Cfpeek User Manual

version 1.2, 6 January 2021

Sergey Poznyakoff.

Short Contents

Table of Contents

1 Introduction

Many programs keep their configurations in files with hierarchical structure. Such files normally define sections, which keep logically separated blocks of statements. These statements may in turn contain subsections, and so on. On the lowest level of hierarchy are *simple statements*, which normally define some basic configuration settings.

Quite often a need arises to parse such files outside of their owner program. For example, one may need to retrieve some configuration settings to use them in a start-up script for that program, to produce a similar configuration file with some settings changed in order to use it on another machine, or to convert entire file into another format for interaction with some other utility.

`Cfpeek` is a utility designed to handle any of these tasks.

2 Overview of this Manual

This book consists of the three main parts. The first part is a tutorial, which provides a gentle (as far as possible) introduction for those who are new to `cfpeek`. The tutorial should help the reader to familiarize himself with the program and to start using it. It does not, however, cover some of the less frequently used features of `cfpeek`.

The chapters that follow complement the tutorial. They describe various input file formats understood by the program and summarize command line syntax and options available to it. These two chapters can be used as a reference by both beginners and for users familiar with the package.

3 Tutorial

The following typographic conventions are used throughout this tutorial.

In the examples, '`$`' represents a typical shell prompt. It precedes lines you should type. Both command line and lines which represent the program output are shown in '`this font`'.

The Scheme code is shown as follows:

```
(do it)
```

In examples, the ⇒ symbol indicates the value of a variable or result of a function invocation, as in:

```
x ⇒ 2
```

3.1 Basic Notions

A structured configuration file contains entities of two basic types. First of them is *simple statement*. A simple statement conceptually consists of an *identifier* (or *keyword*) and a *value*. Depending on the syntactic requirements, some special token may be required between them (such as an equals sign, for example), or at the end of the statement. The *value*, though we use the term in singular, is not necessarily a single scalar value, it may as well be a list of values (the exact form of that list depends on the particular syntax of the configuration file).

Another basic entity is *compound statement*, also known as *block statement* or *section*. Compound statement is used for logical grouping of other statements. It consists of identifier, an optional tag and a list of statements. The tag, if present, is similar to the value in simple statements. The same notes that we made about values apply to tags as well. Tags serve to discern between the statements having the same identifier. The *list of statements* may include statements of both kinds: simple as well as compound ones. Thus, compound statements form a tree-like structure of arbitrary depth, with simple statements as leaf nodes.

Each compound statement can have any number of *subordinate statements*, which are called its *child* statements. Each statement (no matter simple or compound) has only one *parent statement*, i.e. a compound statement of which it is a child.

A special implicit statement, called *root statement*, serves as the parent for the statements at the topmost level of hierarchy.

3.2 Pathnames

Given this hierarchical structure, each statement can be identified by the list of keywords and values (when present) of all compound statements that must be traversed in order to reach that statement. Such a list, written according to a set of conventions, is called a *full pathname* of the statement. The conventions are:

1. Pathname is written from top down.
2. An untagged statement is represented by its identifier.
3. A tagged statement is represented by its identifier, immediately followed by an equals sign, followed by the tag.
4. Identifiers and values which contain whitespace, double quotes or dots are enclosed in double quotes.
5. Within double quotes, a double quote is represented as '`\"`' and a backslash is represented as '`\\`'.
6. Pathname components are separated by dots.

A pathname which begins with a component separator ('`.`') is called *absolute pathname* and identifies the statement with relation to the topmost level of hierarchy.

A pathname beginning with an identifier is called *relative* and identifies the statement in relation to the statement represented by that identifier.

Examples of absolute pathnames are:

```
.database.description
.acl=global.deny
.view=external.zone=com.type
```

Examples of relative pathnames are:

```
description
zone=com.type
```

3.3 Example Configuration

The following configuration file will assist us in further discussion. Its syntax is fairly straightforward:

A simple statement is written as identifier followed value. The two parts are separated by any amount of whitespace. Simple statements are terminated by semicolon.

A compound statement is written as identifier followed by a list of subordinate statements in curly braces. A tag (if present) is put between the identifier and the opening curly brace.

These syntax conventions roughly correspond to the *Grecs configuration format*, which `cfpeek` assumes by default (see Section 4.1 [grecs], page 17).

```
user smith;
group mail;
pidfile "/var/run/example";

logging {
    facility daemon;
    tag example;
}

program a {
    command "a.out";
    logging {
        facility local0;
        tag a;
    }
}

program b {
    command "b.out";
    wait yes;
    pidfile /var/run/b.pid;
}
```

Example 3.1: Sample configuration file

3.4 Listing the Entire File

The only argument `cfpeek` requires is the name of the file to parse. If no other arguments are given, it produces on the standard output a listing of that file in *pathname-value* form. Each simple statement in the input file is represented by a single line in the output listing. The line consists of two main parts: the full pathname of that statement and its value. The two parts are separated by a colon and space character. For example:

```
$ cfpeek sample.conf
.user: smith
.group: mail
.pidfile: /var/run/example
.logging.facility: daemon
.logging.tag: example
.program="a".command: a.out
.program="a".logging.facility: local0
.program="a".logging.tag: a
.program="b".command: b.out
.program="b".wait: yes
.program="b".pidfile: /var/run/b.pid
```

This output can be customized via the `--format` (`-H`) command line option. This option takes a list of *output flags*, each of which modifies some aspect of the output. Most output flags are boolean, i.e. they enable or disable the given feature. To disable the feature, the flag must be prefixed with '`no`'.

