C

Implementation Notes

This appendix contains information mainly of interest to implementors and maintainers of *gawk*. Everything in it applies specifically to *gawk* and not to other implementations.

Downward Compatibility and Debugging

See the section "Extensions in gawk Not in POSIX awk" in Appendix A, *The Evolution of the awk Language*, for a summary of the GNU extensions to the *awk* language and program. All of these features can be turned off by invoking *gawk* with the *––traditional* option or with the *––posix* option.

If *gawk* is compiled for debugging with `-DDEBUG`, then there is one more option available on the command line:

`-W parsedebug`
`--parsedebug`
Prints out the parse stack information as the program is being parsed.

This option is intended only for serious *gawk* developers and not for the casual user. It probably has not even been compiled into your version of *gawk*, since it slows down execution.

Making Additions to gawk

If you find that you want to enhance *gawk* in a significant fashion, you are perfectly free to do so. That is the point of having free software; the source code is available and you are free to change it as you want (see Appendix E, *GNU General Public License*).

This section discusses the ways you might want to change *gawk* as well as any considerations you should bear in mind.

Adding New Features

You are free to add any new features you like to *gawk*. However, if you want your changes to be incorporated into the *gawk* distribution, there are several steps that you need to take in order to make it possible for me to include your changes:

1. Before building the new feature into *gawk* itself, consider writing it as an extension module (see the section "Adding New Built-in Functions to gawk" later in this appendix). If that's not possible, continue with the rest of the steps in this list.
2. Get the latest version. It is much easier for me to integrate changes if they are relative to the most recent distributed version of *gawk*. If your version of *gawk* is very old, I may not be able to integrate them at all. (See the section "Getting the gawk Distribution" in Appendix B, *Installing gawk*, for information on getting the latest version of *gawk*.)
3. Follow the *GNU Coding Standards*. This document describes how GNU software should be written. If you haven't read it, please do so, preferably *before* starting to modify *gawk*. (The *GNU Coding Standards* are available from the GNU Project's FTP site, at *ftp://gnudist.gnu.org/gnu/GNUInfo/standards.text*. Texinfo, Info, and DVI versions are also available.)
4. Use the *gawk* coding style. The C code for *gawk* follows the instructions in the *GNU Coding Standards*, with minor exceptions. The code is formatted using the traditional "K&R" style, particularly as regards to the placement of braces and the use of tabs. In brief, the coding rules for *gawk* are as follows:
   - Use ANSI/ISO style (prototype) function headers when defining functions.
   - Put the name of the function at the beginning of its own line.
   - Put the return type of the function, even if it is `int`, on the line above the line with the name and arguments of the function.
   - Put spaces around parentheses used in control structures (`if`, `while`, `for`, `do`, `switch`, and `return`).

- Do not put spaces in front of parentheses used in function calls.
- Put spaces around all C operators and after commas in function calls.
- Do not use the comma operator to produce multiple side effects, except in `for` loop initialization and increment parts, and in macro bodies.
- Use real tabs for indenting, not spaces.
- Use the "K&R" brace layout style.
- Use comparisons against `NULL` and `'\0'` in the conditions of `if`, `while`, and `for` statements, as well as in the `case`s of `switch` statements, instead of just the plain pointer or character value.
- Use the `TRUE`, `FALSE` and `NULL` symbolic constants and the character constant `'\0'` where appropriate, instead of `1` and `0`.
- Use the `ISALPHA`, `ISDIGIT`, etc. macros, instead of the traditional lowercase versions; these macros are better behaved for non-ASCII character sets.
- Provide one-line descriptive comments for each function.
- Do not use `#elif`. Many older Unix C compilers cannot handle it.
- Do not use the `alloca` function for allocating memory off the stack. Its use causes more portability trouble than is worth the minor benefit of not having to free the storage. Instead, use `malloc` and `free`.

If I have to reformat your code to follow the coding style used in *gawk*, I may not bother to integrate your changes at all.

