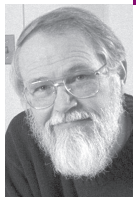
BRIAN KERNIGHAN

code testing and its role in teaching



Brian Kernighan was in the Computing Science Research Center at Bell Labs and now teaches in the CS department at Princeton, where he also writes programs and occasional books. The latter are better than the former, and certainly need less maintenance.

bwk@cs.princeton.edu

FOR THE PAST SIX OR SEVEN YEARS I have been teaching a course called "Advanced Programming Techniques" [7] to (mostly) sophomore and junior computer science majors. The course covers an eclectic mix of languages, tools, and techniques, with regular coding assignments and final group projects.

Dijkstra famously remarked that "Program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence." Nevertheless, programmers who think about testing are more likely to write correct code in the first place. Thus I try to encourage the students to do intensive testing of their code, both in one-week assignments in the first half of the semester and as part of their eight-week projects.

My personal interest in testing comes from maintaining a widely used version of AWK for the past 25 years. In an attempt to keep the program working, and to maintain my sanity as bugs are fixed and the language slowly evolves, I have created somewhat over 1000 tests, which can be run automatically by a single command. Whenever I make a change or fix a bug, the tests are run. Over the years this has caught most careless mistakes—once fixed, things tend to stay fixed and new code rarely breaks old code.

The approach is a good way to think about testing small programs, and many of the techniques scale up to larger programs. Furthermore, the topic is good for teaching about other aspects of programming, such as tools, specialized languages, scripting, performance, portability, standards, documentation—you name it, it's there somewhere. So I find myself using this material in class in a variety of ways, I think to the benefit of the students. And when I describe it to friends who work in the real world, they nod approvingly, since they want to hire people who can write programs that work and who understand something of how to produce software that others will use.

This report from the trenches focuses mainly on testing, with some digressions on topics that have led to useful class lessons. I am ignoring other important issues, such as formal methods both for writing correct programs and for validating them after the fact. These are often of great value, but I am reminded of Don Knuth's apposite comment, "Beware of bugs in the above code; I have only proved it correct, not tried it."

A Bit of History

Al Aho, Peter Weinberger, and I created AWK in 1977 [1]; around 1981 I became the de facto owner and maintainer of the source code, a position I still hold. The language is so small and simple that it remains a widely used tool for data manipulation and analysis and for basic scripting, though there are now many other scripting languages to choose from. There are multiple implementations, of which GNU's GAWK is the most widely used, and there is a POSIX standard.

The language itself is small, and our implementation [6] reflects that. The first version was about 3000 lines of C, Yacc, and Lex; today, it is about 6200 lines of C and Yacc, Lex having been dropped for reasons to be discussed later. The code is highly portable; it compiles without #ifdefs and without change on most UNIX and Linux systems and on Windows and Mac OS X. The language itself is stable; although there is always pressure to add features, the purveyors of various implementations have taken a hard line against most expansion. This limits the scope of AWK's application but simplifies life for everyone.

Test Cases

Because both language and implementation are small, the program is self-contained, and there are multiple implementations, AWK is a good target for thorough testing.

This section describes general classes of test cases. In the early days we collected and invented test cases in an ad hoc way, but gradually we became more systematic. Nevertheless, there is definitely a random flavor to many of the tests. In total, there are nearly 7000 lines of tests, in more than 350 files—there are more lines of tests than of source code. This emphasis on testing is typical of software with stringent reliability requirements, which might well have 10 times as much test as code, but it is way beyond what one encounters in a class. Merely citing the scale of testing wakes up a class; it may help convince them that I am serious when I ask them to include tests with their assignments.

One major test category probes language features in isolation: numeric and string expressions, field splitting, input and output, built-in variables and functions, control flow constructs, and so on. There are also a lot of representative small programs, such as the very short programs in the first two chapters of the AWK book. For example, the first test,

```
{ print }
```

prints each input line and thus copies input to output.

AWK was originally meant for programs like this, only a line or two long, often composed at the command-line prompt. We were surprised when people began creating larger programs, since some aspects of the implementation didn't scale, and large AWK programs are prone to bugs. But bigger complete programs like the chem preprocessor [5] make it possible to test features working together rather than in isolation, so there are a number of tests of this kind.

Some aspects of AWK are themselves almost complete languages—for instance, regular expressions, substitution with `sub` and `gsub`, and expression evaluation. These can be tested by language-based approaches, as we will see below.

