An Introduction to the GNU `Groff` Text Processing System

Manas Laha

mlaha@aero.iitkgp.ernet.in

ABSTRACT

The GNU `groff` text processing system is based upon a markup language very similar to that of UNIX `troff` and is mostly compatible with it. `Groff` is capable of producing typeset output in a variety of formats such as Postscript, TeX dvi, plain text and HTML. It can produce colour output in Postscript. `Groff` can accept input comprising of text, figures, tables, mathematics and embedded Postscript. This article is an introduction to `groff` for the new user. Its aim is to arouse curiosity about `groff` and provide pointers to resources, mostly on the Internet. It does not aim at completeness, for not only would that take up far too much space but, considering the excellent documentation already available from various sources, would also be redundant. This article describes the purpose and nature of `groff` and briefly recounts its history. It also describes the companion programs of `groff` at some length, with many examples. The `ms` macro package and the `pic`, `tbl` and `eqn` preprocessors that handle line art, tables and mathematics, respectively are introduced. Some examples of innovative uses of `groff`, such as its use in making presentation slides, as a back-end processor and as a pseudo-"wysiwyg" text processor are given. Finally, the names of some well-known books produced using either of these tools is mentioned. In order to retain the introductory nature of this article, 'pure' `groff` is talked about hardly at all. This means that features such as programmability and the writing of macros are not discussed at any length.

17 June 2003

An Introduction to the GNU `Groff` Text Processing System

Manas Laha

mlaha@aero.iitkgp.ernet.in

Table of Contents

> "This book was phototypeset by the authors using the excellent software available on the UNIX system. The typesetting command read
>
> ```
> pic files | tbl | eqn | troff -ms
> ```
>
> `Pic` is Brian Kernighan's language for typesetting figures; we owe Brian a special debt of gratitude for accommodating our special and extensive figure-drawing needs so cheerfully. `Tbl` is Mike Lesk's language for laying out tables. `Eqn` is Brian Kernighan and Lorinda Cherry's language for typesetting mathematics. `Troff` is Joe Ossanna's program for formatting text on a phototypesetter, which in our case was a Mergenthaler Linotron 202/N. The `ms` package of `troff` macros was written by Mike Lesk. In addition, we managed the text using `make` due to Stu Feldman. Cross references within the text were maintained using `awk` created by Al Aho, Brian Kernighan, and Peter Weinberger, and `sed` created by Lee McMahon."
>
> — Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, authors of
> *Compilers: Principles, Techniques, and Tools*

> "Camera-ready copy of the book was produced by the authors. It is only fitting that a book describing an industrial-strength software system be produced with an industrial-strength text processing system. Therefore one of the authors chose to use the `groff` package written by James Clark, and the other author agreed begrudgingly."
>
> —Gary Wright and W. Richard Stevens, authors of
> *TCP/IP Illustrated, Volume 2: The Implementation*

1. Preamble

This section tells about the aims of this article, how it is laid out and from where to obtain its source. It also states the license under which this article is released.

1.1. Aims and Limitations of This Article

This article is aimed at giving a first feel of text processing using `groff` (pronounced "gee-roff"), which is a descendant of the classic UNIX text processing system `troff` (pronounced "tee-roff"). If you are an expert user of either, you will not find much here that is likely to be new to you. On the other hand, if you have never used `groff` before and wish to get an introduction to using it, then you may profit from reading this article. I have made no attempt at giving a complete exposition of the `groff` system, which is quite elaborate. However, I have tried to be accurate in the statements I have made here.

Should you become motivated to learn more about `groff`, you can refer to the many excellent articles about `troff` and `groff` that are available, several of which I have listed in the bibliography.

1.2. Organization of This Article

§ 2 defines what the terms 'text processing' and 'document' mean in this article. To keep things simple, this article does not talk about how to lay out or produce books. § 3 is devoted to a quick look at the basics of GNU `groff`. Only the most elementary aspects of the use of `groff` are discussed. Macros are introduced, but only just. § 4 is about writing articles with the `ms` macro package. It also tells how to format and obtain the final form of the document. § 5 touches upon the preprocessors that work with `groff` to

produce figures, tables and mathematics. § 6 elucidates some further capabilities of `groff`, such as usage of colour and importing of Postscript files. § 7 introduces some not-so-obvious uses of `groff`, such as its use as a backend processor, its development into a pseudo-'wysiwyg' text processor and its use in producing presentation slides. § 8 lists some books *produced* with `troff` and `groff`, as well as quotes what some of the authors have to say about these text processors. § 9 is the concluding section.

1.3. Availability of this article

I have written this article about GNU `groff` using GNU `groff`. (Writing it any other way would have made this whole exercise pointless!) The source text, including the bibliographic database, is available in electronic form. Anyone interested may obtain the same from me by sending an e-mail to `mlaha@aero.iitkgp.ernet.in`.

1.4. License

As the author and sole copyright holder of this article, I grant this license giving unhindered access to all those who want to read it. Following are the terms:

A. I, Manas Laha, have written this article entitled *An Introduction to the GNU `Groff` Text Processing System* with the aim of introducing the system to new users. As its only author I am the sole owner of the copyright to it. Nothing that is said in the following paragraphs of this license or in any other license that may be imposed by others who print or electronically host this article should be taken to prejudice my rights as the author and sole holder of the copyright to this article in any manner.

B. Although this article is available in source form, its contents, including its title and bibliography, may not be changed nor any derivative works be made out of it without my express permission. However, authors of articles or books or of any other endeavour to further the cause of GNU `groff` in particular and human knowledge in general may quote parts of it with due acknowledgment.

C. This article is available to all interested persons to read, to access, to host and to distribute, in source form or in any processed form, in print or electronic media. However, the license to do so expressly excludes distribution in any form if such distribution is for monetary gain, personal, institutional or corporate.

2. What is Text Processing?

For the purposes of this article, we shall take "text processing" to mean

a) the creation or assembling together of related information on a topic in a form suitable for storage on and processing by computer, and

b) the serving of it in various formats, printed or web-based, to the reader.

The information presented may consist of running text, tables, figures, pictures and mathematical equations. Web-based documents may also contain hyperlinks.

The text processing software, under the writer's direction, has to take care of proper layout and structuring of the document. This includes, but is not limited to

- paragraph justification,
- proper line- and page-breaks and hyphenation,
- usage of fonts and styles appropriate to the context,
- managing bibliographic information
- inclusion of figures and pictures from outside the document,
- automatic numbering of figures, tables and equations,
- checking for spelling errors and, last but not least,
- giving a professional and harmonious look to the finished document.

Some text processing systems include utilities that evaluate the readability of a document in terms of good English!

2.1. The Structure of a Document

For the purposes of this article we shall consider documents to consist of broadly three different types: letters, articles and books.

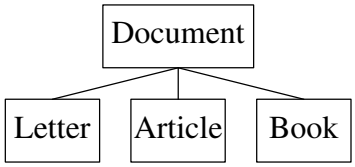

Fig.1 A classification of documents by type

Letters are presumably the simplest kind of documents, consisting of running text without figures or mathematics, and books the most complicated. A letter generally consists of the sender's and receiver's addresses (usually, but not necessarily, written at the top), followed by the body, consisting of paragraphs.

Journalistic articles, such as those written for newspapers and magazines, usually are structured like letters. On the other hand scholastic articles, purporting to make a learned exposition of some subject, generally are not only larger in size than a letter but have more distinctive parts. An article on a mathematical subject also makes heavy use of mathematical symbols and equations, and these often form a significant proportion of the body of the article. Figure 2 shows how such an article is usually structured.

We shall exclude books from the purview of this article, not because `groff` is not suited for producing books but because books are big and usually much more elaborate than letters or articles. Therefore a discussion on how to produce books using `groff` would not be in keeping with the introductory nature of this article.

3. The Basics of GNU `Groff`

GNU `groff` is a *text processing system*. That is, it is a collection of programs to create, on the computer, documents of the type described in § 2.1. The contents of the document are typed into the computer through a text editor, such as is used to edit program source files. Commonly used editors belong either to the `vi` family or to the `emacs` family. The formatting of the document into say, the title, the body, paragraphs, figures and tables is taken care of by *formatting requests* which are sequences of ordinary text characters that

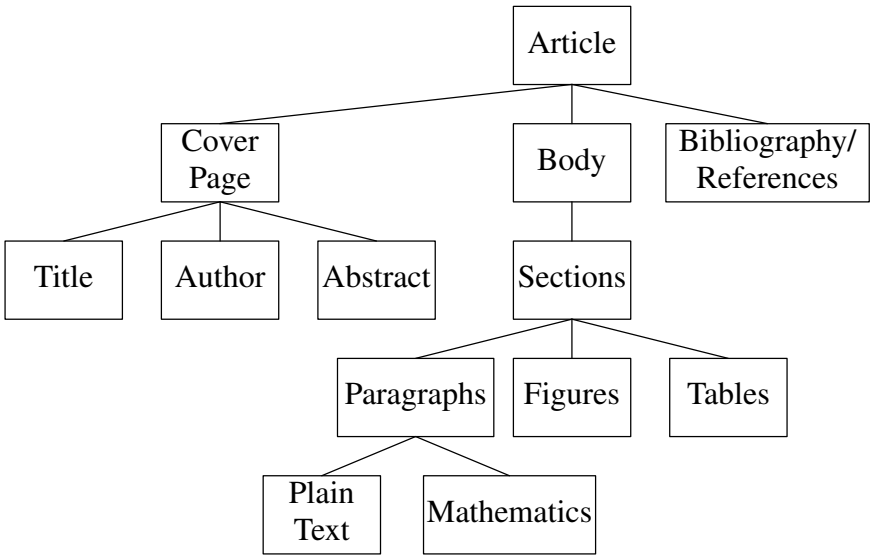

Fig.2 Components of document type 'article'

convey special meaning to the `groff` system. Once the document has been entered and marked up, it is run through the text processor (also known as the *formatter*). The formatter is told what form of output—for example, Postscript, HTML or plain text—is desired. Only *after* the formatting is complete is it possible to see the document in its final form.

In the following sections we shall first take a look at formatting requests at the very lowest level of `groff`. These low-level requests are sometimes known as 'pure' `groff`. They may be likened to the individual instructions of an assembly language. All formatting tasks are ultimately accomplished using these pure `groff` requests. However, just as in an assembly language so here, it is easier to achive common formatting tasks by grouping into *macros* the formatting requests needed to accomplish them. Entire *packages* of macros are available to format documents in common use, mainly letters and articles. This is analogous to the macro libraries that may be available for use with assembly languages.

The formatting commands of `groff` are numerous and require time and effort to describe as well as to learn. Here we shall only give a few very elementary illustrative examples. Complete information and many examples on the subject are available from the references listed in the bibliography, mainly Refs.[1], [2], [3] and [4].

3.1. Creating Documents With `Groff`

There are, broadly speaking, two steps to creating documents with `groff`:

- Preparing the document source
  by entering the contents of the document alongwith the markup into the computer and
- Generating the finished document
  by passing the document source through `groff`.

We shall here see how to prepare the document source. The second step, generating the finished document, we shall take up later.