5. Be prepared to sign the appropriate paperwork. In order for the FSF to distribute your changes, you must either place those changes in the public domain and submit a signed statement to that effect, or assign the copyright in your changes to the FSF. Both of these actions are easy to do and *many* people have done so already. If you have questions, please contact me (see the section "Reporting Problems and Bugs" in Appendix B), or *gnu@gnu.org*.
6. Update the documentation. Along with your new code, please supply new sections and/or chapters for this book. If at all possible, please use real Texinfo, instead of just supplying unformatted ASCII text (although even that is better than no documentation at all). Conventions to be followed in *Effective awk Programming* are provided after the `@bye` at the end of the Texinfo source file. If possible, please update the manpage as well.

   You will also have to sign paperwork for your documentation changes.

7. Submit changes as context diffs or unified diffs. Use `diff -c -r -N` or `diff -u -r -N` to compare the original *gawk* source tree with your version. (I find context diffs to be more readable but unified diffs are more compact.) I recommend using the GNU version of *diff*. Send the output produced by either run of *diff* to me when you submit your changes. (See the section "Reporting Problems and Bugs" in Appendix B, for the electronic mail information.)

   Using this format makes it easy for me to apply your changes to the master version of the *gawk* source code (using `patch`). If I have to apply the changes manually, using a text editor, I may not do so, particularly if there are lots of changes.

8. Include an entry for the *ChangeLog* file with your submission. This helps further minimize the amount of work I have to do, making it easier for me to accept patches.

Although this sounds like a lot of work, please remember that while you may write the new code, I have to maintain it and support it. If it isn't possible for me to do that with a minimum of extra work, then I probably will not.

Porting gawk to a New Operating System

If you want to port *gawk* to a new operating system, there are several steps:

1. Follow the guidelines in the previous section concerning coding style, submission of diffs, and so on.

2. When doing a port, bear in mind that your code must coexist peacefully with the rest of *gawk* and the other ports. Avoid gratuitous changes to the system-independent parts of the code. If at all possible, avoid sprinkling `#ifdef`s just for your port throughout the code.

   If the changes needed for a particular system affect too much of the code, I probably will not accept them. In such a case, you can, of course, distribute your changes on your own, as long as you comply with the GPL (see Appendix E).

3. A number of the files that come with *gawk* are maintained by other people at the Free Software Foundation. Thus, you should not change them unless it is for a very good reason; i.e., changes are not out of the question, but changes to these files are scrutinized extra carefully. The files are *getopt.h*, *getopt.c*, *getopt1.c*, *regex.h*, *regex.c*, *dfa.h*, *dfa.c*, *install-sh*, and *mkinstalldirs*.

4. Be willing to continue to maintain the port. Non-Unix operating systems are supported by volunteers who maintain the code needed to compile and run *gawk* on their systems. If noone volunteers to maintain a port, it becomes unsupported and it may be necessary to remove it from the distribution.

5. Supply an appropriate *gawkmisc.???* file. Each port has its own *gawkmisc.???* that implements certain operating system specific functions. This is cleaner than a plethora of `#ifdef`s scattered throughout the code. The *gawkmisc.c* in the main source directory includes the appropriate *gawkmisc.???* file from each subdirectory. Be sure to update it as well. Each port's *gawkmisc.???* file has a suffix reminiscent of the machine or operating system for the port—for example, *pc/gawkmisc.pc* and *vms/gawkmisc.vms*. The use of separate suffixes, instead of plain *gawkmisc.c*, makes it possible to move files from a port's subdirectory into the main subdirectory, without accidentally destroying the real *gawkmisc.c* file. (Currently, this is only an issue for the PC operating system ports.)
6. Supply a *Makefile* as well as any other C source and header files that are necessary for your operating system. All your code should be in a separate subdirectory, with a name that is the same as, or reminiscent of, either your operating system or the computer system. If possible, try to structure things so that it is not necessary to move files out of the subdirectory into the main source directory. If that is not possible, then be sure to avoid using names for your files that duplicate the names of files in the main source directory.
7. Update the documentation. Please write a section (or sections) for this book describing the installation and compilation steps needed to compile and/or install *gawk* for your system.
8. Be prepared to sign the appropriate paperwork. In order for the FSF to distribute your code, you must either place your code in the public domain and submit a signed statement to that effect, or assign the copyright in your code to the FSF.

