Deprecating groff for BSD manual display

Kristaps Džonsons*
Swedish Royal Institute of Technology

Abstract

There are few GPL-licensed utilities remaining in BSD base installations, most of them written in C++. groff, GNU's *roff* text-processing language implementation, claims a not insignificant share of this count. Why does groff still persist in base? Although its text-processing features have been mostly usurped by LaTeX, groff persists in order to render Unix manual pages. In this paper, we introduce mdocml, a compiler for *mdoc* documents. mdocml replaces a very specific function of groff – namely, that of rendering *mdoc* documents for one or more output devices. It's our intention, in contributing this tool, to deprecate groff as the default utility for Unix manual page display; by doing so, and presuming that manual display is the primary usage of groff, we intend to decouple BSD base installations from another GPL and C++ tool, which constitutes a considerable goal of the BSD-licensed systems.

1 Introduction

The history of the groff[1] utility extends to the earliest implementations of Unix. In fact, the initial popularity of Unix is almost certainly linked to its providing a flexible text-processing system[4]. This history, and related historical intaglios, entirely digress from the focus of this document, which consists of the *roff* type-setting language, compilers, and the deprecation of a considerable GPL and C++ system from base BSD installations.

Our document will focus on the OpenBSD system when giving concrete examples. We note that the other BSD operating systems – FreeBSD, NetBSD and DragonFlyBSD – operate similarly.

We'll begin with an in-depth analysis of the *roff* type-setting language. Formally speaking, we'll find *roff* to implement a context-free grammar; however, our analysis will focus primarily on *roff* as it relates to the display of Unix manual pages.

Following an understanding of *roff*, we'll analyse the Unix troff utility for compiling *roff* documents. troff implements only the compilation phase of *roff* to an intermediate output format; thus, we'll also consider the remaining pipe-line of tools to render *roff* documents, mixed with pre-processor domain-encoded sections, to a specific output device.

Since the focus of this document is on the deprecation of groff, we'll briefly examine the GNU implementation of troff and the other elements of the *roff* processing pipe-line. Our analysis will also consider the utility of *roff* on modern systems, forming the basis of our case for deprecation.

Finally, we'll introduce mdocml[2], a tool for compiling un-mixed *mdoc* documents. mdocml interfaces with a library for parsing *mdoc* documents, libmdoc, which produces an abstract syntax tree completely describing its input. With an intact tree, mdocml is able to convert to any number of outputs, and thus any number of rendering devices. In keeping with our focus on mdocml as a replacement for groff in displaying manual pages, we'll focus on a terminal-encoding output device.

Furthermore, we'll demonstrate how the translation of *mdoc* documents into an intermediate, unambiguous abstract syntax tree allows for clean separation of the parsing mechanism from its output devices. In doing so, we introduce mdocml for producing both terminal-encoded output, toward replacing groff, as well as producing HTML output,

*kristaps@kth.se
[1] http://www.gnu.org/software/groff/
[2] http://mdocml.bsd.lv/

deprecating the many *ad hoc* systems for rendering Unix manuals for on-line display. In doing both, we hope to free BSD operating systems from both a heavy-weight GPL/C++ utility.

2 *roff* Language

In the literature, the term "roff" may refer to the language *roff*, the processing tool-chain for *roff* documents, or the historic utility itself. In this document, we italicise *roff* when referring to it as a language, otherwise, we'll specifically clarify usage.

A full analysis of the *roff* language is beyond the scope of this document. Thus, we refer to [1] and [3] for complete study of the relevant systems. Most of our examples in this section draw from [1].

Formally, *roff* is a context-free grammar. *roff* documents are composed of input lines of two ontological categories: *parse-able text*, which begins with the control character "." and contains function-like macros; and *free-form text*, which constitute the body of a prior scope.