3.2. `Groff` Formatting Requests

The formatting requests that tell `groff` how to format a document are three character strings, with the leading character being a period. For such an entity to be interpreted by `groff` as a formatting request, the period *must* appear in the first column. For example, the formatting request `.ce` tells `groff` to centre the line following this request across the page. So when `groff` sees input like this:

```
.ce
This line is centred across the page
```

it produces the output:

This line is centred across the page

To take another example, the input

```
.rj
This line is flush with the right margin
```

gives, on formatting:

This line is flush with the right margin

The default behaviour of `groff` is to fill and adjust output lines as long as an input line is not completely blank or contains leading spaces or tabs. So, input lines like these:

```
A fox jumped up one winter's night
And begged the moon to give him light,
For he'd many miles to trot that night
Before he reached his den O!
    Den O! Den O!
    Before he reached his den O!
```

format into:

> A fox jumped up one winter's night And begged the moon to give him light, For he'd many miles to trot that night Before he reached his den O!
> Den O! Den O!
> Before he reached his den O!

Note how the first four input lines are joined together to produce a filled and right adjusted paragraph. The fifth and sixth input lines have leading spaces and so a 'break' occurs in the output: the filling and adjusting process stops and a new output line begins; the leading spaces are left undisturbed.

If we wanted to alter the default behaviour so that *no* filling of output lines would occur, we would have to tell that to `groff` with a `.nf` request, like this:

```
.nf
A fox jumped up one winter's night
And begged the moon to give him light,
For he'd many miles to trot that night
Before he reached his den O!
    Den O! Den O!
    Before he reached his den O!
```

to obtain:

A fox jumped up one winter's night
And begged the moon to give him light,
For he'd many miles to trot that night
Before he reached his den O!
  Den O! Den O!
  Before he reached his den O!

In typesetting, two paragraph styles are generally in vogue: one, called the 'normal' paragraph, where the first line is indented to the right by some amount (say 0.3 inches) and another, called the 'block' paragraph, where the first line is flush with the left margin. Also, both kinds of paragraphs are separated from the immediately preceding text by some amount of additional space; let's say this extra space is one line. So, if we have input such as:

```
.sp
.fi
This is a 'block' paragraph. In it the first line is
flush with the left margin, as are all lines that follow.
Note how the .sp request causes an additional space of one
line to be left before the paragraph begins and the .fi
request causes line filling and adjusting to occur.
```

then, upon formatting, we get output like this:

This is a 'block' paragraph. In it the first line is flush with the left margin, as are all lines that follow. Note how the .sp request causes an additional space of one line to be left before the paragraph begins and the .fi request causes line filling and adjusting to occur.

Let us add a normal paragraph to this example, which now looks like this:

```
.sp
.fi
This is a 'block' paragraph. In it the first line is
flush with the left margin, as are all lines that follow.
Note how the .sp request causes an additional space of one
line to be left before the paragraph begins and the .fi
request causes line filling and adjusting to occur.
.sp
.fi
```

```
.ti 0.3i
And this is a 'normal' paragraph, in which the first
line is indented by 0.3 inches to the right. This is
achieved with the .ti request, which stands for
'temporary indent', meaning that only the line
immediately following this request is indented.
Apart from inches, other scaling factors are centimetres
(for example, .ti 1.0c) and points (as in .ti 22p). A point
is about 1/72 of an inch.
```

This piece of input formats into:

> This is a 'block' paragraph. In it the first line is flush with the left margin, as are all lines that follow. Note how the .sp request causes an additional space of one line to be left before the paragraph begins and the .fi request causes line filling and adjusting to occur.
>
> And this is a 'normal' paragraph, in which the first line is indented by 0.3 inches to the right. This is achieved with the .ti request, which stands for 'temporary indent', meaning that only the line immediately following this request is indented. Apart from inches, other scaling factors are centimetres (for example, .ti 1.0c) and points (as in .ti 22p). A point is about 1/72 of an inch.

The power and flexibility of `groff` is much increased by the fact that it is *programmable*. That is, the formatter may be instructed to change its course of action depending upon the current states of the document source and output. For example, it is possible to instruct `groff` as follows:

```
If this is the first page
then
    do not print the page number;
    increase the top margin by 1.2 inches
else
    print page number at bottom centre of page
fi
```

We shall not be going into the programmability of `groff` here.

3.3. `Groff` Macros

Imagine that you are creating a document where you need both the paragraph styles described above. The method explained for creating them will work but will soon lead to problems. Firstly, before every paragraph you will have to type in the appropriate format requests. When the number of paragraphs becomes large this can quickly become irritating. Then, there is the real possibility that you may inadvertently forget a format request, leading to a malformed paragraph. Again, let us say that, having created and marked up the complete document, you feel that the gap between successive paragraphs needs adjusting. Imagine what a boring and error-prone task you have before you: to search out every occurrence of `.sp`, determine if it pertains to a new paragraph, and then change it to something else. And, after all this, if you wish to chage the amount of inter-paragraph

space again, you have to go through this process once more. Just the thing to put you off groff for good.

But wait! There's a trick that groff is holding up its sleeve in the form of the *macro*. A macro is a collection of groff format requests of the kind we have already seen. Let us define macros for creating block and normal paragraphs:

```
.\" Macro to produce a block paragraph
.de lp
.sp
.fi
..
.\" Macro to produce a normal paragraph
.de lp
.sp
.fi
.in 0.3i
..
```

Every macro has a name. The macro for producing block paragraphs is called lp, and that for normal paragraphs is called pp. The definition of a macro begins with the .de request followed by the macro name and is terminated with a .. on a line all by itself. Between these go all the formatting requests that make up the macro. Incidentally, in groff the character sequence .\" introduces a comment.

A macro *must be* defined before it is called. And in most cases a macro may be called simply by writing its name preceded by a period. Our last example may be rewritten as:

```
.lp
This is a 'block' paragraph. In it the first line is
flush with the left margin, as are all lines that follow.
Note how the .sp request causes an additional space of one
line to be left before the paragraph begins and the .fi
request causes line filling and adjusting to occur.
.pp
And this is a 'normal' paragraph, in which the first
line is indented by 0.3 inches to the right. This is
achieved with the .ti request, which stands for
'temporary indent', meaning that only the line
immediately following is indented.
```

where we assume that the macros .lp and .pp have already been defined as shown above. Formatting this little piece produces exactly the same output as before.

Now, having got the macro definitions in place, there is no chance of inadvertently omitting one of the several formatting requests before a paragraph. Also, experimenting with different paragraph formats—such as inter-paragraph spacing or first line indentation—is far easier, with consistent results throughout the entire document assured.

The capability to define macros and write appropriate programs in the groff language is exploited to the hilt by groff users. For example, we have seen two macros,

`.lp` and `.pp`, to mark up two different paragraph styles. These styles are mostly the same, except that in the second paragraph style the opening line is indented by some specified amount. It could be argued that instead of having two macros with largely common code it would be better design to have a single macro with a parameter specifying which kind of paragraph is wanted. So if this new paragraph macro is called just `p`, one might format a normal paragraph with just the request

```
.p
```

and a block paragraph with the request:

```
.p 0
```

where the `0` is a parameter to the `.p` macro. The relevant part of the code for this macro may look something like this:

```
.de p
    ...

    if ( parameter1 = 0 ) then .sp 0.3i
..
```

Many other instances of such usage abound.

3.4. `Groff` Macro Packages

While it is useful to be able to write one's own macros, doing a realy bullet-proof job of it can be tough. It is, in general, easier to use *canned* packages of macros for routine document formatting tasks. These packages provide "a macro for any occasion"—well, almost. With reference to the structure of a typical article as shown in Fig.2 (§ 2.1), these macro packages have macros for the various components such as title, numbered and unnumbered section headings, several kinds of paragraphs, figures, tables and equations. With such a macro package, it is possible to work in *functional* terms, telling `groff`: "This is a section heading", "This is a block paragraph", "This is a figure" and so on; rather than in *procedural* terms, where you would have to tell `groff`: "Make this line bold" or "Indent this line by half an inch".

Among the major macro packages in use with `groff` the `ms` macro package, which originated at Bell Labs in the early days of `troff`, still seems to be the most popular. While it is fine for producing an article like this one, its limitations would soon become apparent were it to be used to produce a book with several chapters, a big bibliographic list with lots of references, table of contents and index.

The `mm` macro package, also from Bell Labs, is an improvement over `ms`. It does not seem to be as popular as `ms`, though.

The `me` macro package from the University of California at Berkeley, developed at around the same time as `ms`, is also widely used. It is more extensive and more easily customizable than the `ms` macro package.

Another important macro package, and perhaps *the* reason why `groff` is found on all Linux systems (just as `troff` is found on all UNIX systems), is the `man` macro package for formatting on-line manual pages.

Lately a major new set of macros named `mom`, written by Peter Schaffter,[5] has been added to the `groff` suite. It is more sophisticated than either `ms` or `me` and is being improved. It works with `groff` version 1.18 or later.

3.5. `Groff` Special Characters

Very often during typesetting it becomes necessary to enter characters that do not have any representation on conventional computer keyboards. In such cases `groff` recognizes a combination of several conventional characters, in what is known as an *escape sequence*, as representing these characters. All escape sequences begin with the `\` (backslash) character. The next character determines what kind of escape sequence this is: special character, font change instruction, size change instruction or something else again.

To take an example of a special character, the *em dash* (—), which is a dash approximately as long as the width of the letter *m*, is written as `\[em]`. Similarly, the *en dash* is written `\[en]`. If you read documentation about `troff`, you will see that the em dash and the en dash are written as `\(em` and `\(en`, respectively. This is the *old style* of writing escape sequences. While this style works, it restricts the escape sequence name to only two characters. The new style can take arbitrarily long names. Special characters with long names may be introduced in future as `groff` becomes more internationalized. The following table lists some of the useful special characters. Several more are available. These are all listed in the Manual.[1]

Character	Composite	Name	Character	Composite	Name
•	`\[bu]`	bullet	□	`\[sq]`	square
¼	`\[14]`	1/4	°	`\[de]`	degree
½	`\[12]`	1/2	†	`\[dg]`	dagger
¾	`\[34]`	3/4	©	`\[co]`	copyright
—	`\[em]`	em dash	®	`\[rg]`	registered TM
–	`\[en]`	en dash	§	`\[sc]`	section
-	`\[hy]`	hyphen	_	`\[ru]`	rule
_	`\[ul]`	underrule	|	`\[br]`	box rule
‡	`\[dd]`	double dagger	○	`\[ci]`	circle
←	`\[<-]`	left arrow	→	`\[->]`	right arrow
↑	`\[ua]`	up arrow	↓	`\[da]`	down arrow
☜	`\[lh]`	left hand	☞	`\[rh]`	right hand
“	`\[lq]`	left quote	”	`\[rq]`	right quote

Table 1 Some `Groff` special characters

3.6. Character Font and Size Changes, Superscripts and Subscripts

In the course of typesetting, it often becomes necessary to set characters in styles different from the default: *italics*, **bold**, `constant width` or perhaps others; and in sizes different from the standard, such as SMALL or big. In this section I shall describe how to do these.

Font changes are effected with `groff` escape sequences. A font change escape sequence consists of the character sequence `\f` followed by the name of the font. For example,

```
\fIHello, World!\fP
```

formats to

Hello, World!