To list only the pathnames, use

```
$ cfpeek --format=path sample.conf
.user
.group
.pidfile
.logging.facility
.logging.tag
.program="a".command
.program="a".logging.facility
.program="a".logging.tag
.program="b".command
.program="b".wait
.program="b".pidfile
```

The default output is equivalent to `--format=path,value,descend`.

The flags '`path`' and '`value`' mean to print the pathname of the statement and its value. The '`descend`' flag affects the output of compound nodes. If this flag is set and a node matching the key is a compound node, `cfpeek` will output this node and all nodes below it (i.e. its descendant nodes). The '`descend`' flag is meaningful only if at least one lookup key is supplied.

You can also use `--format` to change the default component delimiter. For example, to use slash to delimit components:

```
$ cfpeek --format=delim=/ sample.conf
/user: smith
/group: mail
/pidfile: /var/run/example
/logging/facility: daemon
/logging/tag: example
/program="a"/command: a.out
/program="a"/logging/facility: local0
/program="a"/logging/tag: a
/program="b"/command: b.out
/program="b"/wait: yes
/program="b"/pidfile: /var/run/b.pid
```

3.5 Statement Lookups

When given more than one argument, `cfpeek` treats the rest of arguments as *search keys*. It then searches for statements with pathnames matching each of the keys and outputs them. A key can be either a pathname, or a pattern.

The following command looks for the '`pidfile`' statement at the topmost level of hierarchy and prints it:

```
$ cfpeek sample.conf .pidfile
.pidfile: /var/run/example
```

As you see, it uses the same output format as with full listings. If you wish to change it, use the `--format` option, introduced in the previous section. For example, to retrieve only the value:

```
$ cfpeek --format=value sample.conf .pidfile
/var/run/example
```

This approach is quite common when `cfpeek` is used in shell scripts. It will be illustrated in more detail below.

If a key is not found, `cfpeek` prints a message on the standard error and starts searching for the next key (if any). When all keys are exhausted, the program exits with status 1 to indicate that some of them have not been found. To suppress the diagnostics output, use the `--quiet` (`-q`) option.

To illustrate all this, the following example shows how to use `cfpeek` in a start-up script to check whether a program has already been started and to bring it down, if requested:

```
#! /bin/sh
pidfile=`cfpeek -q --format=value sample.conf .pidfile`

if test -f $pidfile; then
  pid=`head -1 $pidfile`
else
  pid=
fi

case $1 in
start)  if test -n "$pid"; then
          echo >&2 "the program is already running"
        else
          # start the program
          sample-start
        fi
        ;;
status) if test -n "$pid"; then
          echo "program is running at pid $pid"
        else
          echo "program is not running"
        fi
        ;;
stop)   test -n "$pid" && kill -TERM $pid
        ;;
esac
```

3.6 Pattern Lookups

Apart from literal pathname, a *pathname pattern* is allowed as a key. A pattern can contain *wildcards* in place of path components. Two wildcards are defined: '`*`' and '`%`'. A '`%`' matches any single keyword:

```
$ cfpeek sample.conf .%.pidfile
.program="b".pidfile: /var/run/b.pid
```

A '`*`' wildcard matches zero or more keywords appearing in its place:

```
$ cfpeek sample.conf .*.pidfile
.pidfile: /var/run/example
.program="b".pidfile: /var/run/b.pid
```

In addition to these wildcards, tags in a pattern can contain traditional globbing patterns, as described in Section "match filename or pathname" in *fnmatch(3) man page*.

```
$ cfpeek sample.conf '.program=[ab].pidfile'
.program="b".pidfile: /var/run/b.pid
```

Pattern lookups can be disabled using the `--literal` (`-L`) command line option. There may be two reasons for doing so. First, literal lookups are somewhat faster, so if you don't need pattern matching using `--literal` can save you a couple of CPU cycles. Secondly, if any of your identifiers contain '`*`' or '`%`' characters, you will have to use `--literal` to prevent them from being treated as wildcards.

3.7 Using Various Parsers

`Cfpeek` can handle input files in various formats. The default one is '`Grecs`' format, introduced in previous sections. To process input files of another format, specify the *parser* to use via the `--parser` (`-p`) command line option. The argument to this option is one of: '`grecs`', '`bind`', '`path`', '`meta1`' or '`git`'. See Chapter 4 [Formats], page 17, for a detailed description of each of these formats.

For example, to select zone statements from the `/etc/named.conf` file:

```
$ cfpeek --parser=bind /etc/named.conf '.*.zone'
```

3.8 Specifying Nodes to Output

Sometimes you may need to see not the node which matched the search key, but its parent or other ancestor node. Consider, for example, the following task: select from the `/etc/named.conf` file the names of all zones for which this nameserver is a master. To do so, you will need to find all '`zone.type`' statements with the value '`master`', ascend to the parent node and print its value.

`Cfpeek` provides several special formatting flags to that effect: `up`, `down`, `parent`, `child` and `sibling`. They are called *relative movement* flags, because they select another node in the tree, relative to the position of the current node.

The `up` flag takes an integer number as its argument. It instructs `cfpeek` to ascend that many parent nodes before actually printing the node. For example, `--format=up=1` means “ascend to the parent of the matched node and print it”. This is exactly what we need to solve the above task, since the ‘`type`’ statement is a child of a ‘`zone`’ statement. Thus, the solution is:

```
cfpeek --format=up=1,nodescend,value --parser=bind \
       /etc/named.conf .*.type=master
```

The `value` flag indicates that we want on output only values, without the corresponding pathnames. The `nodescend` flag tells `cfpeek` to not descend into compound statements when outputting them. It is necessary since we want only values of all relevant ‘`zone`’ statements, no their subordinate statements.

A counterpart of this flag is `down=n` flag, which descends *n* levels of hierarchy.