There are about 20 tests that exercise the most fundamental AWK actions: input, field splitting, loops, regular expressions, etc., on large inputs. The runtimes for old and new versions of the program are compared; although real performance tuning is notoriously difficult, this provides a rough check that no performance bug is inadvertently introduced.

Each time a bug is found, a new set of tests is created. These are tests that should have been present; if they had been, they would have exposed the bug. In general such tests are small, since they are derived from the smallest programs that triggered the bug.

New tests are also added for new features or behaviors. AWK does not change much, but features are added or revised occasionally. Each of these is accompanied by a set of tests that attempt to verify that the feature works properly. For example, the ability to set variables on the command line was added and then refined as part of the POSIX standardization process; there are now about 20 tests that exercise this single feature.

One of the most fruitful places to look for errors is at "boundary conditions." Instances include creating fields past the last one on an input line, trying to set nominally read-only variables like NR or NF, and so on. There is also a group of stress tests: very large strings, very long lines, huge numbers of fields, and the like are all places where implementations might break. In theory, there are no fixed size limits on anything of significance, so such tests attempt to verify proper behavior when operating outside normal ranges.

One useful technique is to move the boundaries closer, by setting internal program limits to small values. For example, the initial size of the hash table for associative arrays is one element; in this way all arrays are forced to grow multiple times, thus exercising that part of the code. This same approach is used for all growable structures, and it has been helpful in finding problems; indeed, it just recently exposed a memory allocation failure on Linux that does not appear on Solaris.

One productive boundary condition test involved trying all AWK "programs" consisting of a single ASCII character, such as

```
awk @        (illegal) single-character program
```

Some of these are legal (letter, digit, comment, semicolon) and possibly even meaningful (non-zero digit), but most are not. This exercise uncovered two bugs in the lexical analyzer, a story to which we will return later.

Every command-line option is tested, and there are also tests that provoke each error message except for those that "can't happen."

I have tried to create enough coverage tests that every statement of the program will be executed at least once. (Coverage is measured with gcov, which works with gcc.) This ideal is hard to achieve; the current set of tests leaves about 240 lines uncovered, although about half of those are impossible conditions or fatal error messages that report on running out of memory.

I found one bug with coverage measurements while preparing this paper—the nonstandard and undocumented option -safe that prevents AWK from writing files and running processes was only partly implemented.

For all tests, the basic organization is to generate the correct answer somehow—from some other version of AWK, by some other program, by copying it from some data source—then run the new version of AWK to produce its version of the answer, and then compare them. If the answers

differ, an error is reported. So each of the examples, such as { print } above, is in a separate file and is tested by a shell loop like this:

```
for i
        do
                echo "$i:"
                awk -f $i test.data >foo1 # old awk
                a.out -f $i test.data >foo2 # new awk
                if cmp -s foo1 foo2
                then true
                else echo "BAD: test $i failed"
        fi
done
```

If all goes well, this prints just the file names. If something goes wrong, however, there will be lines with the name of the offending file and the string BAD that can be grepped for. There is even a bell character in the actual implementation so errors also make a noise. If some careless change breaks everything (not unheard of), running the tests causes continuous beeping.

Test Data

The other side of the coin is the data used as input to test cases. Most test data is straightforward: orderly realistic data such as real users use. Examples include the "countries" file from Chapter 2 of [1]; the password file from a UNIX system; the output of commands such as who or ls -l; or big text files such as the Bible, dictionaries, stock price listings, and Web logs.

Boundary-condition data is another category; this includes null inputs, empty files, empty fields, files without newlines at the end or anywhere, files with CRLF or CR only, etc.

High-volume input—big files, big strings, huge fields, huge numbers of fields—all stress a program. Generating such inputs by a program is easiest, but sometimes they are better generated internally, as in this example that creates million-character strings in an attempt to break sprintf:

```
echo 4000004 >foo1
awk '
BEGIN {
        x1 = sprintf("%1000000s\n", "hello")
        x2 = sprintf("%-1000000s\n", "world")
        x3 = sprintf("%1000000.1000000s\n", "goodbye")
        x4 = sprintf("%-1000000.1000000s\n", "everyone")
        print length(x1 x2 x3 x4)
}' >foo2
cmp -s foo1 foo2 || echo 'BAD: T.overflow huge sprintf'
```

(The very first bug in my record of bug fixes, in 1987, says that a long string in printf causes a core dump.) Again, the error message identifies the test file and the specific test within it.

