Awk As A Major Systems Programming Language

(presented at Winter 1991 Usenix, Dallas)

Henry Spencer

University of Toronto

ABSTRACT

Even experienced Unix programmers often don't know *awk*, or know it but view it as a counterpart of *sed*: useful ‘‘glue’’ for sticking things together in shell programming, but quite unsuited for major programming tasks. This is a major underestimate of a very powerful tool, and has hampered the development of support software that would make *awk* much more useful. There is no fundamental reason why *awk* programs have to be small ‘‘glue’’ programs: even the ‘‘old’’ *awk* is a powerful programming language in its own right. Effective use of its data structures and its stream-oriented structure takes some adjustment for C programmers, but the results can be quite striking. On the other hand, getting there can be a bit painful, and improvements in both the language and its support tools would help.

Introduction

There is a very large gap between the UNIX® shell and the usual UNIX programming language—C—in power and ease of use. Shell programming is easy and the variety of available high-level primitives (programs) is large, but if your needs do not match the available primitives, you are basically out of luck: the low-level primitives are skimpy and extensive use of them is very inefficient. C, by contrast, provides a fairly full set of very efficient low-level* primitives, which are tricky and dangerous to use and often require extensive programming for operations that are trivial in the shell. Programmers need a simple programming language that can fill this gap: one that is easy and safe to use for simple jobs, while being versatile enough to cope with the unexpected, and acceptably efficient for undemanding tasks.

Awk [2] is a good choice for this purpose, and indeed it is widely used for small programs and for building otherwise-unavailable primitives for shell programs. It is available on nearly every UNIX system (and a good many other systems too), and with the exception of occasional niggling details, *awk* programs are highly portable.

Unfortunately, this widespread use as ‘‘glue’’ has hampered acceptance of *awk* as a serious programming language. Worse, a vicious circle has developed: the lack of appreciation of *awk*'s uses for serious programming has prevented development of the support tools that would make it more obviously viable. The combination has given *awk* an reputation as being unsuited to major programming tasks. Recent experiences have convinced us, with reservations, that this reputation is undeserved.

A Note On Terminology

There are two major variants of *awk* currently in circulation: the original [2], released with UNIX Version Seven in 1979, and ‘‘new *awk*’’, featured in the *awk* book [3] and the model for most modern implementations.

Unfortunately, although new *awk* is a considerably superior language, its availability is somewhat limited as yet: many UNIX vendors still ship the old *awk*, the independently-available implementations

* C's libraries are a disgrace [1], and improvements there would help a great deal in making it a more livable programming language, but there is little sign of this happening.

either cost substantial amounts of money or come with troublesome licences, and there are vast numbers of old systems (with old *awk*s) which will be in active use for years to come. As a practical consideration, *awk* programs intended to be portable must be written in old *awk*.

This paper will follow common practice and refer to new *awk* as *nawk*; unqualified references to "*awk*" refer to old *awk*.

A Learning Experience

One reason why *awk*'s acceptance has been slow is that, for a C programmer, it takes getting used to. Programmers who rarely use it, or use it only as "glue", seldom learn it well enough to get much appreciation of its capabilities. As with the UNIX shell and its vast array of "filter" programs, becoming a really proficient *awk* programmer takes time and experience, because not all the uses of its more non-C-like features are obvious at first glance.

In this regard, some experiences with a couple of major *awk* projects are of interest.

Text Formatting With Awk

Long long ago, in a UNIX community that certainly seems far away by modern standards, all UNIXes came with *nroff* (and perhaps *troff* as well, if you were lucky). This made *nroff* source the de-facto exchange format for complex text, notably manual pages and other documentation for software. There were some difficulties, notably the variations in macro packages, but aside from that, one could confidently send a friend program documentation and expect him to be able to read it.

Unfortunately, this happy state of affairs broke down when the UNIX text formatters were "unbundled". The theory is that only those who need the formatters will buy them. The practice is that many who need them, to varying degrees, do not have them and cannot afford them. Support for complex output devices and elaborate formatting is arguably not that common a need, but almost everybody needs a way of printing manual pages.