In other words, free-form lines are interpreted according to the parser state, while parsed lines consist of tokens ("macros") affecting the programme state.

```
.\" -*- macro opens scope -*-
.ul 1
This text is underlined.
```

Parsed *roff* tokens are described by a flexible and powerful grammar, which includes partial loops, branch instructions and arithmetic operators common to most Turing-complete languages. The following demonstrates arithmetic and a simple conditional:

```
.\" -*- arithmetic: subtraction
.nr val \\n(val-2
.\" -*- conditional: strcmp
.if string1string2 stuff
```

Importantly, *roff* allows for strongly-prototyped, opaque function blocks, called macros, to be defined at run-time. Macro packages, like function libraries, provide document authors with a subset of the original language. This simplifies the production of syntactically-similar documents, as authors need only refer to their domain-specific macros and may ignore the powerful but more-complex roff superset.

The following simple macros produces small-caps for the initial argument:

```
.de SM
\s-2\\$1\s+2
```

Executing `.SM TROFF` produces "TROFF".

Unix manual pages are conventionally written with two macro packages: *man* and *mdoc*. As with *roff*, we italicise *mdoc* and *man* when referring to the macro languages.

Of these two, we focus on *mdoc*, which is used for BSD-style manuals. BSD operating systems conventionally use the *mdoc* for manuals in base installations, although manual authors are free to use general *roff* invocations. The *mdoc* macros are documented on Unix systems under the `mdoc.samples` and *mdoc* in-system manuals.

By using a strongly-prototyped, limited subset of the *roff* language, writing technical manuals is considerably simplified – this is especially useful to programmers already burdened with their programming language of choice, with little time to learn another for their documentation. Since most manuals follow a conventional syntax, the limitations of *mdoc* versus generalised *roff* go largely unnoticed.

Earlier in this section, we noted that *roff* implements a context-free grammar; in general, such grammars are straight-forward to programmatically scan and parse. Unfortunately, since macros describe opaque bodies of *roff* tokens, the set of macros in a package may not itself implement a context-free grammar. This is an effect of masking transitions in the deterministic state machine described by the context-free grammar. Although a well-designed macro package may possibly be context-free, the *mdoc* package is not: it is, instead, context-sensitive.

We demonstrate this with the following example:

```
.\" First list:
.Bl -column "xxxxxx" "xxxxxx"
.It 1 Ta 2
.El
.\" Second list:
.Bl -tag
.It 1 Ta 2
.El
```

In defining the first list as columnar, the `Ta` macro describes a break between columns; in the second, the `Ta` is not interpreted. In a context-free grammar, this construction is disallowed. The importance of this discrepancy will be of particular relevance later in this document when we consider implementing a compile directly for mdoc in order, amongst other reasons, to bypass the time-consuming effort of implementing the entire roff language.

3 troff Utility

The troff utility is a compiler accepting *roff* input and optional macro definitions, and producing an intermediate output, referred to as "troff output" and formally documented in [2]. This output is subsequently rendered to an output by device by other utilities: it abstractly describes the layout and style of a document.

In terms of Unix manuals, an *mdoc* document is accepted by the troff utility along with the *mdoc* macro definitions. The compiler in-lines the opaque macro bodies as specified by the macro prototypes; the produced output is interpreted as pure *roff* and converted into the intermediate language.

The transition between *roff*, which may be mixed with macro invocations, and intermediate output conceals a considerable transformation: while intermediate output may be reconstituted fully into *roff*, albeit not necessarily syntax-equivalent with the original document (according to the formal, theoretic properties of context-free grammars), the syntax introduced by using macro invocations will generally be lost.

In other words, the annotation of the macro is lost when converting into the intermediate form. We demonstrate with the following mdoc macro:

```
.Va variable
```

The *mdoc* macro annotates the "variable" parameter as a source code variable; the produced *roff* code is as follows (second line truncated to four arguments):

```
.ds mN Va
.aV \\$1 \\$2 \\$3 \\$4
```

The `aV` macro, with `mN` defined as `Va`, eventually transforms the parameters to a font transformation of `12n`.

Intermediate output similarly labels the "variable" text as having a font size. Thus, the informal annotations introduced by invoking macros are necessarily redacted. The importance of this will become clear when we consider wanting to compile *mdoc* documents to and from various intermediate formats without loss of annotations that encode the *meaning*, not the *style*, of data.