Random input, usually generated by program, provides yet another kind of stress data. A small AWK program generates files with lots of lines containing random numbers of fields of random contents; these can be used for a variety of tests. Illegal input is also worth investigating. A standard example is binary data, since AWK expects everything to be text; for example, AWK survives these two tests:

```
awk -f awk              "program" is raw binary
awk '{print}' awk       input data is raw binary
```

by producing a syntax error as expected for the first and by quietly stopping after some early null byte in the input for the second. The program generally seems robust against this kind of assault, though it is rash to claim anything specific.

Test Mechanization

We want to automate testing as much as possible: let the machine do the work. There are separate shell scripts for different types of tests, all run from a single master script. In class, I describe the idea of running a lot of tests, then type the command and talk through what is happening as the test output scrolls by, a process that today takes two or three minutes depending on the system. If nothing else, the students come away with a sense of the number of tests and their nature.

Regression tests compare the output of the new version of the program to the output of the old version on the same data. Comparing independent implementations is similar to regression testing, except that we are comparing the output of two independent versions of the program. For AWK, this is easy, since there are several others, notably GAWK.

Independent computation of the right answer is another valuable approach. A shell script writes the correct answer to a file, runs the test, compares the results, and prints a message in case of error, as in the bigstring example above. As another illustration, this is one of the tests for I/O redirection:

```
awk 'NR%2 == 1 { print >>"foo" }
        NR%2 == 0 { print >"foo" }' /etc/passwd
cmp -s foo /etc/passwd || echo 'BAD: T.redir (print > and >>"foo")'
```

This prints alternate input lines with the ">" and ">>" output operators; the result at the end should be that the input file has been copied to the output.

Although this kind of test is the most useful, since it is the most portable and least dependent on other things, it is among the hardest to create.

Notice that these examples use shell scripts or a scripting language like AWK itself to control tests, and they rely on I/O redirection and UNIX tools such as echo, grep, diff, cmp, sort, wc. This teaches something about UNIX itself, as well as reminding students of the value of small tools for mechanizing tasks that might otherwise be done by hand.

Specialized Languages

The most interesting kind of test is the use of specialized languages to generate test cases and assess their results. A program can convert a compact specification into a set of tests, each with its own data and correct answer, and run them. Regular expressions and substitution commands are tested this way. For regular expressions, an AWK program (naturally) converts a sequence of lines like this:

```
^a.$  ~     ax
            aa
      !~    xa
            aaa
            axy
            ""
```

into a sequence of test cases, each invoking AWK to run the test and evaluate the answer. In effect, this is a simple language for regular expression tests:

```
^a.$  ~     ax      the pattern ^a.$ matches ax
            aa      and matches aa
      !~    xa      but does not match xa
            aaa     and does not match aaa
            axy     and does not match axy
            ""      and does not match the empty string
```

A similar language describes tests for the sub and gsub commands. A third language describes input and output relations for expressions. The test expression is the rest of the line after the word "try," followed by inputs and correct outputs one per line; again, an AWK program generates and runs the tests.

```
try { print ($1 == 1) ?   "yes" :  "no" }
1                          yes
1.0                        yes
1E0                        yes
0.1E1                      yes
10E-1                      yes
01                         yes
10                               no
10E-2                            no
```

There are about 300 regular expression tests, 130 substitution tests, and 100 expression tests in these three little languages; more are easily added. These languages demonstrate the value of specialized notations, and show how one can profitably separate data from control flow. In effect, we are doing table-driven testing.

Of course, this assumes that there is a version of AWK sufficiently trusted to create these tests; fortunately, that is so basic that problems would be caught before it got this far. Alternatively, they could be written in another language.

Another group of tests performs consistency checks. For example, to test that NR properly gives the number of input records after all input has been read:

```
    { i++ } # add 1 for each input line
END { if (i != NR) print "BAD: inconsistent NR" }
```

Splitting an input line into fields should produce NF fields:

```
{ if (split($0, x) != NF)
        print "BAD: wrong field count, line ", NR
}
```

Deleting all elements of an array should leave no elements in the array:

```
BEGIN {
        for (i = 0; i < 100000; i++) x[i] = i
        for (i in x) delete x[i]
```

```
        n = 0
        for (i in x) n++
        if (n != 0)
                print "BAD: delete count " n " should be 0"
}
```

Checking consistency is analogous to the use of assertions or pre- and post-conditions in programming.

Advice

This section summarizes some of the lessons learned. Most of these are obvious and every working programmer knows them, but students may not have been exposed to them yet. Further advice may be found in Chapter 6 of *The Practice of Programming* [3].

