if $(filter $($(strip $1)),$(all_instances)),,              \
    $(error Invalid instance '$(strip $1)'))                    \
  $(if $(filter $2,$($(call struct-name,$1)_def_slotnames)),,   \
    $(error Instance '$($(strip $1))' does not have slot '$(strip $2)'))
endef

# $(call struct-name, instance_id)
define struct-name
$(firstword $(subst @, ,$($(strip $1))))
endef
```

The `check-params` function simply checks that the instance id passed to the setter and getter functions is contained within the known instances list. Likewise, it checks that the slot name is contained within the list of slots belonging to this structure. The structure name is computed from the instance name by splitting the symbol on the `@` and taking the first word. This means that structure names cannot contain an at sign.

To round out the implementation, we can add a couple of print and debugging functions. The following print function displays a simple user-readable representation of a structure definition and a structure instance, while the dump function displays the implementation details of the two constructs. See Example B-1 for the implementations.

Here's an example defining and using our `file-info` structure:

```
include defstruct.mk

$(call defstruct, file-info,    \
  $(call defslot, path,),       \
  $(call defslot, type,unix),   \
  $(call defslot, host,oscar))

before := $(call new, file-info)
$(call set-value, before, path,/etc/password)
$(call set-value, before, host,wasatch)

after := $(call new,file-info)
$(call set-value, after, path,/etc/shadow)
$(call set-value, after, host,wasatch)

demo:
        # before       = $(before)
        # before.path  = $(call get-value, before, path)
        # before.type  = $(call get-value, before, type)
        # before.host  = $(call get-value, before, host)
        # print before = $(call print-instance, before)
        # dump before  = $(call dump-instance, before)
        #
        # all_instances   = $(all_instances)
        # all_structs     = $(all_structs)
        # print file-info = $(call print-struct, file-info)
        # dump file-info  = $(call dump-struct, file-info)
```

and the output:

```
$ make
# before       = file-info@0
# before.path  = /etc/password
# before.type  = unix
# before.host  = wasatch
# print before = { "/etc/password" "unix" "wasatch" }
# dump before  = {  { file-info@0_path "/etc/password" } { file-info@0_type "unix" }
{ file-info@0_host "wasatch" } }
#
# all_instances   =  file-info@0 file-info@1
# all_structs     =  file-info
# print file-info = { { "path" "" } { "type" "unix" } { "host" "oscar" } }
# dump file-info  = { file-info_def_slotnames " path type host" file-info_def_path_
default "" file-info_def_type_default "unix" file-info_def_host_default "oscar" }
```

Also note how illegal structure uses are trapped:

```
$ cat badstruct.mk
include defstruct.mk
$(call new, no-such-structure)
$ make -f badstruct.mk
badstruct.mk:2: *** new on unknown struct 'no-such-structure'.  Stop.

$ cat badslot.mk
include defstruct.mk
$(call defstruct, foo, defslot(size, 0))
bar := $(call new, foo)
$(call set-value, bar, siz, 10)
$ make -f badslot.mk
badslot.mk:4: *** Instance 'foo@0' does not have slot 'siz'.  Stop.
```

Of course, there are lots of improvements that can be made to the code, but the basic ideas are sound. Here is a list of possible enhancements:

- Add a validation check to the slot assignment. This could be done with a hook function that must yield empty after the assignment has been performed. The hook could be used like this:

  ```
  # $(call set-value, instance_id, slot_name, value)
  define set-value
    $(call check-params,$1,$2)                                  \
    $(if $(call $(strip $1)_$(strip $2)_hook, value),           \
      $(error set-value hook, $(strip $1)_$(strip $2)_hook, failed)) \
    $(eval $($(strip $1))_$(strip $2) := $3)
  endef
  ```

- Support for inheritance. A `defstruct` could accept another `defstruct` name as a superclass, duplicating all the superclass's members in the subclass.
- Better support for structure references. With the current implementation, a slot can hold the ID of another structure, but accessing is awkward. A new version of the `get-value` function could be written to check for references (by looking for *`defstruct@number`*), and perform automatic dereferencing.

Arithmetic

In the previous section, I noted that it is not possible to perform arithmetic in `make` using only its native features. I then showed how you could implement a simple counter by appending words to a list and returning the length of the list. Soon after I discovered the increment trick, Michael Mounteney posted a cool trick for performing a limited form of addition on integers in `make`.

His trick manipulates the number line to compute the sum of two integers of size one or greater. To see how this works, imagine the number line:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Now, notice that (if we could get the subscripts right), we could add, say 4 plus 5, by first taking a subset of the line from the fourth element to the end then selecting the fifth element of the subset. We can do this with native make functions:

```
number_line = 2 3 4 5 6 7 8 9 10 11 12 13 14 15
plus = $(word $2, $(wordlist $1, 15, $(number_line)))
four+five = $(call plus, 4, 5)
```

Very clever, Michael! Notice that the number line starts with 2 rather than 0 or 1. You can see that this is necessary if you run the plus function with 1 and 1. Both subscripts will yield the first element and the answer must be 2, therefore, the first element of the list must be 2. The reason for this is that, for the word and wordlist functions, the first element of the list has subscript 1 rather than 0 (but I haven't bothered to prove it).

Now, given a number line, we can perform addition, but how do we create a number line in make without typing it in by hand or using a shell program? We can create all numbers between 00 and 99 by combining all possible values in the tens place with all possible values in the ones place. For example:

```
make -f - <<< '$(warning $(foreach i, 0 1 2, $(addprefix $i, 0 1 2)))'
/c/TEMP/Gm002568:1:  00 01 02  10 11 12  20 21 22
```

By including all digits 0 through 9, we would produce all numbers from 00 to 99. By combining the foreach again with a hundreds column, we would get the numbers from 000 to 999, etc. All that is left is to strip the leading zeros where necessary.

Here is a modified form of Mr. Mounteney's code to generate a number line and define the plus and gt operations:

```
# combine - concatentate one sequence of numbers with another
combine = $(foreach i, $1, $(addprefix $i, $2))

# stripzero - Remove one leading zero from each word
stripzero = $(patsubst 0%,%,$1)

# generate - Produce all permutations of three elements from the word list
generate = $(call stripzero,                            \
             $(call stripzero,                          \
               $(call combine, $1,                      \
                 $(call combine, $1, $1))))

# number_line - Create a number line from 0 to 999
number_line := $(call generate,0 1 2 3 4 5 6 7 8 9)
length      := $(word $(words $(number_line)), $(number_line))

# plus - Use the number line to add two integers
plus = $(word $2,                                       \
         $(wordlist $1, $(length),                      \
           $(wordlist 3, $(length), $(number_line))))

# gt - Use the number line to determine if $1 is greater than $2
gt = $(filter $1,                                       \
       $(wordlist 3, $(length),                       \
         $(wordlist $2, $(length), $(number_line))))


all:
        @echo $(call plus,4,7)
        @echo $(if $(call gt,4,7),is,is not)
        @echo $(if $(call gt,7,4),is,is not)
        @echo $(if $(call gt,7,7),is,is not)
```

When run, the *makefile* yields:

```
$ make
11
is not
is
is not
```

We can extend this code to include subtraction by noting that subscripting a reversed list is just like counting backwards. For example, to compute 7 minus 4, first create the number line subset 0 to 6, reverse it, then select the fourth element:

```
number_line := 0 1 2 3 4 5 6 7 8 9...
1through6   := 0 1 2 3 4 5 6
reverse_it  := 6 5 4 3 2 1 0
fourth_item := 3
```

Here is the algorithm in make syntax:

```
# backwards - a reverse number line
backwards := $(call generate, 9 8 7 6 5 4 3 2 1 0)

# reverse - reverse a list of words
reverse    = $(strip                                \
               $(foreach f,                         \
                 $(wordlist 1, $(length), $(backwards)), \
                 $(word $f, $1)))

# minus - compute $1 minus $2
minus      = $(word $2,                             \
               $(call reverse,                      \
                 $(wordlist 1, $1, $(number_line))))

minus:
        # $(call minus, 7, 4)
```

Multiplication and division are left as an exercise for the reader.

APPENDIX C

GNU Free Documentation License—GNU Project—Free Software Foundation (FSF)

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

1. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
2. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
3. State on the Title page the name of the publisher of the Modified Version, as the publisher.

of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements.”

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document 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.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation 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. See *http://www.gnu.org/copyleft/*.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Index

We’d like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.