The "unbundled" answer to this is to provide preformatted pages, but these are not much better than printed copies. It is generally impossible to modify them for purposes like noting bugs and local changes, creating indexes into them is difficult at best, and adding your own pages to document local software is impossible without the text formatter.

The C News project [4] ran into this nuisance in connection with distributing documentation. It wasn't hard to invent a little macro package covering the forms used in our simple documentation, and this bypassed the problems of differing macro packages, but the complete lack of formatters at some sites was harder. We didn't want to distribute preformatted pages: they are a nuisance to generate, often problematic to transmit because of very long lines and control characters, and impossible to patch without spectacularly bloating *diff* listings.

In the course of moaning about this problem and cursing those responsible for unbundling, the idea of building a simple text formatter* came up. It looked like quite a bit of work in C, so it got shelved. Then the idea arose: could it be done in *awk*? The more this was investigated, the more promising it looked, for our limited purposes at least. The result was *awf**.

Awf [5] is a simple text formatter, emulating `nroff -man` or a subset of `nroff -ms`*, written entirely in (old) *awk*. It's seriously slow but has proved portable well beyond the author's expectations, with a VMS port published (in *comp.os.vms*) and a wide variety of other uses reported. It can handle almost any `-man` manual page and simple `-ms` documents. The output formatting, for a dumb terminal/printer, is nearly indistinguishable from *nroff*'s.

* Obviously, this does not solve the whole problem, since people on unbundled systems are still stuck with all the preformatted pages from their supplier. However, Geoff Collyer's experimental manual-page "decompiler", *nam* (to appear on Usenet before this paper is published, barring disasters), deals with that issue.

* Actually its original working name was *off*, meant to suggest a rather drastic subset of *nroff/troff*, and also the somewhat repellent concept. The current name seemed superior, however, and the expansion "Amazingly Workable Formatter" was invented to justify it.

* Note that, despite occasional mistaken reports, *awf* is *not* a general-purpose *nroff* emulator. It uses its own simple versions of these two macro packages, and implements very little of the full generality of *nroff*.

Awf has three passes: macro expansion (including parameters, macro nesting, and limited conditionals), command interpretation (including fonts, non-ASCII characters, and limited hyphenation), and page setting (somewhat underdeveloped by comparison, but does centering, margin adjustment, and river avoidance in particular). The sizes, in lines and bytes, are:

Pass	Lines	Bytes
1	212	5179
2	588	12311
3	332	9502

In addition, each macro package has a small piece of package-specific *awk* code, typically 30-40 lines, that is incorporated into the second pass before it is run. This typically handles a few details that cannot easily be expressed as macros in *awf*'s rather limited subset of the *nroff* input language.

Awf actually started out with much more of the semantics of the macro packages imbedded in *awk* code, but as it grew and the number of implemented *nroff* primitives rose, most of the complexity moved out into actual macro packages. The biggest problem in *awf* development was deciding where to call a halt, since simple *nroff* features often required only a few lines of code each. The code was remarkably easy to work with, even compared to normal C code*.

As mentioned above, *awf* is pretty slow, but it's fast enough to be practical when there are no alternatives! Formatting its own manual page, about 2.5 pages of moderate complexity (by manual-page standards), takes about 90 seconds of CPU time on a Sun 3/180. The second pass accounts for about half of this, with the remainder split fairly evenly between the first and the third. The bulk of it is user CPU time, although system time for the I/O is not negligible. A determined effort could probably speed this up somewhat, although almost certainly not enough to compete with *nroff* (which takes about 7 seconds to do the same text).

Lessons From ***Awf***

Awk strongly encourages programming with a ‘‘stream’’ orientation, in which the input is a stream of text lines, processed one at a time to produce output lines. Although *nawk* has added some features to permit multiple inputs, the overall structure of *awk* programs remains dominated by *awk*'s pattern+action paradigm. One can analyze lines in much the way a C program would, with one large action using *if*s to analyze the input, but in practice *awk* code is both simpler and more readable if it is broken into relatively small actions, using *awk*'s powerful and flexible patterns to decide when each is applicable. In the case of *awf*, this structure is weakly visible in the first pass, very strongly visible in the second pass, and essentially nonexistent—inappropriately so—in the third pass.