Here, `\fI` is the escape sequence to change to italic font and `\fP` is the escape sequence to revert back to the *previous* font. Arbitrarily long strings may appear between two font change escape sequences. For example:

```
\fIThis is a sentence in italics.\fP
```

produces

This is a sentence in italics.

The escape sequences `\fB` and `\fR` cause changes to the bold and roman fonts, respectively. Some fonts require two character font names such as `CW`. These may be written either as `\f(CW` (old style) or `\f[CW]` (new style). The old style is limited to accepting only two character names while the new style can accept arbitrarily long names.

Character size changes can be similarly brought about. For example, this document is set in 12 point characters; however, it is possible to switch to a different character size in this way:

```
.ps 24
This is 24 point
.ps 6
and this is 6 point.
.ps 12
This is back to 12 point.
```

which gives:

This is 24 point

and this is 6 point.

This is back to 12 point.

Relative size changes are also possible. The input

```
.ps +12
This is 24 point
.ps -18
and this is 6 point.
.ps +6
This is back to 12 point.
```

leads to the same result as before:

This is 24 point

and this is 6 point.

This is back to 12 point.

The same effect may be produced with escape sequences, absolute and relative. The input

```
\s24This is 24 point.\s0
This is back to 12 point.
\s-6This is 6 point\s0.
And this is back to 12 point.
```

leads to the output

This is 24 point.

This is back to 12 point.

_{This is 6 point.}

And this is back to 12 point.

Note that the escape sequence `\s0` causes the size to revert back to the immediately preceding size (in this case 12 pt).

Superscripts and subscripts are effected with escape sequences that cause upward or downward vertical motion. The input

```
The 15\uth\d day of March.
```

produces the output

The 15th day of March.

The escape sequence `\u` causes an upward motion of half a vertical line, and the escape sequence `\d` causes an equal vertical motion in the downward direction.

In order to prevent them from running into the line of text just above or below the current one, superscripts and subscripts are set in a size than smaller normal. Observe the two following examples and notice the difference:

And the soothsayer said to Caesar:
"Beware the 15th of March."

And the soothsayer said to Caesar:
"Beware the 15th of March."

The second example, with the superscript set in characters three points smaller than normal, was produced like this:

```
And the soothsayer said to Caesar:
"Beware the 15\u\s-3th\s0\d of March."
```

Notice the character size change commands around the "th".

3.7. 'Wysiwyg' or markup?

As you can see, the `groff` approach to formatting documents using markup tags is in marked contrast to the "*w*hat *y*ou *s*ee *i*s *w*hat *y*ou *g*et" (*wysiwyg*) approach used in some other systems. What, if any, are the advantages of the markup approach over the 'wysiwyg' approach? This is a hotly debated topic (as a search of the Internet with the keywords 'wysiwyg' and 'markup' will reveal!). Let me list some of the pros and cons of the markup method.

The positives:

- You can concentrate on *what you are writing* instead of bothering about *how its going to look*. This is because, with a well-designed macro package, the markup tags correspond to the functional parts of the document. When you tell `groff` "The following line is a section heading", then `groff` automatically selects a font size and style that is in keeping with the rest of the document. Later, if you decide to change the style of the document to a smaller font, `groff` automatically adjusts the section heading to suit. Moreover, `groff` takes care to make all items marked 'section heading' of the same style.

- When the document source is in plain text then, at least on UNIX-like systems, it can be examined and manipulated with all the powerful tools available for the purpose.

- Another advantage pertains to the markup scheme being an open one where you know how the document is stored internally: it is that you can develop *your own* programs to manipulate the document in any desired way. If you feel that some feature is missing from your favourite text processing system, you can go ahead and try to implement it yourself. Some examples of such 'creative uses' of `groff` appear in a later section.

The negatives:

- The 'instant feedback' of the final form of the output that a 'wysiwyg' system gives is missing from most markup systems.

  However, there are ways around this, and in a later section is described how to *make your own* pseudo-'wysiwyg' system based on `groff` for near-instant feedback.

- Markup text processors have a steeper learning curve than their 'wysiwyg' counterparts. However, what is also true (and which most people don't realise) is that the curve is equally steep for learning *efficient* use of a 'wysiwyg' system.

This article may help you to decide whether a markup text processing system like `groff` is suitable for you.

3.8. A Brief History of GNU `Groff`

The full flavour of a piece of software that has its roots in UNIX cannot—in my strongly held opinion!—be completely savoured without knowing something about its history, and `groff` is no exception.

When the creators of UNIX at Bell Labs began to look for a bigger machine to run their system on than the 'unused' DEC PDP-7 where it was born, they had to sell to their bosses the idea of developing a 'text formatting system' for their company in order to get the machine they wanted—a PDP-11. And thus, officially, UNIX was first revealed to the world as a text formatting system.[6]

Of several text formatting programs then available for UNIX, the one to become most popular was `troff`, written by Joe Ossanna in PDP-11 assembly language around 1973. This program was used to drive the Wang C/A/T photo-typesetter that Bell Labs owned then, and worked with only that output device. Ossanna later rewrote the entire

program in C and continued making improvements to it until his death in an automobile accident in 1977.

The creators of troff decided at the outset that special aspects of typesetting would be handled by special purpose 'languages', each custom built for the task. These special purpose languages are known as 'preprocessors'. The first of the preprocessing languages to make its appearance was eqn, to typeset mathematics, followed by tbl, to typeset tables. According to Kernighan:

> "Eqn came first. It was the first use of yacc for a non-programming language. It is improbable that eqn would exist if yacc had not been available at the right time. Tbl came next, in the same spirit as eqn, although with an unrelated syntax. Tbl doesn't use yacc, since its grammar is simple enough that it's not worthwhile."[6]

As troff continued to be used, the need was felt for it to work with a variety of output devices. In the early nineteen-eighties, Brian Kernighan rewrote troff, making it produce output that was independent of any physical device. This new troff became known as 'device independent' troff or ditroff. This ditroff output was converted for various physical devices, such as photo-typesetters and printers, by appropriate *post-processors*. This is the model that groff is patterned on. By the time Kernighan wrote ditroff, Postscript and printers than understood it had appeared. To make fuller use of the capabilities of these new output devices came the pic preprocessor, which understood an English-like language for describing simple pictures.

Beginning around 1989, James Clark almost single-handedly developed GNU groff.[7] Clark's goal was to emulate the functionality of ditroff and its pre- and post-processors while removing some of the bugs of the original program and making some enhancements. Clark nursed groff up to version 1.11 and then abandoned it. After that groff remained an orphan for some years, until new maintainers took over in 1999 and development activities resumed once more. The latest groff version at the time of this writing is 1.19, released in May 2003. One of the most notable recent contributions to groff has been the HTML post-processor that can generate HTML from groff source.

Groff continues to be included on every Linux distribution (as does troff on every UNIX distribution), if only for the sole reason that the formatting of the online manpages depends on it. It can, however, be used for tasks of far greater variety and complexity, as this article aims to show.

3.9. Articles and Documentation About Groff

The GNU groff homepage is at http://groff.ffii.org and is the primary source of everything related to groff. The groff source tarball, downloadable from the homepage, includes a fair amount of documentation in various formats: manpage, HTML, PDF and Postscript; as well as some rudimentary examples. Unfortunately, most Linux distributions tend to ignore all but the manpages in order to save CD space.

The documentation accompanying a groff source distribution attempts to keep up to date with the latest developments in groff. But this documentation is still very much 'work in progress' and lacks the completeness and finished touch of classical works (which are, actually, about troff). While the older works cannot tell us about the latest

developments they are, even today, *very good* sources of information about groff and its companion programs. This is because groff has consciously attempted to stay backward compatible with troff. In a later section I shall talk about the historical relation between groff and troff.

And now for the classical sources. As with all things having to do with UNIX, one of the best introductions to troff (and hence also to groff) is by Kernighan and Pike.[6] It is not only very readable but has the additional merit of being readily available in print.

The other classic in this field, out of print but nevertheless available on the Internet thanks to O'Reilly Publications, is *Unix Text Processing* by Dougherty and O'Reilly.[4] This is a *really detailed* exposition of all aspects of troff and how to combine it with other UNIX tools to create a very powerful text processing system. This book is compulsory reading for the serious groffer (or should that be groffian? I can't seem to make up my mind!).

Among the articles written for the original troff, foremost mention must be made of the manual by Ossanna himself, modified later by Kernighan.[1] This is a detailed exposition of all troff commands and their options, and is an essential companion while writing documents with groff. Its relative lack of examples is made up for by the tutorial by Kernighan.[2] A newer, readable and informative, article on groff is *The GROFF and Friends HOWTO* by Provins.[3]

Lesk[8] and Kollar[9] (and also Provins[3]) describe how to format documents using the ms macros. Allman does the same for the Berkeley me macros, with his excellent tutorial[10] and manual,[11] both of which are available with groff source distributions *as* groff *source*. Studying them is an excellent way of learning how to write documents using groff and me. Documentation about the mom macros is available from the mom homepage .[5]

The Jeffreys Copeland and Haemer have written some interesting pieces on UNIX and troff usage.[12] Their site should be of interest to all those who love to learn new ways of using UNIX as well as groff.

4. Creating Documents With Groff and the Ms Macros

This section tells about marking up a document with the ms macros. For this purpose a sample document is first shown as it would appear after formatting by groff. Then the document source is given and the markup tags used in it are discussed. The discussion of ms macros is limited to those actually used in the sample document.

Following this is a description of how to use groff to transform the document source into the final, finished, form. These forms include Postscript, TeX dvi and plain ASCII. The last form is suitable for displaying text such as manual pages on the terminal.

Lastly, the article gives some tips on printing Postscript files on the printer, especially with an eye on conserving paper. Some ideas are given for two-sided printing and 'n-up' printing. The latter format is useful for printing documents that are most conveniently produced in book form (as opposed to the more common loose-leaf form).

4.1. A Sample Document

Here is a short article, purporting to provide succinct yet deep insight into the human condition, which we shall consider as our sample document:

The Road

Yours Truly

Koi Thikana Kahin Nahi

ROAD, n. A strip of land along which one may pass from where it is too tiresome to be to where it is futile to go.

— Ambrose Bierce, *The Devil's Dictionary*[†]

1. The Urge

Oh! The pain and the anguish. The itch and the sore. The dissatisfaction and the discontent. The urge to move on. And on. Isn't this what has brought men, women and children of all nations and races, hues and sizes, castes and creeds to The Road?

Wasn't there the belief and the hope, the expectation and the anticipation of Something Better down The Road? After all isn't the grass always greener, the air always more scented, the fruits always sweeter and the water always cooler further down The Road?

And so it is that countless people, yes, people like you and me (me? maybe not!) have taken to that strip of land called The Road.

2. The Futility

Where does The Road come from? And where does it go? Who do the answers to these questions know?

Having skipped and hopped, jumped and tumbled, frisked and gambolled, walked and trudged, limped and crawled their way down The Road the travellers arrive. Or think they have. Where is the green grass? Where the scented air? Where is the promised land? Where the milk and nectar?