The `parent` flag acts in the similar manner, but it identifies the ancestor by its keyword, instead of the relative nesting level. The statement

```
--format=parent=zone
```

tells `cfpeek`, after finding a matching node, to ascend until a node with the identifier ‘`zone`’ is found, and then print this node.

The `child=id` statement does the opposite of `parent`: it locates a child of the current node which has the identifier *id*.

Similarly, the `sibling` keyword instructs `cfpeek` to find first sibling of the current node wich has the given identifier. For example, to find names of the zone files for all master nodes in the `named.conf` file:

```
cfpeek --parser bind --format=sibling=file,value /etc/named.conf \
       '.*.zone.type=master'
```

A ‘`file`’ statement is located on the same nesting level as ‘`type`’, for example:

```
zone "example.net" {
        type master;
        file "db.example.net";
};
```

Thus, the above command first locates the ‘`type`’ statement, then searches on the same nesting level for a ‘`file`’ statement, and finally prints its value.

3.9 Using Scripts

`Cfpeek` offers a scripting facility, which can be used to easily extend its functionality beyond the basic operations, described in previous chapters. Scripts must be written in Scheme, using ‘`Guile`’, the *GNU’s Ubiquitous Intelligent Language for Extensions*. For information about the language, refer to *Revised(5) Report on the Algorithmic Language Scheme*. For a detailed description of Guile and its features, see Section "Overview" in *The Guile Reference Manual*.

This section assumes that the reader has sufficient knowledge about this programming language.

The scripting facility is enabled by the use of the `--expression` (`-e`) of `--file` (`-f` command line options. The `--expression` (`-e`) option takes as its argument a Scheme expression, which will be executed for each statement matching the supplied keys (or for each statement in the tree, if no keys were supplied). The expression can obtain information about the statement from the global variable `node`, which represents a node in the parse tree describing this statement. The node contains complete information about the statement, including its location in the source file, its type and neighbor nodes, etc. A number of functions is provided to retrieve that information from the node. These functions are discussed in detail in Chapter 7 [Scripting], page 33.

Let's start from the simplest example. The following command prints all nodes in the file:

```
$ cfpeek --expression='(display node)(newline)' sample.conf
#<node .user: "smith">
#<node .group: "mail">
#<node .pidfile: "/var/run/example">
#<node .logging.facility: "daemon">
#<node .logging.tag: "example">
#<node .program="a".command: "a.out">
#<node .program="a".logging.facility: "local0">
#<node .program="a".logging.tag: "a">
#<node .program="b".command: "b.out">
#<node .program="b".wait: "yes">
#<node .program="b".pidfile: "/var/run/b.pid">
```

The format shown in this example is the default Scheme representation for nodes. You can use accessor functions to format the output to your liking. For instance, the function '`grecs-node-locus`' returns the location of the node in the input file. The returned value is a cons, with the file name as its car and the line number as its cdr. Thus, you can print statement locations with the following command:

```
cfpeek --expr='(let ((loc grecs-node-locus))
                 (format #t "~A:~A~%"
                  (car loc) (cdr loc)))' \
       sample.conf
```

Complex expressions are cumbersome to type in the command line, therefore the `--file` (`-f`) option is provided. This option takes the name of the script file as its argument. This file must define the function named `cfpeek` which takes a node as its argument. The script file is then loaded and the `cfpeek` function is called for each matching node.

Now, if we put the expression used in the previous example in a script file (e.g. `locus.scm`):

```
(define (cfpeek node)
  (let ((loc grecs-node-locus))
    (format #t "~A:~A~%" (car loc) (cdr loc))))
```

then the example can be rewritten as:

```
$ cfpeek -f locus.scm sample.conf
```

When both `--file` and `--expression` options are used in the same invocation, the `cfpeek` function is not invoked by default. In fact, it even does not need to be defined. When used this way, `cfpeek` first loads the requested script file, and then applies the expression to each matching node, the same way it always does when `--expression` is supplied. It is the responsibility of the expression itself to call any function or functions defined in the file. This way of invoking '`cfpeek`' is useful for supplying additional parameters to the script. For example:

```
$ cfpeek -f script.scm -e '(process-node node #t)' input.conf
```

It is supposed that the function `process-node` is defined somewhere in `script.scm` and takes two arguments: a node and a boolean.

The `--init=`*expr* (`-i` *expr*) option provides an initialization expression *expr*. This expression is evaluated once, after loading the script file, if one is specified, and before starting the main loop.

Similarly, the option `--done=`*expr* (`-d` *expr*) introduces a Scheme expression to be evaluated at the end of the run, after all nodes have been processed.

3.9.1 Example: Converter to GIT Configuration Format

Here is a more practical example of Scheme scripting. This script converts entire parse tree into a GIT configuration file format. The format itself is described in Section 4.6 [git], page 23.

The script traverses entire tree itself, so it must be called only once, for the root node of the parse tree. The root node is denoted by a single dot, so the invocation syntax is:

```
cfpeek -f togit.scm sample.conf .
```

Traversal is performed by the main function, `cfpeek`, using the `grecs-node-next` and `grecs-node-down` functions. The `grecs-node-next` function returns a node which follows its argument at the same nesting level. For example, if *n* is the very first node in our sample parse tree, then:

```
n ⇒ #<node .user: "smith">
(grecs-node-next n) ⇒ #<node .group: "mail">
```

Similarly, the `grecs-node-down` function returns the first subordinate node of its argument. For example:

```
n ⇒ #<node .logging>
```

```
(grecs-node-down n) ⇒ #<node .logging.facility: "daemon">
```

Both functions return '#f' if there are no next or subordinate node, correspondingly.