The first pass handles macro definitions with patterns and small actions, but macro expansion and conditionals are a different story. They are processed by a single huge loop with a complex mass of conditionals (thankfully, not nested much) inside it. *Nawk*'s user-defined functions could make the control structure more readable, by separating iteration through a macro body from recursion to deal with nesting. However, the control structure could also be ‘‘flattened out’’ using the existing pattern+action structure of *awk*, were it not for a second, rather more subtle, problem: although a simple assignment to `$0` will change the input line that *awk* matches patterns against, there is no way to tell *awk* to start matching all its patterns over again against the existing `$0`. The main loop of the first pass is basically an *awk* program in miniature, with each iteration examining the ‘‘input’’ line for various conditions requiring attention, manipulating a simple stack for nesting, and finally preparing the next ‘‘input’’ line.

The dominant structure of the second pass is pattern+action, interrupted by a couple of large actions with imbedded loops, one to scan a line of text for things requiring special attention, the other to evaluate the terms of an arithmetic expression. Expression evaluation would probably be better as recursive procedures in *nawk*, but input scanning would fit a generalized version of the *awk* paradigm very nicely. This would have to be defined as a control structure in the language, perhaps analogous to C's *switch*.

The third pass *could*, in retrospect, be rewritten to make much heavier use of pattern+action. At present it is mostly one huge monolithic action for historical reasons: its structure changed repeatedly

* The insides of *nroff* cannot be described as normal.

during its evolution, and the possibilities for cleanup present in the final version were not obvious in the earlier ones.

As a minor issue of pattern+action structuring, very often most actions completely deal with the input line and hence end with `next` to make *awk* pick up new input. The analogy to C's *switch* statement appears again, because subtle bugs can appear if one of those `next`s is left out and later patterns rely on not seeing input that matched earlier ones. The situation is more difficult than in C, however, because the greater generality of *awk* patterns make such ‘‘fallthrough’’ much more useful.

Apart from the overall structure, the one other conspicuous lesson from the innards of *awf*'s passes is that old *awk*'s regular-expression primitives are seriously inadequate. It was repeatedly necessary to resort to `substr` loops because, while it is easy to determine whether a regular expression matches a string, it is not possible to determine *how much* of the string it matches. The `substr` loops are not only much messier to write and read, they are also a lot slower than doing the whole scan or substitution in a single primitive. *Nawk* has addressed this problem fairly well, and one can only hope that it becomes more widely available.

A more subtle issue of how *awk* (and in particular, old *awk*) affects programming is the multi-pass structure itself. The structure has its advantages: it does a very good job of ‘‘separation of concerns’’, making the individual passes much more comprehensible than their interwoven counterparts in *nroff*. On the other hand, it requires a lot of I/O activity to emit, pass, and parse the intermediate languages. More subtly, it makes feedback loops between the passes impossible. For example, a conditional statement (in the first pass) cannot do a comparison on a string variable, because it's the second pass that does the character-by-character dissection of lines needed to implement string-variable substitution. Even when feedback loops do not interfere, defining good pass boundaries can be difficult.

Awk's stream orientation and pattern+action structure is very convenient when the problem can be broken down into fairly independent passes, but gets in the way otherwise. Unfortunately, in old *awk* there is no real alternative if one wants to avoid mass duplication of code. The pass structure of *awf*, for example, was heavily influenced by the requirement that all processing of each type of construct be in one place to avoid duplication. If a particular type of data can appear from one of several sources (e.g. text lines from original input or from inside macros), life is often simpler if those sources are merged into one by putting a pass break after them, so that the destination sees a single stream of input. A minor example of this is that the second pass of *awf* emits the equivalent of ‘‘`.ne 999`’’ at the end of input, so that the third pass need not duplicate its complex end-of-page code in its end-of-input action.