Oh! It was then all in vain. Was it to be done all over again? Ah! The sadness. The remorse. The dejection. The hopelessness. The cries of "It was futile!"

3. *Deja vu*

Is this the end? No. Is this even the beginning of the end? Not quite. What is this then? The beginning? Yes! For history repeats itself. Again. And again. And again ...

---

†Ambrose Bierce, *The Devil's Dictionary*, `http://www.alcyone.com/max/lit/devils` (1911).

The next section explains just how to create this document using `groff`.

4.2. Preparing the Document Source

The document source is a plain text file and hence may be created and modified using any text editor. Some writers prefer to write the article completely in plain text and only then mark it up. Others (including me!) prefer to write and mark up at the same time. Both approaches seem to work, so take that which feels more comfortable. (If, like me, you use pseudo-'wysiwyg' tools, you may find the second approach more convenient.)

The source for the sample document of the last section looks like this:

```
.TL
The Road
.AU
Yours Truly
.AI
Koi Thikana Kahin Nahi
.sp 0.3i
.LP
.QS
ROAD, n. A strip of land along which one may pass from where
it is too tiresome to be to where it is futile to go.
.rj
\[em] Ambrose Bierce, \c
.I "The Devil's Dictionary"
.[
%A Ambrose Bierce
%T The Devil's Dictionary
%I http://www.alcyone.com/max/lit/devils/
%D 1911
.]
.QE
.NH
The Urge
.LP
Oh! The pain and the anguish. The itch and the sore. The
dissatisfaction and the discontent. The urge to move on.
And on. Isn't this what has brought men, women and children
of all nations and races, hues and sizes, castes and creeds
to The Road?
.PP
Wasn't there the belief and the hope, the expectation and
the anticipation of Something Better down The Road? After
all isn't the grass always greener, the air always more
scented, the fruits always sweeter and the water always
cooler further down The Road?
.PP
And so it is that countless people, yes, people like you and
me (me? maybe not!) have taken to that strip of land called
The Road.
.NH
The Futility
.LP
Where does The Road come from? And where does it go?
Who do the answers to these questions know?
.PP
Having skipped and hopped, jumped and tumbled, frisked and
```

```
gambolled, walked and trudged, limped and crawled their way
down The Road the travellers arrive. Or think they have.
Where is the green grass? Where the scented air? Where is
the promised land?  Where the milk and nectar?
.PP
Oh! It was then all in vain. Was it to be done all over again?
Ah! The sadness. The remorse. The dejection. The hopelessness.
The cries of "It was futile!"
.NH
\f4Deja vu\fP
.LP
Is this the end? No. Is this even the beginning of the end? Not
quite. What is this then? The beginning? Yes! For history repeats
itself. Again. And again. And again ...
```

This example is marked up using the `ms` macros. These macros provide the functional markup requests needed to format a document. The markup requests used in this example are all macros. Each such macro request is equivalent to a number of pure `groff` requests and consequently does many things. (See the introduction to macros in § 3.3.)

In this example, the first line is `.TL`. This markup request tells `groff` that the following lines make up the *title* of the document. The user need not bother about how the title will look: `groff` will set it in bold, in a font two sizes larger than normal text and centre it on the page. All lines following `.TL` will be taken to be the title, until another request (that is, a line beginning with a period) is encountered.

The next request is `.AU`, that tells `groff` that an author's name follows next. Author names are set in normal size but italicized and centred. After the name, an author's institutional affiliation may be written preceded by a `.AI`. This will be set in normal type. If there is more than one author then this sequence of `.AU`, author name, `.AI` and author affiliation is repeated.

The body of this article begins with a `.LP` request, which calls for a block paragraph. The `.QS` and `.QE` requests enclose a *verbatim quotation*, which is set in at both margins with extra space before and after, to distinguish it from the main text.

A section heading is indicated with `.NH` for *numbered heading*. Different depths of numbering are possible. The depth is indicated by a numeric argument to the `.NH` request. Thus,

```
.NH
This is a heading at depth 1
.NH 2
This is a heading at depth 2
.NH 2
This is another heading at depth 2
.NH
And this is a second heading at depth 1
```

would, after formatting, produce:

1. This is a heading at depth 1
1.1 This is a heading at depth 2
1.2 This is another heading at depth 2
2. And this is a second heading at depth 1

Another kind of heading is the *un-numbered* heading, indicated by `.SH`:

```
.SH
This is an un-numbered heading
```

formats into

This is an un-numbered heading

`Groff` knows that headings are to be set in bold.

A typographical convention followed here is to make the paragraph immediately following a section heading a block paragraph, indicated to `groff` with a `.LP` request. The second and subsequent paragraphs are "normal" paragraphs, meaning that the first line is indented by some amount. A normal paragraph is requested with `.PP`.

If you look again at Fig.2 (§ 2.1), you will see that the major parts of a document of type 'article' are the title, the author(s), an abstract, and a body consisting of sections which in turn consists of paragraphs. The macros provided by the `ms` macro package closely follow the different parts of the structure of a document of type article. There's a macro for the title, there a macro for the author, there are macros for section headings and also for paragraphs in a couple of styles. Although we haven't used them in our example, there are macros for making abstracts and bibliographies.

Using the formatting macros allows the writer to concentrate on *what* he is writing, rather than on *how* its going to finally look. For example, he can tell `groff`: "This is the document title." without bothering about specifying the details of how to make the title centred, bold and so on. `Groff` will do it right!

The construct

```
.[
%A Ambrose Bierce
%T The Devil's Dictionary
%I http://www.alcyone.com/max/lit/devils/
%D 1911
.]
```

is a special one, having to do with `refer`, a component of `groff` that handles bibliographical references. You can see how the bibliographical reference appears as a footnote at the bottom of the sample document. A `refer` reference has several key symbols. Each symbols begins on a new line. Thus, `%A` denotes the author field (there may be more than one such field), `%T` the title of the work, `%I` is the name of the publisher (in this case the URL where this document is to be found), `%D` the date of publication. There are more such key symbols indicating, for example, the city of publication (`%C`), the particular pages within the work that this reference pertains to (`%P`), and so on.

Usually a set of such bibliographical references with all details is defined in a separate file. Each reference is identifies by a unique keyword (the `%K` field). In the final

output, the citation within the document is replaced with a number and the details appear at the end of the document.

4.2.1. Changing Character Fonts and Styles With the `Ms` Macros

Taking style changes first, the input

```
This is a
.I very
big article.
```

produces

This is a *very* big article.

as the output. Similarly, the input

```
It was produced with
.CW groff
text processing system.
```

produces

It was produced with `groff` text processing system.

as the output. Again, the input

```
The article
.B "will be"
improved from time to time.
```

gives as output

The article **will be** improved from time to time.

Therefore the macros `.I`, `.B` and `.CW` affect their arguments by setting them, respectively, in italics, bold or constant width font.

Note how a string has to be enclosed in quotes in order to 'hide' the spaces it contains. Without the quotes,

```
.B will be
```

would format into

willbe

and not into what was perhaps intended. This example shows that if there are *two* arguments to a font change macro, then only the first is affected and, on output, the second argument is *concatenated* to the first without any intervening space. This feature has its uses, for example in:

```
The spellings
.I skil ful
and
.I skill ful
are both acceptable.
```

which formats into

> The spellings *skil*ful and *skill*ful are both acceptable.

If we wish to set a block of text—say an entire paragraph—in a font different from the default, we invoke the corresponding font change macro without any argument. The new font then remains in effect until changed back again. Here is an example:

```
The computer program
.CW
main()
{   printf( "Hello, World!\n" );
}
.R
is perhaps better known than any other.
```

which formats into:

> The computer program
> `main()`
> `{   printf( "Hello, World!\n" );`
> `}`
> is perhaps better known than any other.

The macro `.CW` without arguments changes the font to constant width, and this change remains effective till the macro `.R` changes the font back to roman again.

4.3. Formatting the Document With `Groff`

Once the document source is ready, it can be formatted into any of several output forms. The default output form for `groff` is Postscript. If the document source is stored in the file `example1.txt` then the command:

```
groff -ms < example1.txt > /tmp/example1.ps
```

will produce a Postscript document in the file `/tmp/example1.ps` which, when viewed in a Postscript viewer such as `ghostview`, will look exactly like the example of § 4.1. (I like all my transient output files to go into the `/tmp` directory; `/tmp` on my machine is automatically wiped clean at each reboot.)

The `-ms` option to the `groff` command is the name of the macro package that has been used to mark up the document source. If you had marked up a document with the `me` macros, then to format it you would do:

```
groff -me < example2.txt > /tmp/example2.ps
```

in order to get a Postscript output file. To format a UNIX-style manual page into Postscript, for example, on my Red Hat Linux 8.0 system

```
zcat /usr/share/man/man1/gcc.1.gz | groff -man > \
                                    /tmp/gcc.ps
```

will produce a Postscript version of the `gcc` manual pages. (It actually may be a good idea to print large manual pages, such as those of `gcc` and `bash`, out on paper to read

without too much strain on one's eyes and nerves.) The command

```
man gcc
```

does something similar to:

```
zcat /usr/share/man/man1/gcc.1.gz | \
     groff -man -Tascii | less
```

Here the `-Tascii` option tells `groff` that the output device is going to be a terminal.

4.4. Printing the Resulting Postscript File

In these days of high automation, printing a Postscript file amounts to little more than clicking on a '`Print`' button somewhere. But this method of obtaining a printout often allows little control over the printing process. I, personally, am particularly concerned about making the best use of paper and so take my printouts on both sides of a sheet whenever practicable. If you too wish to do this, and more, then here's how to go about it.

4.4.1. Saving Paper While Printing

The first thing is to learn how to convert the Postscript to a file in your printer's language. (Printers that understand Postscript are expensive.) The tool to effect such conversions is `ghostscript` or `gs`. If, on your screen, you type

```
gs --help | less
```

you will see a list of devices that `gs` can output to. (243 devices are listed in `gs` v7.05!) Not all of these are hardware devices, though. The ones of interest to us here are those concerned with printers: `deskjet` for generic HP Deskjet printers, `laserjet` for generic HP Laserjet printers, `epson` for (you guessed it!) Epson printers and many other devices for specific models of these. The specific model drivers can better access or control the capabilities of printers of those models.

Let's see how to convert our Postscript file into a file for a generic HP Deskjet printer. Here is a shell script, named `ps2dj`, that does this:

```
#!/bin/sh

    gs -dSAFER -dNOPAUSE -sDEVICE=deskjet -sPAPERSIZE=a4 \
       -sOutputFile=${1}%02d.dj \
    ${1}.ps << END
    ^D
END
```

The `gs` manual tells what these options mean. The construct

```
    ${1}.ps << END
    ^D
END
```

is a Bourne shell "here document",[13] and `^D` is Ctrl+V+D. If our Postscript file contains three A4 pages, then invoking the shell script with the filename as argument, like this:

```
ps2dj example1
```

(note that the extension `.ps` is left out; it is supplied by the shell script) produces *three* files, named `example100.dj`, `example101.dj` and `example102.dj`, which are in a language that the printer understands.