The grecs-node-type function is used to determine how to handle that particular node. It returns a *type* of the node given to it as argument. The type is an integer constant, with the following possible values:

Type	**The node is**
grecs-node-root	the root (topmost) node
grecs-node-stmt	a simple statement
grecs-node-block	a compound (block) statement

The print-section function prints a GIT section header corresponding to its node. It ascends the parent node chain to find the topmost node and prints the traversed nodes in the correct order.

To summarize, here is the listing of the togit.scm script:

```
(define (print-section node delim)
  "Print a Git section header for the given node.
End it with delim.

The function recursively calls itself until the topmost
node is reached.
"
  (cond
   ((grecs-node-up? node)
    ;; Ascend to the parent node
    (print-section (grecs-node-up node) #\space)
    ;; Print its identifier, ...
    (display (grecs-node-ident node))
    (if (grecs-node-has-value? node)
        ;; ... value,
        (begin
          (display " ")
          (display (grecs-node-value node))))
    ;; ... and delimiter
    (display delim))
   (else              ;; mark the root node
    (display "["))))  ;;  with a [


(define (cfpeek node)
  "Main entry point.  Calls itself recursively to descend
into subordinate nodes and to iterate over nodes on the
same nesting level (tail recursion)."
  (let loop ((node node))
    (if node
```

```
          (let ((type (grecs-node-type node)))
            (cond
             ((= type grecs-node-root)
              (let ((dn (grecs-node-down node)))
                ;; Each statement in a Git config file must
                ;; belong to a section.  If the first node
                ;; is not a block statement, provide the
                ;; default [core] section:
                (if (not (= (grecs-node-type dn)
                            grecs-node-block))
                    (display "[core]\n"))
                ;; Continue from the first node
                (loop dn)))
             ((= type grecs-node-block)
              ;; print the section header
              (print-section node #\])
              (newline)
              ;; descend into subnodes
              (loop (grecs-node-down node))
              ;; continue from the next node
              (loop (grecs-node-next node)))
             ((= type grecs-node-stmt)
              ;; print the simple statement
              (display #\tab)
              (display (grecs-node-ident node))
              (display " = ")
              (display (grecs-node-value node))
              (newline)
              ;; continue from the next node
              (loop (grecs-node-next node)))))))
```

If run on our sample configuration file, it produces:

```
$ cfpeek -f togit.scm sample.conf .
[core]
        user = smith
        group = mail
        pidfile = /var/run/example
[logging]
        facility = daemon
        tag = example
[program a]
        command = a.out
[program a logging]
        facility = local0
        tag = a
[program b]
```

```
command = b.out
wait = yes
pidfile = /var/run/b.pid
```

4 Supported Configuration File Formats

`Cfpeek` is able to handle input files in several formats. The supported formats differ mostly in syntax. This chapter describes them in detail. If you know of any free software which uses a structured configuration file not understood by `cfpeek`, please let us know (see Chapter 8 [Reporting Bugs], page 37).

4.1 Grecs Configuration File

This is the default input format. It is used, e.g., by GNU Dico[1], GNU Mailutils[2], GNU Radius[3], Mailfromd[4] and others.

The configuration file consists of statements and comments.

There are three classes of lexical tokens: keywords, values, and separators. Blanks, tabs, newlines and comments, collectively called *white space* are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent keywords and values.

4.1.1 Comments

Comments may appear anywhere where white space may appear in the configuration file. There are two kinds of comments: single-line and multi-line comments. *Single-line* comments start with '`#`' or '`//`' and continue to the end of the line:

```
# This is a comment
// This too is a comment
```

Multi-line or *C-style* comments start with the two characters '`/*`' (slash, star) and continue until the first occurrence of '`*/`' (star, slash).

Multi-line comments cannot be nested. However, single-line comments may well appear within multi-line ones.

4.1.2 Pragmatic Comments

Pragmatic comments are similar to usual single-line comments, except that they cause some changes in the way the configuration is parsed. Pragmatic comments begin with a '`#`' sign and end with the next physical newline character.

`#include <`*`file`*`>`
`#include` *`file`*

Include the contents of the file *file*. There are three possible use cases.

---

[1] See *GNU Dico Manual*.
[2] See *GNU Mailutils Manual*.
[3] See *GNU Radius Manual*.
[4] See *Mailfromd Manual*.

If *file* is an absolute file name, the named file is included. An error message will be issued if it does not exist.

If *file* contains wildcard characters ('*', '[', ']' or '?'), it is interpreted as shell globbing pattern and all files matching that pattern are included, in lexicographical order. If no files match the pattern, the statement is silently ignored.

Otherwise, the form with angle brackets searches for file in the *include search path*, while the second one looks for it in the current working directory first, and, if not found there, in the include search path. If the file is not found, an error message will be issued.

The default include search path is:

1. *prefix*/share/*program-name*/1.2/include
2. *prefix*/share/*program-name*/include

where *prefix* is the installation prefix.

#include_once <*file*>
#include_once *file*

Same as #include, except that, if the *file* has already been included, it will not be included again.

#line *num*
#line *num* "*file*"

This line causes the parser to believe, for purposes of error diagnostics, that the line number of the next source line is given by *num* and the current input file is named by *file*. If the latter is absent, the remembered file name does not change.

num "*file*"

This is a special form of #line statement, understood for compatibility with the C preprocessor.

In fact, these statements provide a rudimentary preprocessing features. For more sophisticated ways to modify configuration before parsing, see Section 4.1.4 [Preprocessor], page 20.

4.1.3 Statements

A *simple statement* consists of a keyword and value separated by any amount of whitespace. Simple statement is terminated with a semicolon (';').