Generating Parsers In Awk

Awf was successful enough to inspire a somewhat more whimsical experiment: a parser generator written in *awk*. The input language, dubbed AASL (Amazing Awk Syntax Language), was somewhat inspired by S/SL [6]; it is a simple notation for top-down parsing, analogous to syntax charts (aka ‘‘Railroad Normal Form’’) or to the code skeleton of a recursive-descent parser. As an example, the AASL specification for a simple form of arithmetic expressions could be written like this:

```
expr: term { "+" term ?} ;
term: factor { "*" factor ?} ;
factor: ( number | "(" expr ")" ) ;
```

There are also provisions for lookahead, control of error recovery, and insertion of semantic actions.

The implementation of AASL was fairly straightforward, with AASL itself used to describe its own syntax. An AASL specification is compiled into a table, which is then processed by a table-walking interpreter. The interpreter expects input to be as tokens, one per line, much like the output of a traditional scanner. A complete program using AASL (for example, the AASL table generator) is normally three passes: the scanner, the parser (tables plus interpreter), and a semantics pass. The first set of tables was generated by hand for bootstrapping.

Apart from the minor nuisance of repeated iterations of language design, the biggest problem of implementing AASL was the question of semantic actions. Inserting *awk* semantic routines into the table interpreter, in the style of *yacc*, would not be impossible, but it seemed clumsy and inelegant. *Awk*'s lack of any provision for ‘‘compile time’’ initialization of tables strongly suggested reading them in at run time, rather than taking up space with a huge `BEGIN` action whose only purpose was to initialize the tables. This made insertions into the interpreter's code awkward.

The problem was solved by a crucial observation: traditional compilers (etc.) merge a two-step process, first validating a token stream and inserting semantic action "cookies" into it, then interpreting the stream and the cookies to interface to semantics. For example, *yacc*'s grammar notation can be viewed as inserting fragments of C code into a parsed output, and then interpreting that output. This approach yields an extremely natural pass structure for an AASL parser, with the parser's output stream being (in the absence of syntax errors) a copy of its input stream with annotations. The following semantic pass then processes this, momentarily remembering normal tokens and interpreting annotations as operations on the remembered values. (The semantic pass is, in fact, a classic pattern+action *awk* program, with a pattern and an action for each annotation, and a general "save the value in a variable" action for normal tokens.)

The one difficulty that arises with this method is when the language definition involves feedback loops between semantics and parsing, an obvious example being C's `typedef`. Dealing with this really does require some imbedding of semantics into the interpreter, although with care it need not be much: the in-parser code for recognizing C `typedef`s, including the complications introduced by block structure and nested redeclarations of type names, is about 40 lines of *awk*. The in-parser actions are invoked by a special variant of the AASL "emit semantic annotation" syntax.

A side benefit of top-down parsing is that the context of errors is known, and it is relatively easy to implement automatic error recovery. When the interpreter is faced with an input token that does not appear in the list of possibilities in the parser table, it gives the parser one of the possibilities anyway, and then uses simple heuristics to try to adjust the input to resynchronize. The result is that the parser, and subsequent passes, always see a syntactically-correct program. (This approach is borrowed from S/SL and its predecessors.) Although the detailed error-recovery algorithm is still experimental, and the current one is not entirely satisfactory when a complex AASL specification does certain things, in general it deals with minor syntax errors simply and cleanly without any need for complicating the specification with details of error recovery. Knowing the context of errors also makes it much easier to generate intelligible error messages automatically.

The AASL implementation is not large. The scanner is 78 lines of *awk*, the parser is 61 lines of AASL (using a fairly low-density paragraphing style and a good many comments), and the semantics pass is 290 lines of *awk*. The table interpreter is 340 lines, about half of which (and most of the complexity) can be attributed to the automatic error recovery.

As an experiment with a more ambitious AASL specification, one for ANSI C was written. This occupies 374 lines excluding comments and blank lines, and—with the exception of the messy details of C declarators—is mostly a fairly straightforward transcription of the syntax given in the ANSI standard. Generating tables for this takes about three minutes of CPU time on a Sun 3/180; the tables are about 10K bytes.