Mechanize. This is the most important lesson. The more automated your testing process, the more likely that you will run it routinely and often. And the more that tests and test data are generated automatically from compact specifications, the easier it will be to extend them. For AWK, the single command REGRESS runs all the tests. It produces several hundred lines of output, but most consist just of file names that are printed as tests progress. Having this large and easy-to-run set of tests has saved me from much embarrassment. It's all too easy to think that a change is benign when, in fact, something has been broken. The test suite catches such problems with high probability.

Watching test results scroll by obviously doesn't work for large suites or ones that run for a long time, so one would definitely modify this to automate reporting of errors if scaling up.

Make test output self-identifying. You have to know what tests ran and especially which ones caused error messages, core dumps, etc.

Make sure you can reproduce a test that fails. Reset random number generators and files and anything else that might preserve state from one test to the next. Each test should start with a clean slate.

Add a test for each bug. Better tests originally should have caught the bug. At least this should prevent you from having to find this bug again.

Add tests for each new feature or change. A good time to figure out whether a new feature or change works correctly is while it's fresh; presumably there was some testing anyway, so make sure it's preserved.

Never throw away a test. A corollary to the previous point.

Make sure that your tester reports progress. Too much output is bad, but there has to be some. The AWK tests report the name of each file that is being tested; if something seems to be taking too long, this gives a clue about where the problem is.

Watch out for things that break. Make the test framework robust against the many things that can go wrong: infinite loops, tests that prompt for user input then wait forever for a response, tests that print spurious output, and tests that don't really distinguish success from failure.

Make your tests portable. Tests should run on more than one system; otherwise, it's too easy to miss errors in both your tests and your programs. Shell commands, built-ins (or not) like echo, search paths for commands, and the like are all potentially different on different machines; just because something works one place is no guarantee that it will work elsewhere. I

eventually wrote my own echo command, since the shell built-ins and local versions were so variable.

A few years ago I moved the tests to Solaris from the SGI Irix system, where they had lived happily for more than a decade. This was an embarrassing debacle, since lots of things failed. For instance, the tests used grep -s to look for a pattern without producing any output; the -s option means "silent," i.e., status only. But that was true in 7th Edition UNIX, not on other systems, where it often means "don't complain about missing files." The -q of Linux means "quiet," but it's illegal on Solaris. printf on some systems prints -0 for some values of zero. And so on. It was a mess, and although the situation is now better, it's still not perfect.

A current instance of this problem arises from the utter incompatibility of the time command on different UNIX systems. It might be in /bin or in /usr/bin or be a shell built-in (in some shells), and its output format will vary accordingly. And if it's a built-in its output can't be redirected! It's tough to find a path through this thicket; I eventually wrote my own version of time.

It has also been harder than anticipated to use GAWK as a reference implementation; although the AWK language is ostensibly standardized, there are enough dark corners—for instance, when does a change in a field-splitting string take effect?—that at least some tests just produce different answers. The current test suite marks those as acceptable differences, but this is not a good long-term solution.

Check your tests and scaffolding often. It's easy to get into a rut and assume that your tests are working because they produce the expected (i.e., mostly empty) output. Go back from time to time and take a fresh look—paths to programs and data may have changed underfoot and you could be testing the wrong things. For instance, a few years ago, my "big" data set somehow mutated into a tiny one. Machines have sped up to the extent that I recently increased the "big" data by another order of magnitude.

Keep records. I maintain a FIXES file that describes every change to the code since the AWK book was published in 1988; this is analogous to Knuth's "The Errors of TEX" [4], though far less complete. For example, this excerpt reveals a classic error in the C lexer:

> Jul 31, 2003: fixed, thanks to andrey chernov and ruslan ermilov, a bug in lex.c that mis-handled the character 255 in input. (it was being compared to EOF with a signed comparison.)

As hinted at above, the C lexer has been a source of more than one problem:

> Feb 10, 2001: fixed an appalling bug in gettok: any sequence of digits, +, -, E, e, and period were accepted as a valid number if it started with a period. this would never have happened with the lex version.

And one more, just to show how bugs can hide for very long periods indeed:

> Nov 22, 2003: fixed a bug in regular expressions that dates (so help me) from 1977; it's been there from the beginning. an anchored longest match that was longer than the number of states triggered a failure to initialize the machine properly. many thanks to moinak ghosh for not only finding this one but for providing a fix, in some of the most mysterious code known to man.

Teaching

I've mentioned several places where a discussion of testing is a natural part of some other class topic; here are a handful of others.