To actually obtain double-sided printouts, you will have to bypass your system's print spooler (for example `LPRng` or `CUPS`). This is necessary because, when you send several print requests through the spooler, there is no guarantee that the jobs will be printed in the same order in which they were submitted. This, as you can imagine, can be disastrous in two-sided printing.

Next, you will have to make your printer port globally writable. If, for example, you have a parallel printer and your printer device is `/dev/lp0` (the most common case), then saying

```
chmod a+w /dev/lp0
```

will make the printer device writable by all. Now, if you simply copy the `.dj` files to `/dev/lp0` they should be printed on the printer. Put two sheets into the paper tray, then type

```
cp example100.dj /dev/lp0
cp example102.dj /dev/lp0
```

This will print pages 1 and 3 on one side of each sheet.

Next, feed the sheet containing the printout of the first page such that any additional printouts would be appearing on the blank side, and say

```
cp example101.dj /dev/lp0
```

to obtain a printout of the second page on the back of the first one.

With a little practice, you should have no trouble obtaining double sided printouts. If you do this regularly then, for all but the smallest documents, a shell script to manage the printing is very useful. I use a very simple one, named `prinman`, that looks like this:

```
#!/bin/sh

# Print listed pages

suff=$1
shift
pref=$1
shift

while [ "x$1" != "x" ]
do
    fnam=${pref}${1}.${suff}
    if [ -r $fnam ]
    then
        cat $fnam >/dev/lp0
    else
```

```
                echo $0: file $fnam not found, exiting

                exit 1
            fi

            shift
        done
```

and, for our example above, is invoked in two passes, like this:

```
prinman dj example1 00 02
```

and

```
prinman dj example1 01
```

where the first argument is the filename suffix, the second argument is the "constant" part of the filename and all the remaining arguments are the (two-digit) numeric parts of the filename.

For long documents, I've found that I get the best results on my old faithful DeskJet 600 by printing the even pages out *in order*, followed by the odd pages *in reverse order*. In this way the pages all come out in the right order and need no shuffling. Let's say the document `example1.ps` has ten pages and the printer files are named `example100.dj` to `example109.dj`. So I print out the even pages in order:

```
prinman dj example1 01 03 05 07 09
```

(the files with odd numeric extensions contain even pages and vice versa!) and then I take the entire bunch of five pages out of the tray, turn the bunch around without disturbing the order, put it back in the tray and print the odd pages out in reverse order:

```
prinman dj example1 08 06 04 02 00
```

This order should work for injet printers. Laser printers *may* require a different order, though. Figure it out for yourself!

4.4.2. Saving *Even More* Paper While Printing

If, like me, you believe that every bit of paper wasted is that much less for coming generations (think of your children, and their children and ...) then you will like the following ideas. Not only is it great to be able to print on both sides of the paper, but it is even better to be able to fit *two* (or more!) pages on one side of the paper.

The program that allows you to do this is `psnup`, a part of the `psutils` package. Taking our example of the ten page document once more, suppose you have produced the Postscript file `example1.ps` from the `groff` source in the usual way. Now when you say:

```
psnup -2up -pa4 < example1.ps > ex2up.ps
```

what you're doing is producing another Postscript file with *two* normal pages side by side on one page of A4 paper. The pages are automatically turned around so the side-by-side juxtaposition happens with the physical sheet in landscape mode. The original pages 1

and 2 go on the new page 1, original pages 3 and 4 go on the new page 2 and so on. So you now have a *four-fold* saving of paper over the automatic printing method.

Of course, there are limitations to this process. If the original pages were printed in a point size smaller than about 12, then the reduced pages are going to be hard to read unless you printer is a good one and you print in 'high quality' mode. Also, if you are printing out a large document (like this one?) then you can't get the pages stitched down their middle like a book because they will not be in the right order.

If you wish to make a book, then you have to arrange the pages into 'signatures' of two pages each, in the right order. Then you can stitch the pages down the middle and fold them like a small book. To do this, use:

```
psbook < example1.ps | psnup -2up -pa4 > exbook.ps
```

Before actually printing `exbook.ps` out, take a look at it with `ghostview` to convince yourself that it will really work.

4.5. Output in Other Than Postscript Format

`Groff` is capable of producing output in formats other than Postscript. Some of the more important of these other formats are given in Table 2.

Output Type	Device Name
Postscript	`ps`
TeX dvi	`dvi`
plain text	`ascii`
HTML	`html`

Table 2 Groff output formats other than Postscript

To format a document marked up with the `me` macros to produce a `dvi` output file, its necessary to do

```
groff -Tdvi -me < example1.txt > /tmp/example1.dvi
```

and to format a manual page that would be displayable on the terminal with `cat` or `less`, it would be necessary to do:

```
groff -Tascii -man < example1.txt > /tmp/example1.asc
```

Generating output in HTML is similar (`-Thtml`) but requires additional options. The manpage for `grohtml`, read in conjunction with that of `groff`, will tell about all that.

5. `Groff` Preprocessors—Equations, Figures and Tables

As Fig.2 indicates, an article may have embedded in it figures, tables and mathematics, and it may make references to a bibliography. There are special companion programs that `groff` uses to handle these. These companion programs are known as *preprocessors*. Some of the more important of these will be described below.

In a broad sense, these preprocessors all function in a similar manner. Each type of entity—a picture or a table or mathematics—is described in its own language and it is the

appropriate preprocessor's job to translate that language into pure `groff` requests. The resulting file is then processed by `groff` as usual.

5.1. The `Pic` Preprocessor

The `pic` preprocessor translates picture descriptions in the `pic` language into equivalent `groff` requests. `Pic` looks in the `groff` source for blocks of text enclosed between `.PS` and `.PE` macro pairs and attempts to process those. For instance the following figure:

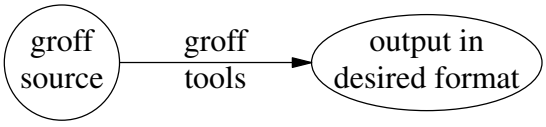

Fig.3

was produced with this `pic` code:

```
.PS
circle rad 0.3i "groff" "source"
arrow 1.0i "groff" "tools"
ellipse wid 1.2i "output in" "desired format"
.PE
```

`Pic` source for Fig.3

Here we have used the elementary object types `arrow`, `circle` and `ellipse`. These, together with the `line`, `box`, `arc` and `spline`, are the most useful object types. `Pic` object types have default sizes which may be overridden. For instance the default length of a line or an arrow is 0.5 inches. In the example the default length of the arrow has been overridden by specifying a length of 1.0 inches immediately after the keyword `arrow`. The example also shows how to override the default radius of a circle (0.5 inches) and the default width (0.75 inches) of an ellipse.

By default, `pic` puts each new object to the immediate right of the last one. This behaviour can be altered by specifying directions such as `up`, `down` or `right`, like this:

```
.PS
circle rad 0.3i "\f[CW]groff\fP" "source"
arrow 1.2i "\f[CW]pic\fP, \f[CW]tbl\fP" \
           "\f[CW]eqn\fP, \f[CW]refer\fP"
circle rad 0.4i "\f[CW]groff\fP" "+" "macros"
down; arrow 0.7i at last circle.s
box invis with .w at last arrow .c+(0.05i,0i) \
          "\f[CW]gtroff +\fP" "macros e.g." \
          "\f[CW]ms\fP or \f[CW]me\fP"
circle at last circle + (0.0i, -1.35i) "ditroff"
left; arrow 0.8i at last circle.w "device" "driver"
circle rad 0.3i "output"
.PE
```

`Pic` source for Fig.4

The `down` and `left` modifiers tell `pic` which way to move. Each object has associated with it certain reference points. Closed objects such as boxes and circles have the compass points `.e`, `.w`, `.n`, `.s`, `.ne`, `.nw`, `.se`, `.sw` and the geometric centre, `.c`, associated with them. Open objects such as lines, arcs and splines have the reference points `start`, `centre` and `end`. `Centre` refers to the middle of the arc length between the `start` and `end` points. And, finally, two or more `pic` statements may be written on one line if they are separated by semicolons.

Objects may be referred to by their relative order of occurrence (`last arrow` or `2nd last box`, for example). The sizes of objects may be changed. For example, `line 1.2i` means a line 1.2 inches long and `circle rad 0.5i` means a circle of radius 0.5 inches. And objects may be rendered invisible by the `invis` attribute, as in `box invis`. The above `pic` code makes the following picture:

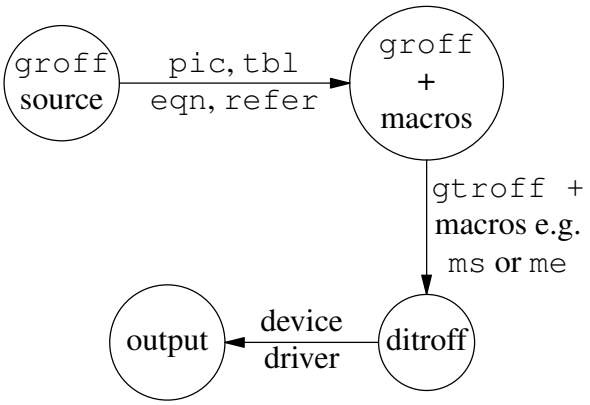

Fig.4

Here is another example, with

```
.PS
circle "circle" "default"
circle dashed radius 0.5i "circle" "rad 0.5i" "dashed"
ellipse "ellipse" "default"
ellipse ht 1.0i wid 0.8i "ellipse" "ht 1.0i" "wid 0.8i"
.PE
```

`Pic` source for Fig.5

producing

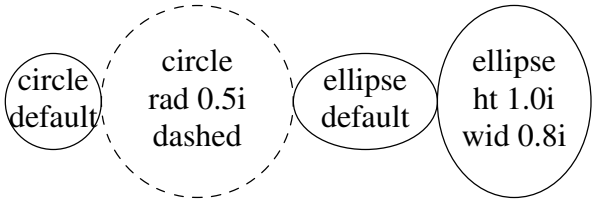

Fig.5

Note that a circle may have the attribute `dashed` specified, leading it to be drawn in a dashed line. Lines, arrows and boxes may also be dashed. Ellipses and splines may *not* be dashed.