The performance of the resulting ANSI C parser is not impressive: in very round numbers, averaged over a large program, it parses about one line of C per CPU second. (The scanner, 164 lines of *awk*, accounts for a negligible fraction of this.) Some attention to optimization of both the tables and the interpreter might speed this up somewhat, but remarkable improvements are unlikely. As things stand—in the absence of better *awk* implementations or a rewrite of the table interpreter in C—it's a cute toy, possibly of some pedagogical value, but not a useful production tool. On the other hand, there does not appear to be any *fundamental* reason for the performance shortfall: it's purely the result of the slow execution of *awk* programs.

Lessons From AASL

Many of the earlier comments on results from *awf* also apply to AASL. The scanner would be *much* faster with better regular-expression matching, because it can use regular expressions to determine whether a string is a plausible token but must use `substr` to extract the string first. *Nawk* functions would be very handy for modularizing code, especially the complicated and seldom-invoked error-recovery procedure. A *switch* statement modelled on the pattern+action scheme would be useful in several places.

Another troublesome issue is that arrays are second-class citizens in *awk* (and continue to be so in *nawk*): there is no array assignment. This lack leads to endless repetitions of code like:

```
for (i in array)
        arraystack[i ":" sp] = array[i]
```

whenever block structuring or a stack is desired. *Nawk*'s multi-dimensional arrays supply some syntactic sugar for this but don't really fix the problem. Not only is this code clumsy, it is woefully inefficient compared to something like

```
arraystack[sp] = array
```

even if the implementation is very clever. This significantly reduces the usefulness of arrays as symbol tables and the like, a role for which they are otherwise very well suited.

It would also be of some use if there were some way to initialize arrays as constant tables, or alternatively a guarantee that the `BEGIN` action would be implemented cleverly and would not occupy space after it had finished executing.

A minor nuisance that surfaces constantly (in *awf* as well as AASL) is that getting an error message out to the standard-error descriptor is painfully clumsy: one gets to choose between putting error messages out to a temporary file and having a shell ‘‘wrapper’’ process them later, or piping them into ‘‘`cat >&2`’’ (!).

As with *awf*, the multi-pass input-driven structure that *awk* naturally lends itself to produces very clean and readable code with different phases neatly separated, but creates substantial difficulties when feedback loops appear. (In the case of AASL, this perhaps says more about language design than about *awk*.)

Support Tools

Although there are a few places where *awk* could really use language improvements, by far its biggest shortcomings right now are problems of implementation and support. The language itself is, as demonstrated by some of the above, not that big a barrier to writing major programs. Unfortunately, aspiring *awk* programmers get very little help from their environment. The usual *awk* implementation is an interpreter, well-suited to small ‘‘glue’’ programs and to fast-turnaround testing but ill-adapted to production use with large programs. Its performance for substantial computing is poor, and its facilities for debugging and tuning are nonexistent (to the point of *awk* being notorious for not even being able to produce intelligible complaints about syntax errors, although *nawk* is much better).

Surprisingly, the first support tool that *awk* would benefit from is a precise language specification. Portability of *awk* programs is annoyingly hampered by small differences in fine points of syntax, points which are not resolved by the rather informal specifications published to date. For example, putting a slash into a character class in a regular expression simply cannot be done in a portable way, because the obvious

```
/...[.../].../
```

is a syntax error in some implementations, and the fix

```
/...[...\/].../
```

puts backslash into the character class in others. For another example, while all implementations agree that a regular expression by itself is a valid pattern, implicitly matching against `$0`, there is substantial disagreement on whether this form of pattern can be combined with others by using `&&` and `||`; some *awk*s will take the combined form only if the match against `$0` is made explicit. Yet another: the original *awk* implementation was happy to accept multiple pattern+action pairs on one line, which was very convenient for trivial ‘‘glue’’ programs, e.g.

```
awk ' { x += $1 }   END { print x }' "$@"
```

but some of the more recent implementations have retroactively declared this illegal, based on vague implications in the *awk* manual (still vague in the *nawk* book) that each action should be followed by a newline. And so on. A precise, nit-picking specification of the *exact* syntax of the language would aid *awk* portability by eliminating this senseless diversity.

The next, and much more obvious, *awk* tool of importance would be a fast implementation. For example, AASL would be perfectly viable, at least for small-scale use, if its interpreter were not so slow. It's hardly surprising that an interpreter implemented in an interpreter is a bit on the sluggish side. The obvious way out of this is an *awk* compiler, preferably generating something like C as output.