The following is a simple statement:

```
standalone yes;
pidfile /var/run/slb.pid;
```

A *keyword* begins with a letter and may contain letters, decimal digits, underscores ('_') and dashes ('-'). Examples of keywords are: 'expression', 'output-file'.

A *value* can be one of the following:

number A number is a sequence of decimal digits.

boolean A boolean value is one of the following: '`yes`', '`true`', '`t`' or '`1`', meaning *true*, and '`no`', '`false`', '`nil`', '`0`' meaning *false*.

unquoted string
An unquoted string may contain letters, digits, and any of the following characters: '`_`', '`-`', '`.`', '`/`', '`@`', '`*`', '`:`'.

quoted string
A quoted string is any sequence of characters enclosed in double-quotes ('`"`'). A backslash appearing within a quoted string introduces an *escape sequence*, which is replaced with a single character according to the following rules:

Sequence	Replaced with
\a	Audible bell character (ASCII 7)
\b	Backspace character (ASCII 8)
\f	Form-feed character (ASCII 12)
\n	Newline character (ASCII 10)
\r	Carriage return character (ASCII 13)
\t	Horizontal tabulation character (ASCII 9)
\v	Vertical tabulation character (ASCII 11)
\\	A single backslash ('\')
\"	A double-quote.

Table 4.1: Backslash escapes

In addition, the sequence '`\`*`newline`*' is removed from the string. This allows to split long strings over several physical lines, e.g.:

```
"a long string may be\
 split over several lines"
```

If the character following a backslash is not one of those specified above, the backslash is ignored and a warning is issued.

Here-document
A *here-document* is a special construct that allows to introduce strings of text containing embedded newlines.

The `<<`*`word`* construct instructs the parser to read all the following lines up to the line containing only *word*, with possible trailing blanks. Any lines thus read are concatenated together into a single string. For example:

```
<<EOT
A multiline
string
EOT
```

The body of a here-document is interpreted the same way as a double-quoted string, unless *word* is preceded by a backslash (e.g. '`<<\EOT`') or enclosed in double-quotes, in which case the text is read as is, without interpretation of escape sequences.

If *word* is prefixed with `-` (a dash), then all leading tab characters are stripped from input lines and the line containing *word*. Furthermore, if `-` is followed by a single space, all leading whitespace is stripped from them. This allows to indent here-documents in a natural fashion. For example:

```
<<- TEXT
    The leading whitespace will be
    ignored when reading these lines.
TEXT
```

It is important that the terminating delimiter be the only token on its line. The only exception to this rule is allowed if a here-document appears as the last element of a statement. In this case a semicolon can be placed on the same line with its terminating delimiter, as in:

```
help-text <<-EOT
        A sample help text.
EOT;
```

list

A *list* is a comma-separated list of values. Lists are enclosed in parentheses. The following example shows a statement whose value is a list of strings:

```
alias (test,null);
```

In any case where a list is appropriate, a single value is allowed without being a member of a list: it is equivalent to a list with a single member. This means that, e.g.

```
alias test;
```

is equivalent to

```
alias (test);
```

A *block statement* introduces a logical group of statements. It consists of a keyword, followed by an optional value, and a sequence of statements enclosed in curly braces, as shown in the example below:

```
server srv1 {
  host 10.0.0.1;
  community "foo";
}
```

The closing curly brace may be followed by a semicolon, although this is not required.

4.1.4 Preprocessor

Before actual parsing, the configuration file is preprocessed. The built-in preprocessor handles only file inclusion and `#line` statements (see Section 4.1.2

[Pragmatic Comments], page 17), while the rest of traditional preprocessing facilities, such as macro expansion, is supported via m4, which serves as external preprocessor.

The detailed description of m4 facilities lies far beyond the scope of this document. You will find a complete user manual in Section "GNU M4" in *GNU M4 macro processor*. For the rest of this subsection we assume the reader is sufficiently acquainted with m4 macro processor.

The external preprocessor is invoked with -s flag, which instructs it to include line synchronization information in its output. This information is then used by the parser to display meaningful diagnostic.

An initial set of macro definitions is supplied by the pp-setup file, located in *prefix*/share/*program-name*/1.2/include directory.

The default pp-setup file renames all m4 built-in macro names so they all start with the prefix 'm4_'. This is similar to GNU m4 --prefix-builtin option, but has an advantage that it works with non-GNU m4 implementations as well.

4.2 Path Configuration File

A *pathname configuration file* format corresponds exactly to the default output format of cfpeek, i.e. it lists each terminal keyword as its full pathname, followed by a semicolon, a single space and its value, as in the example below:

```
.user: "smith"
.group: "mail"
.pidfile: "/var/run/example"
.logging.facility: "daemon"
.logging.tag: "example"
.program="a".command: "a.out"
.program="a".logging.facility: "local0"
.program="a".logging.tag: "a"
.program="b".command: "b.out"
.program="b".wait: "yes"
.program="b".pidfile: "/var/run/b.pid"
```

This format is similar to the one used in X-resources.

4.3 BIND Configuration File

This is the format used by the ISC BIND configuration files. In general, it is pretty similar to the 'Grecs', except that it does not support neither here-documents, not list values. Some of its features, such as 'acls' and 'allow-*' lists do resemble lists, but are not them in reality. Such "suspicious" statements are represented as simple statements. For example, the following statement in named.conf:

```
allow-transfer {
    allow-dns;
```

```
        !10.10.10.1;
        10.10.10.0/8;
    };
    .allow-transfer.allow-dns:
    .allow-transfer.!: "10.10.10.1"
    .allow-transfer."10.10.10.0/8":
```

Another exception is the 'controls' statement, which doesn't fall well into the general syntax of BIND configuration file. Therefore a special rule is applied to handle it. In the effect, the following statement:

```
controls {
    inet 127.0.0.1 port 953
        allow { 127.0.0.1; 127.0.0.2; } keys { "rndc-key"; };
};
```

produces

```
.controls: (inet, 127.0.0.1, port, 953, allow, \
              (127.0.0.1, 127.0.0.2), keys, (rndc-key))
```

4.4 DHCPD Configuration File

This is the format used by the ISC DHCPD configuration files (/etc/dhcpd.conf and any files it might include). It is very similar to 'Bind', with some minor differences:

- Block statements do not end with a semicolon.
- Tags or values can contain lists of quoted strings delimited by commas.