One early assignment asks the students to program some variant of the compact regular expression code in Chapter 9 of *The Practice of Programming* [3]. As part of the assignment, they are required to create a number of tests in a format similar to the specialized language shown above and to write a program to exercise their code using their tests. Naturally, I combine all their tests with my own. It's sobering to see how often programs work well with their author's tests but not with tests written by others; I continue to experiment with assignments that explore this idea. (It's also sobering to see how often the purported tests are in fact not correct, which is another important lesson.)

I've also tried this assignment with unit tests—self-contained function calls in a special driver routine—instead of a little language. The results have been much less successful for checking individual programs, and it's harder to combine tests from a group of sources. For this application, the language approach seems better.

Another assignment asks the students to write a Base64 encoder and decoder from the one-page description in RFC 2045. This is a good example of writing code to a standard, and since there are reference (binary) implementations like OpenSSH, it's possible to mix and match implementations, all controlled by a shell script, to verify interoperability. I also ask students to write a program to generate a collection of nasty tests, which forces them to think about boundary conditions. (It's a good idea to write a program anyway, since it's easier to create arbitrary binary inputs by program than with a text editor. A surprising number of student programs don't handle non-ASCII inputs properly, and this potential error has to be tested for.)

Yet another assignment gives students a taste of a frequent real-world experience: having to make a small change in a big unfamiliar program without breaking anything. The task is to download AWK from the Web site, then add a couple of small features, like a repeat-until loop or a new built-in function. This is easily done by grepping through the source looking for affected places, then adding new code by pattern-matching old code. Naturally, they also have to provide some self-contained tests that check their implementations, and I can run my own tests to ensure that nothing else was affected.

Two years ago, an especially diligent student ran some GAWK tests against the AWK he had built, and encountered an infinite loop in parsing a program, caused by a bug in my lexer. In 1997 I had replaced the ancient Lex lexical analyzer with handcrafted C code in an effort to increase portability. As might have been predicted, this instead decreased reliability; most of the bugs of the past few years have been in this C code.

In any case, I eventually found the bug but by then it was time for the next class. So I assigned the new class the task of finding and fixing the bug (with some generous hints), and also asked them to find the shortest test case that would display it. Most students fixed the bug, and several came up with tests only two characters long (shorter than I had found) that triggered the infinite loop. Unfortunately, since that bug fix is now published, I can no longer use the assignment. Fortunately, the `-safe` bug described above should work well in its place.

Conclusions

For working programmers, there's no need to belabor the importance of testing. But I have been pleased to see how much testing can be included in a programming course—not as an add-on lecture but as an integral part of a wide variety of other topics—and how many useful lessons can be drawn from it.

It's hard work to test a program, and there are often so many other pressures on one's time and energy that thorough testing can slide to the back burner. But in my experience, once some initial effort has gone into creating tests and, more important, a way to run them automatically, the incremental effort is small and the payoff very large. This has been especially true for AWK, a language that has lived on far beyond anything the authors would have believed when they wrote it nearly 30 years ago.

Acknowledgments

I am deeply indebted to Arnold Robbins and Nelson Beebe for nearly two decades of invaluable help. Arnold, the maintainer of GAWK, has provided code, bug fixes, test cases, advice, cautionary warnings, encouragement, and inspiration. Nelson has provided thoughtful comments and a significant number of test cases; his meticulous attention to portability issues is without peer. My version of AWK is much the better for their contributions. I am also grateful to many others who have contributed bug reports and code. They are too numerous to list here but are cited in the FIXES file distributed with the source. Jon Bentley's essays on scaffolding and little languages [2] have influenced my thinking on testing and many other topics. My thanks also to Jon, Gerard Holzmann, and David Weiss for most helpful comments on drafts of this paper.

REFERENCES

[1] Al Aho, Brian Kernighan, and Peter Weinberger, *The AWK Programming Language,* Addison-Wesley, 1988.

[2] Jon Bentley, *Programming Pearls,* Addison-Wesley, 2000.

[3] Brian Kernighan and Rob Pike, *The Practice of Programming,* Addison-Wesley, 1998.

[4] Donald E. Knuth, "The Errors of TEX," *Software—Practice and Experience,* vol. 19, no. 7, July 1989, pp. 607–685.

[5] Jon Bentley, Lynn Jelinski, and Brian Kernighan, "CHEM—A Program for Phototypesetting Chemical Structure Diagrams," *Computers and Chemistry,* vol. 11, no. 4, 1987, pp. 281–297.

[6] Source code for AWK is available at http://cm.bell-labs.com/cm/cs/awkbook.

[7] The Web site for COS 333 is http://www.cs.princeton.edu/courses/archive/spring06/cos333.