Following these steps makes it much easier to integrate your changes into *gawk* and have them coexist happily with other operating systems' code that is already there.

In the code that you supply and maintain, feel free to use a coding style and brace layout that suits your taste.

Adding New Built-in Functions to gawk

Beginning with *gawk* 3.1, it is possible to add new built-in functions to *gawk* using dynamically loaded libraries. This facility is available on systems (such as GNU/Linux) that support the `dlopen` and `dlsym` functions. This section describes how to write and use dynamically loaded extentions for *gawk*. Experience with programming in C or C++ is necessary when reading this section.

The facilities described in this section are very much subject to change in the next *gawk* release. Be aware that you may have to redo everything, perhaps from scratch, upon the next release.

A Minimal Introduction to gawk Internals

The truth is that *gawk* was not designed for simple extensibility. The facilities for adding functions using shared libraries work, but are something of a "bag on the side." Thus, this tour is brief and simplistic; would-be *gawk* hackers are encouraged to spend some time reading the source code before trying to write extensions based on the material presented here. Of particular note are the files *awk.h*, *builtin.c*, and *eval.c*. Reading *awk.y* in order to see how the parse tree is built would also be of use.

With the disclaimers out of the way, the following types, structure members, functions, and macros are declared in *awk.h* and are of use when writing extensions. The next section shows how they are used:

`AWKNUM`
: An `AWKNUM` is the internal type of *awk* floating-point numbers. Typically, it is a C `double`.

`NODE`
: Just about everything is done using objects of type `NODE`. These contain both strings and numbers, as well as variables and arrays.

`AWKNUM force_number(NODE *n)`
: This macro forces a value to be numeric. It returns the actual numeric value contained in the node. It may end up calling an internal *gawk* function.

`void force_string(NODE *n)`
: This macro guarantees that a `NODE`'s string value is current. It may end up calling an internal *gawk* function. It also guarantees that the string is zero-terminated.

`n->param_cnt`
: The number of parameters actually passed in a function call at runtime.

`n->stptr`
`n->stlen`
: The data and length of a `NODE`'s string value, respectively. The string is *not* guaranteed to be zero-terminated. If you need to pass the string value to a C library function, save the value in `n->stptr[n->stlen]`, assign `'\0'` to it, call the routine, and then restore the value.

`n->type`
: The type of the `NODE`. This is a C `enum`. Values should be either `Node_var` or `Node_var_array` for function parameters.

`n->vname`
: The "variable name" of a node. This is not of much use inside externally written extensions.

`void assoc_clear(NODE *n)`
: Clears the associative array pointed to by `n`. Make sure that `n->type == Node_var_array` first.

`NODE **assoc_lookup(NODE *symbol, NODE *subs, int reference)`
: Finds, and installs if necessary, array elements. `symbol` is the array, `subs` is the subscript. This is usually a value created with `tmp_string` (see below). `reference` should be `TRUE` if it is an error to use the value before it is created. Typically, `FALSE` is the correct value to use from extension functions.

`NODE *make_string(char *s, size_t len)`
: Take a C string and turn it into a pointer to a `NODE` that can be stored appropriately. This is permanent storage; understanding of *gawk* memory management is helpful.

`NODE *make_number(AWKNUM val)`
: Take an `AWKNUM` and turn it into a pointer to a `NODE` that can be stored appropriately. This is permanent storage; understanding of *gawk* memory management is helpful.