4.5 MeTA1 Configuration File

This type of configuration file is used by *MeTA1*, an advanced MTA program. See http://www.meta1.org for details about the program and its configuration.

The syntax is similar to both 'Grecs' and 'Bind' in that it uses curly braces to delimit subordinate statements. The syntax for strings is similar to 'Grecs' (see Section 4.1.3 [quoted string], page 18). As in 'Grecs', adjacent quoted strings are concatenated to produce a single string.

The principal syntactic differences are:

- Only '#' comments are understood.
- An equal sign is required between identifier and value in simple statements, e.g.:

  ```
  log_level = 12;
  ```
- List values are enclosed in curly braces.
- Here-document is not supported.

4.6 GIT Configuration File

This is the format used by Git (http://git-scm.com). It is described in detail in See Section "CONFIGURATION FILE" in *git-config(1) man page*.

The syntax is line-oriented. Comments are introduced by '#' or ';' character and extend up to the next physical newline. Statements are delimited by newlines.

The syntax for simple statement is:

```
ident = value
```

Compound statements or *sections* begin with a *section header*, i.e. a full pathname of that section using single space as a separator and enclosed in a pair of square brackets. Any identifier in the path which contains whitespace characters must be quoted using double quotes. Double quotes and backslashes appearing in a section name must be escaped as '\"' and '\\' correspondingly. For example:

```
[section "subsection name" subsubsection]
```

An alternative syntax for section headers is a full pathname of the section using single dot as a separator and enclosed in a pair of square brackets. When this syntax is used, whitespace is not allowed in section names:

```
[section.subsection.subsubsection]
```

A section begins with the section headers and continues until the start of next section or end of file, whichever occurs first.

Simple statements must occur only within a section. In other words, each non-empty configuration file must contain at least one section.

String values may be entirely or partially enclosed in double quotes, similarly to shell syntax. The following escape sequences are recognized within a value:

Sequence	**Stands for**
'\"'	'"'
'\\'	'\'
'\b'	Backspace (ASCII 8)
'\t'	Horizontal tab (ASCII 9)
'\n'	Newline (ASCII 10)

A backslash immediately preceding a newline indicates line continuation. Both characters are removed and the remaining characters are joined with line that follows.

5 Cfpeek Command Line Syntax

The format of `cfpeek` invocation is:

```
cfpeek options file [keys]
```

where *options* are command line options, *file* is the configuration file to operate upon, and optional *keys* are pathnames of the keywords to locate in that configuration file.

If *keys* are supplied, `cfpeek`, for each *key*, looks up in the parse tree for any nodes matching the key and prints them on the standard output. An error message is displayed for any key which has no matching statements in the input file. In this case, program continues iterating over the rest of *keys*. When the list is exhausted, `cfpeek` will exit with the status 1 (see Chapter 6 [Exit Codes], page 31).

If either `-f` (`--file`) or `-e` (`--expression`) has been given, a Scheme expression or the default `cfpeek` function is evaluated for each matching node. If `-e` (`--expression`) is given, the node is passed to it in the global '`node`' variable. Otherwise, if `-f` (`--file`) is given, the node is passed as argument to `cfpeek` function.

If both `--file=`*`script`* and `--expression=`*`expression`* options are given, the script file *script* is loaded first, and the *expression* is evaluated for each matching node. The expression can then refer to any variables and call any functions defined in the *script*.

If no keys are supplied, the program operates as if given a single '`.*`' key (see Section 5.1 [Patterns], page 25), which matches any node in the parse tree (i.e., it iterates over the entire parse tree).

5.1 Patterns

By default `cfpeek` treats keys as *wildcard patterns*. When matching statement identifiers (keywords), two characters have special meaning: '`%`' and '`*`'.

A '`%`' character in place of an identifier matches any single keyword. Thus, e.g.:

```
cfpeek file.conf .%.bar.baz
```

will match '`.foo.bar.baz`', '`.qux.bar.baz`', but will not match '`.bar.baz`' or '`.x.y.bar.baz`'.

A single '`*`' character in place of a keyword matches zero or more keywords appearing in its place, so that:

```
cfpeek file.conf .*.bar.baz
```

The tags in block statement are matched using the traditional globbing patterns. See Section "match filename or pathname" in *fnmatch(3) man page*.

For example, this:

```
cfpeek file.conf .*.program="mh-*"
```

will match any '`program`' block statement whose tag begins with '`mh-`'.

5.2 Output Control

`-H` *`flags`*
`--format=`*`flags`*
: Set output format flags. The argument is a comma-separated list of format flags and *relative movement* options. Relative movement options select another node, relative to the one found. They are:

  '`parent=`*`id`*'
  : Find a parent of the matching node, which has *id* as its identifier.

  '`child=`*`id`*'
  : Find a child of the matching node, which has *id* as its identifier.

  '`child=`*`id`*'
  : Find a sibling of the matching node, which has *id* as its identifier.

  '`up=`*`n`*'
  : Ascend *n* parent nodes and print the node at which the ascent stopped.