However, on closer inspection, it's not quite so simple. Generating C for *awk* is a straightforward exercise, given an *awk* parser. Unfortunately, the generated C is a mass of function calls. Essentially all the data operations remain more or less interpretive, done by run-time library functions, with only the flow of control truly compiled. This is a worthwhile speedup, but not entirely satisfactory. To remove *awk* from its current status as a second-class citizen, what is wanted is an *optimizing* compiler.

For example, there is no inherent reason why an *awk* variable used only as a counter should not be compiled into a C integer, so that statements like

```
for (i = 1; i < NF; i++)
```

would run at essentially the same speed as if they were written that way in C. (In the general case there would be a slight added overhead, because integer overflow would have to be caught and referred to a more general version of the code, but most *awk* implementations limit the maximum value of `NF` to the point where even that would be unnecessary.) Naive code generation for this, however, spends vast amounts of time checking to be certain that `i` is never a string, which can usually be established by inspection of the program at compile time.

For another example, there is no need to do dynamic space allocation for the result of, say

```
s = substr($0, 1, 5)
```

since it is known to be at most 5 characters long. Space allocation is a prominent feature in the run-time profiles of *awk* programs that do a lot of string manipulation. Not *all* of it can be eliminated, but with careful data-flow analysis, a worthwhile fraction could be.

On a broader scale, *awk* programs that are written using the pattern+action scheme can spend a lot of time repeatedly checking input lines for various conditions. Often the time needed for this could be greatly cut down if the patterns were compiled together, rather than as completely independent entities. As a gross example from *awf*, if the pattern

```
/^\.(ta|ll|in|ti|po|ne|sp|pl|nr)/
```

is not matched, there is no need to even consider the later pattern

```
/^\.sp/
```

(The reason for this slightly odd-looking arrangement is that the first pattern picks out requests that need to have an arithmetic expression processed before they are executed.) More mundanely, an input line that fails to match `/^\.ne/` because its second character is not "`n`" need not even be tried against `/^\.nr/` later.

Even more broadly, multi-pass *awk* programs often are clearer and simpler than a single monolith that does the same job. However, they suffer from the high overhead of I/O on their connecting data streams. Often there is no fundamental obstacle to compiling them into a single C program, eliminating the overhead entirely, by correlating output from one pass with input to the next and making the link directly.

There are other useful tools that are not merely too slow, but completely missing as a result of *awk*'s heritage as a "glue" language. The one that almost any *awk* programmer wishes for, usually very quickly, is an *awk* debugger. As it is, *awk* debugging is back in the dark ages of inserting `print` statements and staring at the code.

Another missing tool is an *awk* profiler. This is particularly galling given the poor performance of current *awk* implementations, since there is great incentive to tune for speed and no good way of doing so. At present, the only feasible tuning technique is to rely on intuition—a notoriously unreliable guide in this area—to identify bottlenecks, and then do before-and-after timings to try to decide whether a possible improvement really helped. The unsatisfactory nature of this procedure helps to explain why a lot of *awk* programs are slow.

The various more minor tools for *awk* programming—customized editing facilities, libraries of useful functions, cross-referencers, etc.—are also worthy of note, but many of them would start to evolve fairly naturally if the bigger problems were solved and *awk* became a credible language for major programs.

The obvious question at this point is whether existing tools could be adapted to solve some of the big problems. Unfortunately, the situation doesn't look promising.

The hard part of the optimizing *awk* compiler is its optimization, not the mundane issues of parsing etc. Although concepts can be borrowed from existing work on data-flow analysis and the like, much of the implementation seems specialized enough that it would probably have to be done from scratch (unless, perhaps, an optimizing compiler for a similar language were available as a starting point).

Existing multi-language debuggers would provide at least a minimal debugging facility for compiled *awk* programs, but there might be difficulties with data representations and the presentation of source code, especially given serious attempts at optimization. Also, debugging is one area where a suitably instrumented interactive interpreter is generally superior. Given how slowly UNIX acquired such a tool even for C, *awk* programmers probably should not hold their breaths. More modest tools like customization for existing multi-language debuggers and tracing options for existing interpretive implementations would be easier.