`NODE *tmp_string(char *s, size_t len);`
: Take a C string and turn it into a pointer to a `NODE` that can be stored appropriately. This is temporary storage; understanding of *gawk* memory management is helpful.

`NODE *tmp_number(AWKNUM val)`
: Take an `AWKNUM` and turn it into a pointer to a `NODE` that can be stored appropriately. This is temporary storage; understanding of *gawk* memory management is helpful.

`NODE *dupnode(NODE *n)`
: Duplicate a node. In most cases, this increments an internal reference count instead of actually duplicating the entire `NODE`; understanding of *gawk* memory management is helpful.

`void free_temp(NODE *n)`
: This macro releases the memory associated with a `NODE` allocated with `tmp_string` or `tmp_number`. Understanding of *gawk* memory management is helpful.

`void make_builtin(char *name, NODE *(*func)(NODE *), int count)`
: Register a C function pointed to by `func` as new built-in function `name`. `name` is a regular C string. `count` is the maximum number of arguments that the function takes. The function should be written in the following manner:

```
/* do_xxx --- do xxx function for gawk */

NODE *
do_xxx(NODE *tree)
{
    ...
}
```

`NODE *get_argument(NODE *tree, int i)`
: This function is called from within a C extension function to get the `i`-th argument from the function call. The first argument is argument zero.

`void set_value(NODE *tree)`
: This function is called from within a C extension function to set the return value from the extension function. This value is what the *awk* program sees as the return value from the new *awk* function.

`void update_ERRNO(void)`
: This function is called from within a C extension function to set the value of *gawk*'s `ERRNO` variable, based on the current value of the C `errno` variable. It is provided as a convenience.

An argument that is supposed to be an array needs to be handled with some extra code, in case the array being passed in is actually from a function parameter. The following boilerplate code shows how to do this:

```
NODE *the_arg;

the_arg = get_argument(tree, 2); /* assume need 3rd arg, 0-based */

/* if a parameter, get it off the stack */
if (the_arg->type == Node_param_list)
    the_arg = stack_ptr[the_arg->param_cnt];

/* parameter referenced an array, get it */
if (the_arg->type == Node_array_ref)
    the_arg = the_arg->orig_array;

/* check type */
if (the_arg->type != Node_var && the_arg->type != Node_var_array)
    fatal("newfunc: third argument is not an array");

/* force it to be an array, if necessary, clear it */
the_arg->type = Node_var_array;
assoc_clear(the_arg);
```

Again, you should spend time studying the *gawk* internals; don't just blindly copy this code.

Directory and File Operation Built-ins

Two useful functions that are not in *awk* are `chdir` (so that an *awk* program can change its directory) and `stat` (so that an *awk* program can gather information about a file). This section implements these functions for *gawk* in an external extension library.

Using chdir and stat

This section shows how to use the new functions at the *awk* level once they've been integrated into the running *gawk* interpreter. Using `chdir` is very straightforward. It takes one argument, the new directory to change to:

```
...
newdir = "/home/arnold/funstuff"
ret = chdir(newdir)
if (ret < 0) {
    printf("could not change to %s: %s\n",
                    newdir, ERRNO) > "/dev/stderr"
    exit 1
}
...
```

The return value is negative if the `chdir` failed, and `ERRNO` (see the section "Built-in Variables" in Chapter 6, *Patterns, Actions, and Variables*) is set to a string indicating the error.

Using `stat` is a bit more complicated. The C `stat` function fills in a structure that has a fair amount of information. The right way to model this in *awk* is to fill in an associative array with the appropriate information:

```
file = "/home/arnold/.profile"
fdata[1] = "x"    # force 'fdata' to be an array
ret = stat(file, fdata)
if (ret < 0) {
    printf("could not stat %s: %s\n", file, ERRNO) > "/dev/stderr"
    exit 1
}
printf("size of %s is %d bytes\n", file, fdata["size"])
```

The `stat` function always clears the data array, even if the `stat` fails. It fills in the following elements:

`"name"`
: The name of the file that was `stat`'ed.