4 troff Pipe-line

The troff pipe-line centres on the troff utility. It is divided into pre-processors and post-processors, which may alternately be described as pre-troff and post-troff. Pre-troff utilities produce *roff* output, which is accepted by the troff utility; post-troff utilities accept the intermediate language and render to their respective output mediums.

troff pre-processors accept domain-encoded input, usually *roff* mixed with domain-encoded sections, and replace domain-encoded sections with *roff*. Since a pipe-line may include any number of pre-processors, it's clear that an input document may include several non-overlapping domain-encoded sections; by the time the pre-processing phase completes, this sections should be replaced and the resulting document be well-formed *roff*.

The reason for this method, instead of providing complex, domain-specific macros, is largely flexibility. Encoding mathematical or chemical formulae,

for example, may be easy to do with a domain-specific language, but very complicated when operating within the limitations imposed by *roff* or strongly-prototype *roff* macros. Two standard pre-processors, eqn and chm, do exactly this.

Furthermore, pre-processors are clued as to the desired output device, allowing them to choose the format of their output. This is particularly useful when comparing whether to write formulae to a text or graphical terminal.

Post-processors accept intermediate language and render the document to their respective output mediums. The most popular, mentioned later in this document, is grotty, which terminal-encodes output.

5 groff Utility

groff is a GPL-licensed implementation of the *roff* pipe-line. The term "groff" describes both the suite of utilities that collectively render *roff* documents, and the stand-alone executable orchestrating this pipe-line. In this document, we generally refer to groff as encompassing the suite of utilities as a whole.

Although groff introduces a number of extensions, it's largely feature-compatible with the original Unix troff pipe-line. The current implementation is written in C++, with the source code alone amounting to roughly 60 000 lines of mixed C and C++ source, which implement standard pre-processors, the troff utility, and standard post-processors.

The existence of groff in BSD operating systems poses an issue: not only is the system GPL licensed, which conflicts with the BSD license, but the utility is written in C++. This prevents deprecating the industry-standard GNU compiler collection, gcc, with a simpler C-only compiler.

6 Conventions

In this section, we examine the most common usage of groff in modern BSD systems. We address OpenBSD for specific invocations; FreeBSD and NetBSD operate similarly.

There are three in-system invocations of groff: for on-line conversion of *mdoc* documents to terminal-encoded output with the man utility; when generating a cached "catman" manual; or when compiling various kernel documents. We discount the third scenario as the rendered documents appear not to be in use.

In the first case, the man utility sources the `man.conf` file to determine how best to display manuals (what follows is a relevant fragment):

```
_build  .[1-9n] nroff -man %s
_build  .tbl    tbl %s | nroff -man
```

Within system compilation, the `bsd.man.mk` Make-file fragment invokes, similarly to the man utility, groff or tbl as piped to groff. Additionally, the `bsd.man.mk` script can generate Postscript output (with the `-Tps` instead of the `-Tascii` or `-Tlatin1` arguments for terminal output).

A tool deprecating groff must handle, in decreasing order of importance, on-line display of *mdoc* documents by piping through groff; off-line display by caching groff output in a file; Postscript output; and, least, tbl pre-processed tbl manual files.

In the remaining sections of this document, we introduce the mdocml tool, which explicitly compiles *mdoc* documents into a variety of output formats. This satisfies, as listed above, the majority of groff utility.

7 mdocml Utility

The mdocml tool is composed of two components: a scanner-parser, which produces an abstract syntax tree representing the *mdoc* input; and the output filter, which accepts the abstract syntax tree and transforms it into a domain-encoded format.

The mdocml utility uses a simplified version of the troff pipe-line. Foremost, no pre-processors are necessary or allowed, as input is assumed to be well-formed *mdoc*. This is overwhelmingly the case in those manuals distributed with the BSD systems,

although some complexities do exist (see §8 for details).

Like troff, mdocml produces an intermediate output in the form of an abstract syntax tree (or simply "AST"), which fully encodes the annotations of the original *mdoc*.

Finally, the AST is piped an output filter, which re-encodes the AST in a domain-specific language. This is most commonly terminal-encoded output, but may also be HTML for on-line manuals, or PDF or PS for printable media.

In accepting *mdoc* macro input, mdocml focusses on the meaning of tokens, not their style. This allows output filters to evaluate stylistic annotation based on the original annotation: instead of the AST obscuring a variable as a font-size, this logic is delegated to the output filter.