Objects are *always* positioned one after another with extremities touching. This may not always have the desired effect. For example, the above example certainly looks better this when drawn this way:

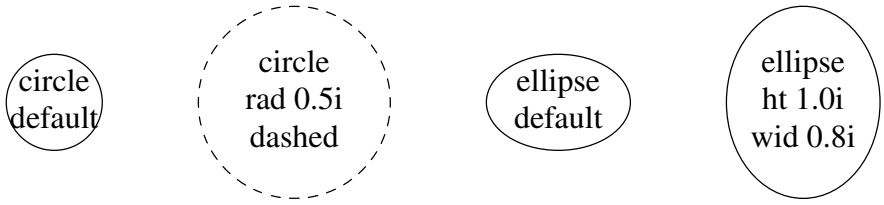

Fig.6

where 'invisible' lines of the default length were inserted between the objects as you may see below:

```
.PS
circle "circle" "default"
line invis
circle dashed radius 0.5i "circle" "rad 0.5i" "dashed"
line invis
ellipse "ellipse" "default"
line invis
ellipse ht 1.0i wid 0.8i "ellipse" "ht 1.0i" "wid 0.8i"
.PE
```

`Pic` source for Fig.6

Finally, here is a larger example, showing how `groff` source files are processed:

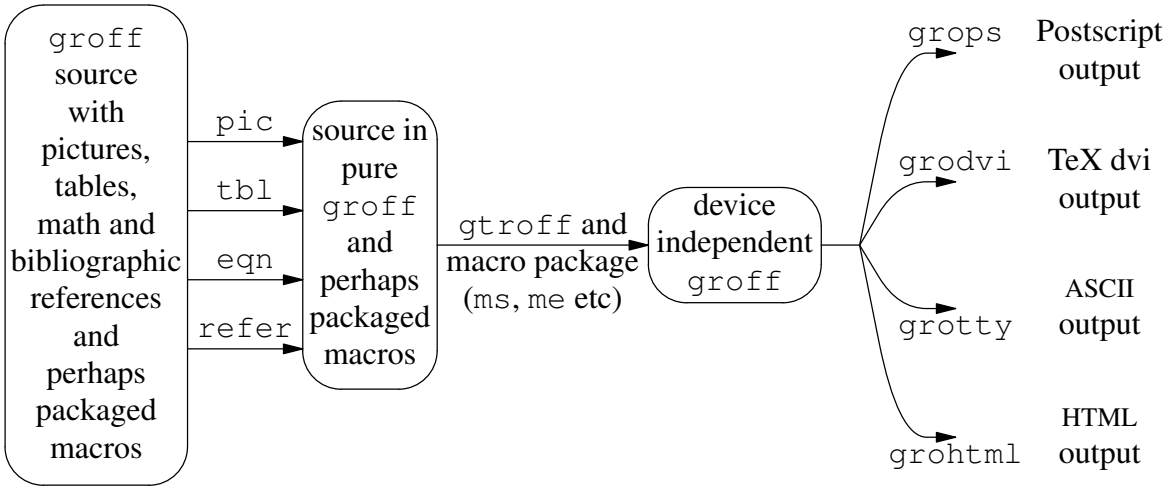

Fig.7

The boxes show the state of the document at any stage and the arrows the processes that transform the document from one form to the next. `Grops`, `grodvi`, `grotty` and `grohtml` are some of the device drivers available with `groff`. These device drivers transform the "device independent" output into a form suitable for display with a particular device. Here is the `pic` code that produced Fig.7:

```
.PS
A:box rad 0.2i ht 2.5i wid 0.95i "\f[CW]groff\fP" \
                    "source" "with" "pictures," \
                    "tables," "math and" "bibliographic" \
                    "references" \
                    "and" "perhaps" "packaged" "macros"
s=0.6i
arrow s at (A.e.x,A.e.y+0.54i) "\f[CW]pic\fP" ""
arrow s at (A.e.x,A.e.y+0.18i) "\f[CW]tbl\fP" ""
arrow s at (A.e.x,A.e.y-0.18i) "\f[CW]eqn\fP" ""
arrow s at (A.e.x,A.e.y-0.54i) "\f[CW]refer\fP" ""
box rad 0.2i ht 1.5i wid 0.7i with .w at (A.e.x+s, A.e.y) \
                    "source in" "pure" "\f[CW]groff\fP" \
                    "and" "perhaps" "packaged" "macros"
arrow 1.1i at last box.e "" "\f[CW]gtroff\fP and" \
                        "macro package" \
                        "(\f[CW]ms\fP, \f[CW]me\fP etc)"
box rad 0.2i ht 0.6i wid 0.9i "device" "independent" "\f[CW]groff\fP"
B:line 0.2i
spline -> from B.e to (B.e.x+0.2i,B.e.y+1.0i) \
              to (B.e.x+0.5i,B.e.y+1.0i)
box invis at last spline.end "\f[CW]grops\fP" ""
box invis "Postscript" "output"
spline -> from B.e to (B.e.x+0.2i,B.e.y+0.33i) \
              to (B.e.x+0.5i,B.e.y+0.33i)
box invis at last spline.end "\f[CW]grodvi\fP" ""
box invis "TeX dvi" "output"
spline -> from B.e to (B.e.x+0.2i,B.e.y-0.33i) \
              to (B.e.x+0.5i,B.e.y-0.33i)
box invis at last spline.end "" "\f[CW]grotty\fP"
box invis "\s-2ASCII\s0" "output"
spline -> from B.e to (B.e.x+0.2i,B.e.y-1.0i) \
              to (B.e.x+0.5i,B.e.y-1.0i)
box invis at last spline.end "" "\f[CW]grohtml\fP"
box invis "\s-2HTML\s0" "output"
.PE
```

`Pic` source for Fig.7

The `rad` attribute of a `box` object is the radius of the box corners. The default value is 0. An interesting usage here is of a `pic` *variable.* The variable `s` has been assigned a value of 0.6 inches. Later, wherever `s` occurs, it is replaced with 0.6i.

A complete description of `pic` would require much space. Fortunately, excellent descriptions already exist. The original `pic` is described by Kernighan[14] and GNU `pic` is described by Raymond.[15] The latter has some enhancements, the most notable of which is the ability to use colour. These articles are essential reading for anyone desiring to use `pic`. Raymond's article is available as `groff` source alongwith the `groff` source

tarball. Stevens[16, 17] describes the `pic` macros he used to write his books as well as a way of converting `pic` to HTML.

The eponymous `gnuplot` and `xfig` programs can save pictures in the `pic` format (although, sadly, they cannot read `pic` files). If the file `plotfile.pic` contains the output from `gnuplot` saved in the `pic` format, then the picture may be included in the document by either physically inserting the file or, even better, by using the `copy` statement:

```
.PS
copy "plotfile.pic"
.PE
```

The advantage of the second method is that the file may be modified by `gnuplot` without the need for changing the `groff` source. In this method the `.PS–.PE` macros occur twice: once surrounding the `copy` statement, and once inside the file `plotfile.pic`, surrounding the `pic` statements generated by `gnuplot`. (The latter pair are produced by `gnuplot`).

If a document uses `pic`, this is how its source, `example.ps`, is processed:

```
groff -p -ms < example.ms > /tmp/example.ps
```

5.2. The `Tbl` Preprocessor

`Tbl` is a language that describes the layout of tables. A description of a table in this language is converted into pure `groff` by the `tbl` preprocessor. This section gives a brief introduction on using `tbl`. A complete description is given by Cherry and Lesk.[18]

The description of a table has several conceptually distinct parts. In broad terms these are:

- The delimiting macros `.TS and .TE,`
- A one-line set of *global* attributes for the entire table,
- Formatting information
  There may be one or more rows of these. If there is only one format row, it applies to all data rows. If there is more than one format row, these are successively applied to the rows of data beginning with the first. When the last format row is reached, it applies to all the remaining rows of data. That is, if there are *n* format rows specified, then the first *n*-1 of them apply to the first *n*-1 rows of data in sequence, and the n^{th} format row applies to all the remaining data rows.
- The data itself
  arranged in rows, with the columns separated by a delimiter that is declared in the global attributes section.

Table 2 (§ 4.5) was produced with the following description:

```
.TS
box,tab(:),centre;
c|c
c|c
l|lfCW.
Output:Device
Type:Name
_
Postscript:ps
TeX dvi:dvi
plain text:ascii
HTML:html
.TE
```

and this is what it all means:

- There are three global attributes:

  `box` this says that a box is to be drawn around the outer boundary of the table,
  `tab(:)` this says that the column separator is the colon character (`:`),
  `centre` this tells `groff` to centre the table across the page.

  Note how the global attributes list is terminated with a semicolon (`;`).

- There are three format rows:

  ```
  c|c
  c|c
  l|lfCW.
  ```

  These say that there are two columns in the table. The `c|c` in the first format row say that data items in the first data row are to be centred in their columns; the `c|c` in the second format row say the same thing about the data items in the second row; and the `l|lfCW` in the third and final format row say that data items in the third *and subsequent* rows are to be left aligned in their respective columns; also, the items in the second column are to be printed in constant width characters.

  Note how the final format row is terminated with a dot.

  There are other format specifiers besides `l` and `c`. In particular, the `n` data format is used when data items in a column are numeric. If all items in the column are integers, they will be right-aligned. If they include decimal points, then they will be arranged so that all decimal points in the column are vertically aligned.

- The `|` character separating the two format specifiers of each format row tells `tbl` to draw a vertical line between the columns. If no vertical line between columns was desired, the format rows would have been

  ```
  c:c
  c:c
  l:lfCW.
  ```

  that is, the separator would have been the tab character as specified in the global attributes.

- The _ (underscore) character appearing all by itself
  on the row immediately following the first data row is not part of the data. Instead, it signals `tbl` that a *single horizontal line* is to be drawn across the width of the table at this point (in this case after the first row of data).

Here is another table

```
.TS
centre,tab(:),box;
c:c
l:l.
Preprocessor:Used to typeset
=
\f[CW]pic\fP:Line drawings
\f[CW]tbl\fP:Tables
\f[CW]eqn\fP:Mathematics
\f[CW]refer\fP:Bibliographies
:and references
.TE
```

`Tbl` source for Table 3

that, on formatting, looks like this:

Preprocessor	Used to typeset
`pic`	Line drawings
`tbl`	Tables
`eqn`	Mathematics
`refer`	Bibliographies and references

Table 3

Two points may be noted:

- The = (equals) character appearing all by itself
  on a row is like the _ of the last example but calls for a *double* horizontal line.
- In the last data row, there is no data in the first column.

And now a slightly more complicated example:

Macro packages available with groff	
Package	Comments
ms	Origin: Bell Labs. First macro package to gain wide acceptance. Still widely used. Good for short documents.
me	Origin: UC Berkeley. Not as popular as ms. But more flexible and better suited to large documentation projects. Also, better documented.
mm	A "better" ms. Incorporates some features of me.
man	Used to format on-line manual pages on Linux systems. Perhaps the sole reason why groff is alive today!
mom	Under development. Seems to hold great potential as a macro package. Works with groff v1.18 or later.

Table 4 Macro packages for groff

that was produced with the following description:

```
.TS
centre,box,tab(:);
c:s
l:c
lfCW:lw(4i).
T{
Macro packages available with
groff
T}
_
Package:Comments
=
ms:T{
Origin: Bell Labs.
First macro package to gain wide acceptance.
Still widely used.
Good for short documents.
T}
_
me:T{
Origin: UC Berkeley. Not as popular as
.CW ms .
But more flexible and better suited to
large documentation projects.
Also, better documented.
T}
_
mm:T{
A "better"
.CW ms .
```

```
Incorporates some features of
.CW me .
T}
_
man:T{
Used to format on-line manual pages on Linux systems.
Perhaps the sole reason why
.CW groff
is alive today!
T}
_
mom:T{
Under development. Seems to hold great potential as
a macro package. Works with
.CW groff
v1.18 or later.
T}
.TE
```

Tbl source for Table 4

The main changes from the earlier example are

- The first format row has an `s` as the format specifier in the second column.
  This means that the first data field of the first data row is to *span across* the second field as well. That is why the "Macro packages available with `groff`" entry appears centred across the overall width of the table.