`descend=`*`n`*
: Descend *n* child nodes.

  Any number of relative movement options can be specified. They are executed in the order of their appearance in the `--format` statement. For example, `--format=up=2,sibling=foo,child=bar` means: ascend two levels of hierarchy, find a node named '`foo`', look for a node named '`bar`' among the children of that node and print the result.

  If evaluation of the relative movement options results in an empty node (e.g. the '`up`' option attempts to go past the root of the tree), nothing is output.

  The `delim` flag controls how keyword paths is printed:

  '`delim=`*`char`*'
  : Sets path component delimiter, instead of the default '`.`'.

  The following flags control the amount of information printed for each node. These are boolean flags: when prefixed with '`no`' they have the meaning opposite to the described.

'locus' Print source location of each configuration statement. A location is printed as the file name, followed by a semicolon, followed by the line number and another semicolon. Locations are separated from the rest of output by a single space character.

'path' Print statement paths.

'value' Print statement values.

'quote' Always quote string values.

'never-quote'
Never quote string values.

'quote-hex'
Print non-printable characters as C hex escapes. This option is ignored if 'noquote' is set.

'descend' Descend into subnodes. Set default options.

The default format options are: 'path,value,quote,descend'.

-q
--quiet Suppress error diagnostics. See Section 3.5 [quiet], page 8.

5.3 Modifiers

The following options modify the way cfpeek processes the parse tree and search keys.

-L
--literal
Use literal matching, instead of pattern matching. See [literal], page 10.

-S
--sort Before further processing, sort parse tree lexicographically in ascending order.

-m
--matches=*number*
Output at most *number* matches for each key.

-p
--parser=*type*
Set parser type for the input file. The argument is one of: 'grecs', 'path', 'meta1', 'bind', 'dhcpd', and 'git' (case-insensitive). See Chapter 4 [Formats], page 17, for a description of each type.

-r
--reduce Reduce the parse tree, so that each keyword occurs no more than once at each tree level.

`-s path=val`
`--set=path=val`
: Set a keyword *path* to *value*. The produced parse tree node will be processed as usual.

5.4 Scripting Options

The following options control the scripting facility of `cfpeek`.

`-e expression`
`--expression=expression`
: Apply this expression to each node found. The global variable `node` is set to the node being processed before evaluating. When used together with `--file=script`, the expression can refer to any variables and call any functions defined in the *script* file.

`-f file`
`--file=file`
: Load the script *file*. Unless `--expression` is also given, the script must define the function named '`cfpeek`' which takes a node as its only argument. This function will be called for each matching node.

  If `--expression` is given, this behavior is suppressed. It is then the responsibility of the expression to call any functions defined in this file.

`-i expr`
`--init=expr`
: The `--init=expr` (`-i expr`) option provides an initialization expression *expr*. This expression is evaluated once, after loading the script file, if one is specified, and before starting the main loop.

`-l script-language`
`--lang=script-language`
: Select scripting language to use. This option is reserved for further use. As of version 1.2, the only possible value for *script-language* is '`scheme`'.

5.5 Preprocessor Control Options

The options described below control the preprocessor facility. They are meaningful only for '`GRECS`' and '`BIND`' configuration files. Preprocessor is not used for another configuration file formats.

`-Dname[=value]`
`--define=name[=value]`
: Define the preprocessor symbol *name* as having *value*, or empty. See Section 4.1.4 [Preprocessor], page 20.

`-I dir`
`--include-directory=dir`
Add *dir* to include search path.

See Section 4.1.2 [Pragmatic Comments], page 17.

`-N`
`--no-preprocessor`
Disable preprocessor. see Section 4.1.4 [Preprocessor], page 20.

`-P command`
`--preprocessor=command`
Use *command* instead of the default preprocessor. see Section 4.1.4 [Preprocessor], page 20.

5.6 Debugging Options

The options below enable trace output which helps understand how configuration parser works. They are mainly useful for `cfpeek` developers.

`-X`
`--debug-lexer`
Trace configuration file lexer.

`-x`
`--debug-parser`
Trace configuration file parser.

5.7 Informational Options

`--help`
`-h` Print a concise usage summary and exit.

`--usage` Print a summary of command line syntax and exit.

`--version`
`-v` Print the program version and exit.

6 Exit Codes

When cfpeek terminates, it reports the result of its invocation via its exit code. Exit code of 0 indicates normal termination. Exit code 1 indicates that not all search keys has been found. Exit codes greater than 1 indicate various error conditions. The exact cause of failure is reported on the standard error.

The exit codes are as follows:

2 Error parsing the input file.

3 Script failure.

64 The command was used incorrectly, e.g., with the wrong number of arguments, a bad option, a bad syntax in a parameter, or whatever.

69 The requested script file does not exist, contains syntax errors, or cannot be parsed for whatever other reason.

70 An internal software error has occurred. Please, report it, along with any error diagnostics produced by the program, if you ever stumble upon this error code. See Chapter 8 [Reporting Bugs], page 37, for detailed instructions.

78 The script file parses correctly, but does not define all the symbols required by `cfpeek`.

7 Scripting

This chapter describes the Scheme functions available for use in `cfpeek` scripts. For an introduction to `cfpeek` scripting facility, see Section 3.9 [Scripts], page 11.

`grecs-node?` *obj* [Scheme Procedure]
Returns '`#t`' if *obj* is a valid tree node.

`grecs-node-root` *node* [Scheme Procedure]
Returns the topmost node that can be traced up from *node*.