7.1 mdocml AST

In this section, we discuss the reasoning behind and structure of the mdocml intermediate abstract syntax tree. This tree is particularly important because it must unambiguously encode, in a regular fashion, context-sensitive and irregular data.

A side-effect of generating an AST is that the contents of an *mdoc* document are fully validated. Validation covers argument counts, layout, compatibility with groff and nroff, and so on.

The AST itself is generated by the libmdoc function library, which is interfaced by the mdocml utility. The libmdoc library is distributed with the mdocml package as a static library that may be interfaced by any caller. In this simple library, a parsing session is opened, lines are parsed, then the result is collected after all lines are parsed.

The syntax tree is composed of typed nodes, expressed by the following diagram in an abbreviated Wirth form, where capitalised non-terminals are tree nodes and terminals are always tree leaves:

```
MDOC  = NODE {NODE}
NODE  = ELEM | BLK | text
ELEM  = {text}
BLK   = (HEAD BODY | HEAD | BODY) [TAIL]
HEAD  = {[ELEM | BLK | {text}] {punct}}
BODY  = [ELEM | BLK | {text}] {punct}
TAIL  = {text} {punct}
```

That is to say, a document contains one or more nodes. Nodes are of type element, block, or un-typed text. An element node may optionally contain text; a block node contains at least a head and/or body node and optionally a tail, all of which may be separated by un-typed punctuation nodes (a sub-type of text node that only contains punctuation).

Informally, an element corresponds to an in-line value such as an italicised or underlined parameter, or possibly a variable or function type.

Blocks are somewhat more complicated, as they are scoped to have a head and possibly a body and tail. These correspond to lists and block displays.

Syntactically, elements are macros with line parameters, while blocks usually, but not necessarily, have multi-line scope.

The following demonstrates a simple document fragment of elements:

```
.Va var1
.Va var2 var3
```

This parses into the following parse tree:

```
MDOC
  ELEMENT
      TEXT::var1
  ELEMENT
      TEXT::var2
      TEXT::var3
```

A more complex example includes a block scope:

```
.Sh NAME
.Bl -enum
.It
Hello
.El
World
```

This parses into the following nested block (with a spanner for clarity of tree siblings):

```
MDOC
  BLOCK
    HEAD
      TEXT::NAME
    BODY
    +-BLOCK
    |   BODY
    |     TEXT::Hello
    +-TEXT::World
```

Our choice in this ontology is largely to fit the irregular syntax of *mdoc* macros. An example of irregularity is in parsing lists, whose sub-type compositions are sensitive to the context of the caller.

```
.Bl -column "xxx" "xxx"
.It Fl 1 Ta Fl 2
.El
.Fl 1 Ta Fl 2
```

In the above example, the `Ta` macro is interpreted differently according to its context. In the first case, the `Ta` separates different head nodes (columns) for the list item; in the second case, it's interpreted as a text string.

```
MDOC
  BLOCK
    HEAD
      ELEMENT
        TEXT::1
    HEAD
      ELEMENT
        TEXT::2
    TEXT::Ta
  ELEMENT
    TEXT::2
```

The list block opened by `Bl`, which is among the more complicated macros to represent in regular form, can have two "head" macros to denote multiple columns only if `-column` is included. The latter invocation is far cleaner:

```
MDOC
  ELEMENT
    TEXT::1
```

Not all block nodes are explicitly opened and closed (`Bl` and `El` in the above examples). Implicitly-scoped nodes, such as `Sh`, have their scopes closed out by subsequent invocation. Some implicitly-scope nodes can close out others, such as a section `Sh` closing out contained sub-sections `Ss`.

Furthermore, not all block nodes span multiple lines: the quotation macros, such as `So` and `Sc`, may be executed in a single line enclosure.

Due to these and more regularities, the mdocml parser implemented in libmdoc is *ad hoc*, meaning that macros are parsed according to the chain of parents, siblings, and a table for condition rules.