`"dev"`
`"ino"`
: The file's device and inode numbers, respectively.

`"mode"`
: The file's mode, as a numeric value. This includes both the file's type and its permissions.

`"nlink"`
: The number of hard links (directory entries) the file has.

`"uid"`
`"gid"`
: The numeric user and group ID numbers of the file's owner.

`"size"`
: The size in bytes of the file.

`"blocks"`
: The number of disk blocks the file actually occupies. This may not be a function of the file's size if the file has holes.

`"atime"`, `"mtime"`, `"ctime"`
: The file's last access, modification, and inode update times, respectively. These are numeric timestamps, suitable for formatting with `strftime` (see the section "Built-in Functions" in Chapter 8, *Functions*).

`"pmode"`
: The file's "printable mode." This is a string representation of the file's type and permissions, such as what is produced by `ls -l`—for example, `"drwxr-xr-x"`.

`"type"`
: A printable string representation of the file's type. The value is one of the following:

  `"blockdev"`
  `"chardev"`
  : The file is a block or character device ("special file").

  `"directory"`
  : The file is a directory.

  `"fifo"`
  : The file is a named-pipe (also known as a FIFO).

`"file"`
: The file is just a regular file.

`"socket"`
: The file is an `AF_UNIX` ("Unix domain") socket in the filesystem.

`"symlink"`
: The file is a symbolic link.

Several additional elements may be present depending upon the operating system and the type of the file. You can test for them in your *awk* program by using the `in` operator (see the section "Referring to an Array Element" in Chapter 7, *Arrays in awk*):

`"blksize"`
: The preferred block size for I/O to the file. This field is not present on all POSIX-like systems in the C `stat` structure.

`"linkval"`
: If the file is a symbolic link, this element is the name of the file the link points to (i.e., the value of the link).

`"rdev"`, `"major"`, `"minor"`
: If the file is a block or character device file, then these values represent the numeric device number and the major and minor components of that number, respectively.

C code for chdir and stat

Here is the C code for these extensions. They were written for GNU/Linux. The code needs some more work for complete portability to other POSIX-compliant systems:*

```
#include "awk.h"

#include <sys/sysmacros.h>

/*  do_chdir --- provide dynamically loaded chdir() builtin for gawk */

static NODE *
do_chdir(tree)
NODE *tree;
{
    NODE *newdir;
    int ret = -1;

    newdir = get_argument(tree, 0);
```

* This version is edited slightly for presentation. The complete version can be found in *extension/filefuncs.c* in the *gawk* distribution.

The file includes the `"awk.h"` header file for definitions for the *gawk* internals. It includes `<sys/sysmacros.h>` for access to the `major` and `minor` macros.

By convention, for an *awk* function `foo`, the function that implements it is called `do_foo`. The function should take a `NODE *` argument, usually called `tree`, that represents the argument list to the function. The `newdir` variable represents the new directory to change to, retrieved with `get_argument`. Note that the first argument is numbered zero.

This code actually accomplishes the `chdir`. It first forces the argument to be a string and passes the string value to the `chdir` system call. If the `chdir` fails, `ERRNO` is updated. The result of `force_string` has to be freed with `free_temp`:

```
    if (newdir != NULL) {
        (void) force_string(newdir);
        ret = chdir(newdir->stptr);
        if (ret < 0)
            update_ERRNO();

        free_temp(newdir);
    }
```

Finally, the function returns the return value to the *awk* level, using `set_value`. Then it must return a value from the call to the new built-in (this value ignored by the interpreter):

```
    /* Set the return value */
    set_value(tmp_number((AWKNUM) ret));

    /* Just to make the interpreter happy */
    return tmp_number((AWKNUM) 0);
}
```