`grecs-node-head` *node* [Scheme Procedure]
Returns the first node having the same parent and located on the same nesting level as *node*. I.e. the following always holds true:

```
(let ((head (grecs-node-head node)))
  (and
    (eq? (grecs-node-up node) (grecs-node-up head))
    (not (grecs-node-prev? head))))
```

`grecs-node-tail` *node* [Scheme Procedure]
Returns the last node having the same parent and located on the same nesting level as node. In other words, the following relation is always '`#t`':

```
(let ((tail (grecs-node-tail node)))
  (and
    (eq? (grecs-node-up node) (grecs-node-up tail))
         (not (grecs-node-next? tail))))
```

`grecs-node-up?` *node* [Scheme Procedure]
Return true if *node* has a parent node.

`grecs-node-up` *node* [Scheme Procedure]
Return parent node of *node*.

`grecs-node-down?` *node* [Scheme Procedure]
Returns '`#t`' if *node* has child nodes.

`grecs-node-down` *node* [Scheme Procedure]
Returns the first child node of *node*.

`grecs-node-next?` *node* [Scheme Procedure]
Returns '`#t`' if *node* is followed by another node on the same nesting level.

`grecs-node-next` *node* [Scheme Procedure]
Returns the node following *node* on the same nesting level.

`grecs-node-prev?` *node* [Scheme Procedure]
Returns '`#t`' if *node* is preceded by another node on the same nesting level.

`grecs-node-prev` *node* [Scheme Procedure]
Returns the node preceding *node* on the same nesting level.

`grecs-node-ident` *node* [Scheme Procedure]
Returns identifier of the node *node*.

`grecs-node-ident-locus` *node* [*full*] [Scheme Procedure]
Returns locus of the *node*'s identifier. Returned value is a cons whose parts depend on *full*, which is a boolean value. If *full* is '`#f`', which is the default, then returned value is a cons:

```
(file-name . line-number)
```

Oherwise, if *full* is '`#t`', the function returns the locations where the node begins and ends:

```
((beg-file-name beg-line beg-column) .
 (end-file-name end-line end-column))
```

`grecs-node-path-list` *node* [Scheme Procedure]
Returns the full path to the node, converted to a list. Each list element corresponds to a subnode identifier. A subnode which has a tag is represented by a cons, whose car contains the subnode identifier, and cdr its value. For example, the following path:

```
.foo.bar=x.baz
```

is represented as

```
'("foo" ("bar" . "x") "baz")
```

`grecs-node-path` *node* [*delim*] [Scheme Procedure]
Returns the full path to the *node* (a string).

`grecs-node-type` *node* [Scheme Procedure]
Returns the type of the node. The following constants are defined:

grecs-node-root
: The node is a root node. The following is always '`#t`':

```
(and (= (grecs-node-type node) grecs-node-root)
     (not (grecs-node-up? node))
     (not (grecs-node-prev? node)))
```

grecs-node-stmt
: The node is a simple statement. The following is always '`#t`':

```
(and (= (grecs-node-type node) grecs-node-stmt)
     (not (grecs-node-down? node)))
```

grecs-node-block
: The node is a block statement.

`grecs-node-has-value?` *node* [Scheme Procedure]
Returns '`#t`' if *node* has a value.

`grecs-node-value` *node* [Scheme Procedure]
Returns the value of *node*.

`grecs-node-value-locus` *node* [*full*] [Scheme Procedure]
Returns locus of the *node*'s value. Returned value is a cons whose parts depend on *full*, which is a boolean value. If *full* is '`#f`', which is the default, then returned value is a cons:

```
(file-name . line-number)
```

Oherwise, if *full* is '`#t`', the function returns the locations where the node begins and ends:

```
((beg-file-name beg-line beg-column) .
 (end-file-name end-line end-column))
```

`grecs-node-locus` *node* [*full*] [Scheme Procedure]
Returns source location of the *node*. Returned value is a cons whose parts depend on *full*, which is a boolean value. If *full* is '`#f`', which is the default, then returned value is a cons:

```
(file-name . line-number)
```

Oherwise, if *full* is '`#t`', the function returns the locations where the node begins and ends:

```
((beg-file-name beg-line beg-column) .
 (end-file-name end-line end-column))
```

`grecs-find-node` *node path* [Scheme Procedure]
Returns the first node whose path is *path*. Starts search from *node*.

`grecs-match-first` *node pattern* [Scheme Procedure]
Returns the first node whose path matches *pattern*. The search is started from *node*.

`grecs-match-next` *node* [Scheme Procedure]
Node must be a node returned by a previous call to `grecs-match-first` or '`grecs-match-next`'. The function returns next node matching the initial pattern, or '`#f`' if no more matches are found. For example, the following code iterates over all nodes matching *pattern*:

```
(define (iterate-nodes root pattern thunk)
  (do ((node (grecs-match-first root pattern)
             (grecs-match-next node)))
      ((not node))
     (thunk node)))
```

8 How to Report a Bug

Please, report bugs and suggestions to bug-cfpeek@gnu.org.ua.

You hit a bug if at least one of the conditions below is met:

- `cfpeek` terminates on signal 11 (SIGSEGV) or 6 (SIGABRT).
- `cfpeek` terminates with exit code 70 (internal software error).
- The program fails to do its job as described in this manual.

If you think you've found a bug, please be sure to include maximum information available to reliably reproduce it, or at least to analyze it. The information needed is:

- Version of the package you are using.
- Command line options and input file (or files) used.
- Conditions under which the bug appears.

Any errors, typos or omissions found in this manual also qualify as bugs. Please report them, if you happen to find any.

Appendix A GNU Free Documentation License

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:

A. 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.

B. 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.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements 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 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.

A.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

This is a general index of all issues discussed in this manual

#

%

*

-

A

B

C

D

E

F

G

H

I

K

L

M

N

P