Internally, the parsing routine, which generates a sub-tree of the resulting abstract syntax tree, has three phases for each parsed macro in the line:

1. Parse macro arguments: arguments begin with hyphens and may contain zero or more values.
2. Pre-validate macro: check for invocation context (list items must be in lists, displays may not nest, etc.) and sane arguments.
3. Parse child nodes: element text children, block sub-nodes, and so on.
4. Post-validate macro: check for child node validity and count, and so on.
5. Perform per-macro actions: setting the resident document name, prologue values, and so on.

Post-parsing is the most complex area, as nested macros must be post-processed recursively when their scopes close. Post-parsed validity also checks for groff compatibility, including maximum line arguments, macro argument limitations, strange handling of child text, and so on.

7.2 mdocml Filters

When a parsing session is closed and the AST is intact, it may be further processed. mdocml may

either generate a binary file representing the AST, with suffix `.mdoc`, or pipe it to another utility.

When run without an output filter, mdocml merely constructs the AST (validating its input) and exits.

mdocml has, at present, two output filters, with one more in design. The first, mdocml-tree, outputs the AST in an indented tree. It's largely intended for testing.

The second, mdocml-term, generates a terminal-encoded output similar to running `nroff -Tascii -mandoc`. This utility is intended to deprecate usage of groff for terminal-encoded output. Internally, it links to the termcap library routines in order to format style (bold and underlined strings) and layout (indentation, centring, and so on). mdocml-term can either generate decorated text (with stylistic mode) or un-decorated text.

The third, in development, is mdocml-html. This filter generates an HTML file from the original manual, and deprecates various other HTML generators for manuals such as man.cgi.

8 Status

A considerable amount of work remains to be done in order to have full compatibility with groff and other features useful to operators and programmers.

The libmdoc parser is generally complete, although some corner cases (such as multi-head list columns described in §7.1) are still experimental in terms of their end design.

The `Xo` and `Xc` macros, which extend the number of arguments for a macro, are un-implemented for list item heads. Whether to implement this macro pair is still under debate: although their function has been long deprecated by groff (which earlier considerably limited the number of arguments), they're still in use in a fair number of base installation manuals.

An open question is whether to deprecate some irregular groff behaviour, such as the strange syntax of the `At` macro and the artificially-limiting maximum number of line arguments.

mdocml-term is still experimental and has no backward call-syntax compatibility with groff. This isn't necessarily a bad idea: mdocml only implements *mdoc*, and even when called with `-mdoc`, manuals may be mixed with pure *roff*.

mdocml does not, and probably never will, support the standard groff pre-processors eqn, refer, and chm. To date, tbl is the only pre-processor used. It's not unreasonable to fold this into the mdocml parser library: since list item columns (mentioned in §7.1) are represented by several "head" nodes, it's conceivable to design a converted from tbl to *mdoc* list-item, or directly parse these within mdocml. However, to date, only fifteen manuals make use of this pre-processor.

9 Conclusion

The groff utility consists of over 700 files, totalling over 200 000 lines of bitmaps, C and C++ source and headers, *groff* macros, output device specifications, and so on. Of these, over 50 000 lines are C++ sources. These sources are predominantly licensed under the GPL. We described the groff utility itself in §5.

As described in §6, the overwhelming use of groff is for manual display, which decomposes into compiling *mdoc* documents into terminal-encoded text output. We demonstrated in §7 the capability of mdocml to match this usage as a sub-set of its functionality.

By deprecating the primary function of groff, the utility itself may be relegated to the BSD operating systems' respective ports (third-party) collections, freeing a considerable chunk of C++ and GPL sources. Furthermore, by deprecating a considerable C++ code-base, we hope to further step toward a non-C++ base distribution, without need for a complex C++ compiler in the base distribution.

References

[1] Brian W. Kernighan. A troff tutorial, 1978.

[2] Brian W. Kernighan. A typesetter independent troff. Technical Report 97, Bell Labs, 1981.

[3] Joseph F. Ossanna. Nroff/troff user's manual. Technical Report 54, Bell Labs, 1976.

[4] Dennis Ritchie. The evolution of the unix time-sharing system. In *Proceedings of a Symposium on Language Design and Programming Methodology*, pages 25–36, London, UK, 1980. Springer-Verlag.