The `stat` built-in is more involved. First comes a function that turns a numeric mode into a printable representation (e.g., 644 becomes `-rw-r--r--`). This is omitted here for brevity:

```
/* format_mode --- turn a stat mode field into something readable */

static char *
format_mode(fmode)
unsigned long fmode;
{
    ...
}
```

Next comes the actual `do_stat` function itself. First come the variable declarations and argument checking:

```
/* do_stat --- provide a stat() function for gawk */

static NODE *
do_stat(tree)
NODE *tree;
{
    NODE *file, *array;
    struct stat sbuf;
    int ret;
    char *msg;
    NODE **aptr;
    char *pmode;    /* printable mode */
    char *type = "unknown";

    /* check arg count */
    if (tree->param_cnt != 2)
        fatal(
    "stat: called with incorrect number of arguments (%d), should be 2",
            tree->param_cnt);
```

Then comes the actual work. First, we get the arguments. Then, we always clear the array. To get the file information, we use `lstat`, in case the file is a symbolic link. If there's an error, we set `ERRNO` and return:

```
    /* directory is first arg, array to hold results is second */
    file = get_argument(tree, 0);
    array = get_argument(tree, 1);

    /* empty out the array */
    assoc_clear(array);

    /* lstat the file, if error, set ERRNO and return */
    (void) force_string(file);
    ret = lstat(file->stptr, & sbuf);
    if (ret < 0) {
        update_ERRNO();

        set_value(tmp_number((AWKNUM) ret));

        free_temp(file);
        return tmp_number((AWKNUM) 0);
    }
```

Now comes the tedious part: filling in the array. Only a few of the calls are shown here, since they all follow the same pattern:

```
    /* fill in the array */
    aptr = assoc_lookup(array, tmp_string("name", 4), FALSE);
    *aptr = dupnode(file);

    aptr = assoc_lookup(array, tmp_string("mode", 4), FALSE);
    *aptr = make_number((AWKNUM) sbuf.st_mode);

    aptr = assoc_lookup(array, tmp_string("pmode", 5), FALSE);
    pmode = format_mode(sbuf.st_mode);
    *aptr = make_string(pmode, strlen(pmode));
```

When done, we free the temporary value containing the filename, set the return value, and return:

```
    free_temp(file);

    /* Set the return value */
    set_value(tmp_number((AWKNUM) ret));

    /* Just to make the interpreter happy */
    return tmp_number((AWKNUM) 0);
}
```

Finally, it's necessary to provide the "glue" that loads the new function(s) into *gawk*. By convention, each library has a routine named `dlload` that does the job:

```
/* dlload --- load new builtins in this library */

NODE *
dlload(tree, dl)
NODE *tree;
void *dl;
{
    make_builtin("chdir", do_chdir, 1);
    make_builtin("stat", do_stat, 2);
    return tmp_number((AWKNUM) 0);
}
```

And that's it! As an exercise, consider adding functions to implement system calls such as `chown`, `chmod`, and `umask`.

Integrating the extensions

Now that the code is written, it must be possible to add it at runtime to the running *gawk* interpreter. First, the code must be compiled. Assuming that the functions are in a file named *filefuncs.c*, and *idir* is the location of the *gawk* include files, the following steps create a GNU/Linux shared library:

```
$ gcc -shared -DHAVE_CONFIG_H -c -O -g -Iidir filefuncs.c
$ ld -o filefuncs.so -shared filefuncs.o
```

Once the library exists, it is loaded by calling the `extension` built-in function. This function takes two arguments: the name of the library to load and the name of a function to call when the library is first loaded. This function adds the new functions to *gawk*. It returns the value returned by the initialization function within the shared library:

```
# file testff.awk
BEGIN {
    extension("./filefuncs.so", "dlload")

    chdir(".")  # no-op

    data[1] = 1 # force 'data' to be an array
    print "Info for testff.awk"
    ret = stat("testff.awk", data)
    print "ret =", ret
    for (i in data)
        printf "data[\"%s\"] = %s\n", i, data[i]
    print "testff.awk modified:",
        strftime("%m %d %y %H:%M:%S", data["mtime"])
}
```