It is *almost* possible to profile *awk* programs using the existing UNIX profiling facilities. Of course, one can do profiling, but it tells one much more about the *awk* interpreter than about the *awk* program in question, and data about the former's execution is only occasionally informative about the latter. The problem is that the existing facilities profile based on the hardware's program counter, not the equivalent in the *awk* interpreter. This could be dealt with by extending the UNIX profiling facility very slightly, so that assignment of profiling ''ticks'' to bins could be done based on the value of a programmer-supplied variable rather than the program counter.

Alternatives

Another possibility for dealing with *awk*'s problems is not to fix *awk*, but rather to attempt to identify its best features and transplant them to another language, the obvious candidates being C and C++. The potential of this approach is limited, since the concise notation will be lost to some degree. However, better subroutine libraries for C and class libraries for C++ would be substantial improvements in those languages too [1], so this is worth pursuing even if *awk* does become a credible tool for large jobs. Libraries for dynamically-allocated strings (including input and output), field structuring of input, regular expressions, and associative arrays could make a wide variety of C/C++ programs more robust and easier to write and read.

The other alternative solution is simply to use a different language, such as ICON [7]. This is potentially a satisfactory solution for a single site, although in general *awk*'s competitors are less concise for simple problems. The major difficulty with this approach is portability. *Awk* at least is widespread within the world of UNIX and UNIX-like systems, and there is hope that *nawk* may achieve similar status eventually. In terms of availability over a large number of sites and variety of machines, the only competitor for old and new *awk* is *perl*, a much uglier language (it has been described as ''*awk* with skin cancer'') with similar performance problems.

Conclusions

Awk is really a much-underestimated language. Contrary to popular belief, using it for large programs is quite feasible. The programs are a fraction of the size of C code, much easier to write and modify, and much easier to verify against specs.

UNIX support for *awk* is poor, however, most especially in the lack of a compiler. Compiling *awk* well appears to be possible, although doing good optimization is tricky. Better tools for debugging are also desirable, and very small changes to existing software would make useful *awk* profiling practical.

Given a good implementation and tool set, *awk* could take its place beside C as the preferred programming language for many Unix applications, to the great benefit of programmers and users.

References

[1] Henry Spencer, ‘‘How To Steal Code—or—Inventing The Wheel Only Once’’, *Proceedings of the USENIX Technical Conference*, February 1988, pp. 335-345.

[2] A.V. Aho, B.W. Kernighan, and P.J. Weinberger, ‘‘AWK—A Pattern Scanning and Processing Language’’, in *UNIX Programmer's Manual*, Seventh Edition, Volume 2, Holt, Rinehart, and Winston 1983, pp. 451-459.

[3] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, ‘‘The AWK Programming Language’’, Addison-Wesley 1988.

[4] Geoff Collyer and Henry Spencer, ‘‘News Need Not Be Slow’’, *Proceedings of the USENIX Technical Conference*, January 1987, pp. 181-190.

[5] Henry Spencer, ‘‘nroff -man/-ms clone written in (old) awk’’, Usenet newsgroup *comp.sources.unix*, vol. 23 issue 27, July 1990.

[6] R.C. Holt, J.R. Cordy, and D.B. Wortman, ‘‘An Introduction to S/SL: Syntax/Semantic Language’’, *ACM Transactions on Programming Languages and Systems*, Vol 4 No 2, April 1982.

[7] Ralph E. Griswold and Madge T. Griswold, ‘‘The ICON Programming Language,’’ Prentice-Hall, 1983. ISBN: 0134497775.

The Author

Henry Spencer is head of the Zoology computer facilities at University of Toronto. He was educated at University of Saskatchewan and University of Toronto, both of which may prefer not to admit it. He is the author of several freely-redistributable software packages, notably the original public-domain *getopt*, the redistributable regular-expression library, and the *awf* text formatter. He can be reached as henry@zoo.toronto.edu, or, if you must, as Henry Spencer, Zoology Computer Systems, 25 Harbord St., University of Toronto, Toronto, Ont. M5S 1A1 Canada.