  The second format specifier in the third format row, `lw(4i)` is the `l` format specifier augmented with a "column width" specifier. Here the width of this column is specified to be 4 inches. This is the *minimum* width the column will have. If necessary, the column width will automatically expand to accommodate the widest data item in this column.

- When it becomes necessary to have a data field occupy more than one line in the table description, then that data field is enclosed between `T{` and `T}` markers. The `T{` must begin immediately after a data field separator or a newline, and must be followed by a newline. The `T}` must begin immediately after a newline and must be followed by a data field separator or another newline.

And, finally, a largish example:

Components of groff and their relationship to each other			
Type	Function		
Preprocessor	Name	Description	Groff command line option
	eqn	Language for typesetting mathematics	-e
	tbl	Language for typesetting tables	-t
	pic	Language for typesetting line drawings. Xfig and gnuplot can export files in the pic format	-p
Macro package	Name	Description	Groff command line option
	ms	Adequate for letters and articles but not for books or large scale projects.	-ms
	me	More flexible. Better suited to large projects than ms. Better documented.	-me
	mm	A "better" ms. Incorporates features of me.	-mm
	man	Used to format the on-line manual pages on Linux systems.	-man
	mom	Still under development. Holds great potential. Works with groff v1.18 or later.	-mom
Groff language processor	Name	Description	Groff command line option
	gtroff	Converts groff source to "device independent troff" or ditroff.	Always invoked by the groff command.
Device driver	Name	Description	Groff command line option
	grops	Converts ditroff to Postscript. Default driver for groff. With this, groff can include Postscript files and/or execute Postscript commands in the source file.	-Tps
	grodvi	Converts ditroff to TeX dvi format.	-Tdvi
	grotty	Converts ditroff to ASCII.	-Tascii
	grohtml	Converts ditroff to HTML. Still in alpha stages.	-Thtml

Table 5 Components of the groff system

which is produced by:

```
.TS
centre,box,tab(:);
c:s:s:s
c|c:s:s.
T{
Components of
```

```
.CW groff
and their relationship to each other
T}
_
Type:Function
=
.T&
c|c|cw(3.0i)|c
c|^|^|c
c|cfCW|lw(3.0i)|c
c|cfCW|lw(3.0i)|cfCW
^|cfCW|lw(3.0i)|cfCW.
:Name:Description:T{
.CW Groff
command
T}
:::line option
:_:_:_
Preprocessor:eqn:T{
Language for typesetting mathematics
T}:-e
:tbl:T{
Language for typesetting tables
T}:-t
:pic:T{
Language for typesetting line drawings.
.CW Xfig
and
.CW gnuplot
can export files in the
.CW pic
format
T}:-p
=
.T&
c|c|cw(3.0i)|c
c|^|^|c
c|cfCW|lw(3.0i)|c
c|cfCW|lw(3.0i)|cfCW
^|cfCW|lw(3.0i)|cfCW.
:Name:Description:T{
.CW Groff
command
T}
:::line option
:_:_:_
T{
```

```
.nf
Macro
package
.fi
T}:ms:T{
Adequate for letters and articles but
not for books or large scale projects.
T}:-ms
:me:T{
More flexible. Better suited to large projects than
.CW ms .
Better documented.
T}:-me
:mm:T{
A "better"
.CW ms .
Incorporates features of
.CW me .
T}:-mm
:man:T{
Used to format the on-line manual pages on Linux systems.
T}:-man
:mom:T{
Still under development. Holds great potential. Works with
.CW groff
v1.18 or later.
T}:-mom
=
.T&
c|c|cw(3.0i)|c
c|^|^|c
c|cfCW|lw(3.0i)|c
c|cfCW|lw(3.0i)|cfCW.
:Name:Description:T{
.CW Groff
command
T}
:::line option
:_:_:_
T{
.nf
.CW Groff
language
processor
T}:gtroff:T{
Converts
.CW groff
```

```
source to "device independent
.CW troff """
or
.CW ditroff .
T}:T{
.R
Always invoked by the
.CW groff
command.
T}
=
.T&
c|c|cw(3.0i)|c
c|^|^|c
c|cfCW|lw(3.0i)|c
c|cfCW|lw(3.0i)|cfCW
^|cfCW|lw(3.0i)|cfCW.
:Name:Description:T{
.CW Groff
command
T}
:::line option
:_:_:_
T{
.nf
Device
driver
T}:grops:T{
Converts
.CW ditroff
to Postscript.
Default driver for
.CW groff .
With this,
.CW groff
can include Postscript files and/or execute Postscript
commands in the source file.
T}:-Tps
:grodvi:T{
Converts
.CW ditroff
to TeX dvi format.
T}:-Tdvi
:grotty:T{
Converts
.CW ditroff
to
```

```
.sm ASCII .
T}:-Tascii
:grohtml:T{
Converts
.CW ditroff
to
.sm HTML .
Still in alpha stages.
T}:-Thtml
.TE
```

Tbl source for Table 5

and is a significantly larger `tbl` program than any we have seen so far. Of course, the table it lays out is also fairly complicated. A couple of new things are introduced here:

- The `.T&` macro
  When used in a table, this macro allows the formats of the succeeding columns to be re-specified. It does not, however, allow the number of columns to be changed.
- The `^` format descriptor
  Just as the `s` format descriptor causes "horizontal spanning" of a data field, so the `^` format specifier causes *vertical spanning* a column. This is what happens with all the items in the "Type" column, such as "Preprocessor" and "Macro Package". Observe how they are vertically centred relative to the items on their immediate right.

If a document source makes use of `tbl` constructs, this is how it is processed by `groff`:

```
groff -t -ms < example.ms > /tmp/example.ps
```

where the document source is assumed to be in the file `example.ms`.

5.3. The `Eqn` Preprocessor

The `eqn` language describes the typesetting of mathematical symbols. It is simpler than either `tbl` or `pic`, because a simple language is sufficient to describe mathematics. If you are not convinced, here's an example to convince you. The trigonometrical identity, $\sin(\alpha + \beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$, is written in `eqn` as

```
%sin ( alpha + beta ) =
sin alpha cos beta + cos alpha sin beta%
```

The entire `eqn` construct is enclosed within the delimiters `%%` to distinguish it from the surrounding text. This is an example of *inline* `eqn` usage. The characters `%%` that are used to distinguish the `eqn` construct from the surrounding text are so defined by telling `eqn`:

```
.EQ
delim %%
.EN
```

These delimiting characters may be redefined.

Greek letters, as you can see, are spelt out in English. By this rule, θ is `%theta%` and δ is `%delta%`; Θ is `%THETA%` and Δ is `%DELTA%`. Spaces have meaning in an `eqn` statement. Thus, `%sin ( alpha + beta )%` produces $\sin(\alpha + \beta)$, whereas `%sin(alpha+beta)%` produces *sin*(*alpha* + *beta*), not quite what was intended.

The mean of a population, $x_1 \cdots x_n$, of size n :

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \qquad (1)$$

is produced by

```
.EQ (1)
x bar = 1 over n { sum from i=1 to n { x sub i } }
.EN
```

and its standard deviation

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2} \qquad (2)$$

by

```
.EQ (2)
sigma = sqrt { 1 over n }
{ sum from i=1 to n { ( x sub i - x bar ) sup 2 } } }
.EN
```

There are several points to note:

- The braces `{` and `}`
  play the same role as they do in mathematics—they "bind" several items together so these items behave as a single entity.
- Any string passed as an argument to the `.EQ` macro
  appears as the "equation number" on the right margin.
- The keywords `sub` and `sup`
  denote subscripts and superscripts, respectively.
- Fractions are produced
  with the `over` keyword separating the numerator from the denominator.

More complicated expressions, such as

$$\cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \cdots}}}} = 0.6180\cdots$$

may be produced by judicious use of braces:

```
.EQ
1 over
{ 1 + 1 over
  { 1 + 1 over
    { 1 + 1 over
      { 1 + ...
      }
    }
  }
}
= 0.6180 ...
.EN
```

If a document source makes use of `eqn` constructs, this is how it is processed by `groff`:

```
groff -e -ms < example.ms > /tmp/example.ps
```

where the document source is assumed to be in the file `example.ms`.

To learn more about `eqn` (and there *is* more to learn, despite its simplicity), read the articles by Kernighan and Cherry.[19, 20]

5.4. Documents that use `Pic`, `Tbl` and `Eqn`

Formatting documents that use all of `pic`, `tbl` and `eqn` is easy:

```
groff -p -e -t -ms < example.ms > /tmp/example.ps
```

does it! The first three options to `groff` may even be combined to get an abbreviated form:

```
groff -pet -ms < example.ms > /tmp/example.ps
```

5.5. `Refer`, `Chem` and Other Preprocessors

In addition to the preprocessors described above, there are other, perhaps less commonly used ones. `Refer` is a preprocessor to manage bibliographical references within a document. Most commonly, the bibliographic information is held in a plain-text file in a carefully defined format, and each item is identified by a unique keyword. Within the text, a reference to the keyword inserts an appropriate citation of the item in question. The bibliographic references in this article were produced with `refer`. I was unable to locate a public source for the article by Bill Tuthill entitled *Refer – A Bibliography System* that is dedicated to documenting `refer`. The article by Lesk[21] is the work upon which `refer` is based. But it is only partly devoted to `refer`.

`Chem` is another "small language" that is very useful in drawing molecular structures of chemicals. It is described by Bentley, Jelinski and Kernighan.[22]

Bentley and Kernighan also describe a tool for printing indexes.[23] This tool is more in the nature of a 'co-processor' for `groff`, since it works in parallel with the latter to index marked terms in a document.

`Chem` and `index` were originally written for UNIX `troff` They have not been rewritten for GNU `groff`. Hence they are not part of the `groff` distribution although they work with it.

`Style` and `diction` are two unique programs from UNIX `troff`, whuch have been recreated as GNU `diction` by Michael Haardt.[24] This is is a quote from the homepage:

> `Diction` identifies wordy and commonly misused phrases. `Style` analyses surface characteristics of a document, including sentence length and other readability measures. These programs cannot help you structure a document well, but they can help to avoid poor wording and compare the readability (not the understandability!) of your documents with others.

6. Exploring `Groff`'s Capabilities Further

The capabilities of `groff` that we have learnt about so far are useful in producing conventional documents: letters, articles like this one, or even books (although not without a lot of bookkeeping!). All this would, of course, be in black-and-white, literally.

In the following sections we shall explore `groff`'s features and capabilities in the direction of colour and images. These capabilities are due to `groff`'s ability to execute Postscript instructions and are available *only* with the Postscript output driver `ps`. (Let me state at the outset that I *don't know* the Postscript language. I cannot explain *why* or *how* the things in the following sections work. All I know is that they do.)