Here are the results of running the program:

```
$ gawk -f testff.awk
Info for testff.awk
ret = 0
data["blksize"] = 4096
data["mtime"] = 932361936
data["mode"] = 33188
data["type"] = file
data["dev"] = 2065
data["gid"] = 10
data["ino"] = 878597
data["ctime"] = 971431797
data["blocks"] = 2
data["nlink"] = 1
data["name"] = testff.awk
data["atime"] = 971608519
data["pmode"] = -rw-r--r--
data["size"] = 607
data["uid"] = 2076
testff.awk modified: 07 19 99 08:25:36
```

Probable Future Extensions

This section briefly lists extensions and possible improvements that indicate the directions we are currently considering for *gawk*. The file *FUTURES* in the *gawk* distribution lists these extensions as well.

Following is a list of probable future changes visible at the *awk* language level:

Loadable module interface
: It is not clear that the *awk*-level interface to the modules facility is as good as it should be. The interface needs to be redesigned, particularly taking namespace issues into account, as well as possibly including issues such as library search path order and versioning.

`RECLEN` *variable for fixed-length records*
: Along with `FIELDWIDTHS`, this would speed up the processing of fixed-length records. `PROCINFO["RS"]` would be `"RS"` or `"RECLEN"`, depending upon which kind of record processing is in effect.

Additional `printf` *specifiers*
: The 1999 ISO C standard added a number of additional `printf` format specifiers. These should be evaluated for possible inclusion in *gawk*.

Databases
: It may be possible to map a GDBM/NDBM/SDBM file into an *awk* array.

Large character sets
: It would be nice if *gawk* could handle UTF-8 and other character sets that are larger than eight bits.

More `lint` *warnings*
: There are more things that could be checked for portability.

Following is a list of probable improvements that will make *gawk*'s source code easier to work with:

Loadable module mechanics
: The current extension mechanism works (see the earlier section "Adding New Built-in Functions to gawk)", but is rather primitive. It requires a fair amount of manual work to create and integrate a loadable module. Nor is the current mechanism as portable as might be desired. The GNU *libtool* package provides a number of features that would make using loadable modules much easier. *gawk* should be changed to use *libtool*.

Loadable module internals
: The API to its internals that *gawk* "exports" should be revised. Too many things are needlessly exposed. A new API should be designed and implemented to make module writing easier.

Better array subscript management
: *gawk*'s management of array subscript storage could use revamping, so that using the same value to index multiple arrays only stores one copy of the index value.

Integrating the DBUG library
: Integrating Fred Fish's DBUG library would be helpful during development, but it's a lot of work to do.

Following is a list of probable improvements that will make *gawk* perform better:

An improved version of `dfa`
: The `dfa` pattern matcher from GNU *grep* has some problems. Either a new version or a fixed one will deal with some important regexp matching issues.

Compilation of awk programs
: *gawk* uses a Bison (YACC-like) parser to convert the script given it into a syntax tree; the syntax tree is then executed by a simple recursive evaluator. This method incurs a lot of overhead, since the recursive evaluator performs many procedure calls to do even the simplest things.

  It should be possible for *gawk* to convert the script's parse tree into a C program which the user would then compile, using the normal C compiler and a special *gawk* library to provide all the needed functions (regexps, fields, associative arrays, type coercion, and so on).

  An easier possibility might be for an intermediate phase of *gawk* to convert the parse tree into a linear byte code form like the one used in GNU Emacs Lisp. The recursive evaluator would then be replaced by a straight line byte code interpreter that would be intermediate in speed between running a compiled program and doing what *gawk* does now.

Finally, the programs in the test suite could use documenting in this book.

See the earlier section "Making Additions to gawk" if you are interested in tackling any of these projects.