6.1. `Groff` and Colour

The manpage for `grops` describes a mechanism by which `groff` is able to execute Postscript commands. Let us define some `groff` strings, as follows:

```
.ds RED     \X'ps: exec 1 0 0 setrgbcolor'
.ds GREEN   \X'ps: exec 0 1 0 setrgbcolor'
.ds BLUE    \X'ps: exec 0 0 1 setrgbcolor'
.ds YELLOW  \X'ps: exec 1 1 0 setrgbcolor'
.ds MAGENTA \X'ps: exec 1 0 1 setrgbcolor'
.ds CYAN    \X'ps: exec 0 1 1 setrgbcolor'
.ds BLACK   \X'ps: exec 0 0 0 setrgbcolor'
.ds WHITE   \X'ps: exec 1 1 1 setrgbcolor'
```

Then, an input of the form

```
\*[RED]This is RED
\*[GREEN]This is GREEN
\*[BLUE]This is BLUE
\*[YELLOW]This is YELLOW
\*[MAGENTA]This is MAGENTA
\*[CYAN]This is CYAN
\*[WHITE]This is WHITE
\*[BLACK]This is BLACK
```

produces the following list of *coloured* colour names:

This is RED
This is GREEN
This is BLUE
This is YELLOW
This is MAGENTA
This is CYAN

This is BLACK

(The line in white will most likely not be visible because the white foreground colour merges with the background.)

Let me explain the `groff` part of what's happening here. The `.ds` request defines a string whose name is the first argument and whose value is the second argument. For example, the definition:

```
.ds RED     \X'ps: exec 1 0 0 setrgbcolor'
```

defines a string named `RED` to have the value `\X'ps: exec 1 0 0 setrgbcolor'`. Later, usage of the form

```
\*[RED]This is red
```

causes the value of the string `RED` to be *interpolated* (that is the proper `groff` term) just before the `T` in `This`. This string, which is a Postscript statement to set the foreground colour to red, is then executed by `grohtml`'s internal Postscript interpreter.

Coupled with the ability to change fonts and sizes (see § 3.6), this makes it possible to produce presentation slides or simple coloured posters.

6.2. Importing a File Containing Postscript into a `groff` document source

Let's say you've just been to Darjeeling and got this most wonderful picture of a sunrise on Tiger Hill with your new digital camera. You're making a writeup about the trip (using `groff`, of course) and you would like to include this picture in the document, which will be printed by one of the glossy magazines. Its possible, provided you convert the picture into Postscript.

According to the `grops` manpage, a statement of the form

```
.PSPIC [L|R|I n] file [width [height]]
```

inserted into the appropriate place in the document source will cause the Postscript graphic to be imported. Here `file` is the name of the file containing the illustration; `width` and `height` give the desired width and height (the default unit is inches) of the graphic. This macro will scale the graphic uniformly in the x and y directions so that it is no more than `width` wide and `height` high. By default, the graphic will be horizontally centered. The `L` and `R` cause the graphic to be left-aligned and right-aligned respectively. The `I` option causes the graphic to be indented by `n` units (inches, by default).

So, if your picture is in a file called `tigerhill.ps` then simply saying

```
.PSPIC tigerhill.ps
```

will import the picture and centre it on the page.

6.3. Output in Landscape Mode

`Groff` normally produces output in portrait mode. There are occasions, such as when making slides for a presentation, when output is desired in landscape mode. The Postscript device driver `grops` accepts an argument `-l` that tells it to produce output in landscape mode. Since `grops` is never invoked directly, `groff` has a mechanism of passing parameters to it (and to other downstream programs). In the command

```
groff -ms -P-l < slides.ms > slides.ps
```

the 'compound option' `-P-l` is the mechanism by which `groff` passes the `-l` option to `grops`.

7. Innovative Uses of `Groff`

If UNIX is the "programmer's operating system", then surely `groff` is the "programmer's text processor". There are chiefly two reasons for this:

Firstly

> `groff`, like its predecessor `troff`, is itself programmable. The macros that we have been using are actually short programs whose statements are pure `groff` requests.

And secondly,

> since the document source is in plain text, all the powerful UNIX utilities for operating on plain text files (sorting, searching, joining, merging, splitting and non-interactive editing, for example) can be invoked to manipulate these source files before handing them over to `groff`.

Thus it is possible to look for mis-spelt words using the standard UNIX spell checker, to count the number of words in a document, to keep track of changes in a large document as it evolves, to look for occurrences of "variant phrases" (occurrences of "semirigid", "semi-rigid" and "semi rigid" in the same article, for example) and so on.

With some programming using standard UNIX scripting languages such as `bash`, `sed` and `awk` (or `perl`) it is possible to create many preprocessors beyond the standard ones described earlier. One such preprocessor could be a mailmerge system. Another could be a pre-processor to automatically number equations, tables and figures and update references to them.

In the next couple of sections I shall outline some ideas that will substantiate the above statements. Many more innovations are possible, the limit being your imagination and your mastery over the UNIX programming environment.

7.1. Producing Presentation Slides With `Groff`

We now have all the background required to produce slides for making presentations with `groff`. These are going to be of the no-frills type: spartan and to the point. But that's what is required of a presentation slide, right?

We shall use the definitions of the colour names of § 6.1 here. With these, the `groff` source

```
.bp
\s[24]\*[BLUE]Linux is a great piece of software
.sp 0.5i
.ce
but ...
.sp 0.5i
\*[RED]How good is it for scientific applications?
.sp 0.5i
```

produces the Postscript output:

Linux is a great piece of software

but ...

How good is it for scientific applications?

The markup tag `.bp` tells `groff` to begin a new page; and the escape sequence `\s[24]` tells it to change the font size to 24 point.

Of course we are not limited to letters of a maximum 24 point size. Here is something in 144 point (about 2 inches high):

Hi!

When using characters of large size it is usually convenient to produce the output in landscape mode, something we already discussed in § 6.3. This is the formatting command again:

```
groff -ms -P-l < slides.ms > slides.ps
```

We may sometimes like to draw a box around a slide to emphasize the border and hold the viewers' attention to the display material within it. You can try these `ms` macros to draw a box although, be warned, the result may not always be what you want:

Linux is a great piece of software

but ...

How good is it for scientific applications?

Here is how it was produced:

```
.bp
.B1
.sp 0.5i
\s[24]\*[BLUE]Linux is a great piece of software
.sp 0.5i
.ce
but ...
.sp 0.5i
\*[RED]How good is it for scientific applications?
.sp 0.5i
\*[GREEN]
.B2
```

The `.B1` and `.B2` macros are responsible for the box. Note that if you want to specify the colour of the line that the box will be drawn in, you must do that just before the `.B2`.

How do you actually project these slides? If you want to go the old-fashioned route and project them with an overhead projector then you have to print the Postscript file on special transparency film with a colour inkjet printer. If you eschew colour then you can save money and print the slides out on ordinary transparency film using a laser printer. In any case your slides become "frozen" once you commit them to film.

The least expensive and also the most flexible way of projection is with an electronic projector (provided of course you don't have to pay for the projector!). The greatest advantage is that you can modify your slides at will, since they are in software. As far as projecting them goes, if the projector is connected to a Linux system then you can work directly with the Postscript file, using `ghostview` to display it and adjust the size so that the window fills up as much of the screen as possible. (To do this most effectively you may have to try out various screen resolutions; an A4 page in landscape mode seems to fit 800×600 screen resolutions best.) With the proper options, you can cut down on the bells and whistles that `ghostview` displays on its screen borders.

For platform-independent projection capability, however, you should use a web browser to show your slides. Browsers on any platform are able to display files in `jpeg` and `gif` formats. `Ghostscript` can convert Postscript files into a large number of formats, including `jpeg`, `pnm` and `pbm` (but not `gif`, presumably because of patent restrictions). `Pbm` can again be converted to many formats, including `gif`, through the `NetPBM` utilities. Here's a little shell script, called `ps2jpg`, to invoke `ghostscript` to convert a Postscript file into `jpeg` files, one file per page (or slide):

```
#!/bin/sh

    gs -dSAFER -dNOPAUSE -sDEVICE=jpeg -sPAPERSIZE=a4 \
       -sOutputFile=${1}%02d.jpg \
    ${1}.ps << END
    ^D
END
```

Suppose your Postscript file is named `slides.ps` and consists of four A4 pages. Then, saying

```
ps2jpg slides
```

(note that we omit the `.ps` extension from the name of the Postscript file) will produce *four* `jpeg` files named `slides00.jpg` to `slides03.jpg`. The simplest method of displaying these is to load them in sequence into the browser *before* the presentation, then go back to the first one. The browser could be kept minimised until such time that the presentation begins. (A lively screensaver helps during the hiatus!) With this scheme, clicking on the browser's 'Forward' button (with a remote mouse, perhaps) takes you to the next slide.

7.2. `Groff` as a 'Back-End' Processor

Let's say we're running a small business and we have an on-line data base to track customer orders. We pride ourselves on the promptness of our service so, every morning, one of the first things we do is find out what orders are outstanding. The query

```
find all order with (order_date < todays_date)
```

(where `todays_date` is automatically set by the data base system) brings forth the response

```
342 2003-5-29 Hari Ghosh
97 2003-5-28 Kenaram Sau
43 2003-6-1 Chandidas Chingri
403 2003-5-30 Becharam Chatterjee
```

the first column being the order number and the second column the order date. Methodical organization that we are, we would like to have this information neatly formatted and printed, with copies sent to the manager and the service department. We therefore save this information to a file named `ord.txt`:

```
find all order with (order_date < todays_date) save "ord.txt"
```

and invoke a `bash` script named `printord` with two arguments:

```
printord ord.txt 2
```

upon which two copies of the above information, neatly formatted, appear on the printer. What does this `bash` script named `printord` contain? Here it is:

```
#!/bin/bash

if [ $# -ne 2 ]
then
    echo "Usage: $0 order-file number-of-copies"

    exit 1;
fi

# we check if the named file exists and is readable

if [ ! -r $1 ]
then
    echo "$0: File $1 not-existent or not readable. Exiting"

    exit 2;
fi

# and that the number of copies seems reasonable

if [ $2 -ge 1 -a $2 -le 3 ]
then
    echo "$0: printing $2 copies"
else
    echo "$0: too many copies - ${2}. Exiting"

    exit 3;
fi

# we make two temporary files with unique names:

TMPNAM=/tmp/${$}.order
PS_NAM=/tmp/${$}.ps

# we also get today's day and date from the system in a
# custom format:

DATE=$(date +"%A %d-%m-%Y")

# and now we make our table:
```

```
1720        echo ".ps 12" >> $TMPNAM
            echo ".vs 14" >> $TMPNAM
            echo ".ce" >> $TMPNAM
            echo "\fBOutstanding orders on $DATE\fP" >> $TMPNAM
            echo ".TS" >> $TMPNAM
            echo 'tab (:), center, box;' >> $TMPNAM
            echo "c|c|c" >> $TMPNAM
            echo "c|c|c" >> $TMPNAM
            echo "n|l|l." >> $TMPNAM
            echo "Order:Order:Customer" >>$TMPNAM
1730        echo "Number:Date:Name" >>$TMPNAM
            echo "=" >> $TMPNAM

            # The rest of the table, that is, the data rows.
            # we assume that a customer's name has at most four words
            awk '{ print $1":"$2":"$3" "$4" "$5" "$6 }' < $1 >> $TMPNAM

            # the table ends
            echo ".TE" >> $TMPNAM

            # and format it with groff -t

            groff -t < $TMPNAM > $PS_NAM

            # and print the requisite number of copies

1740        lpr -#$2 $PS_NAM

            # and, finally, remove the temporary files

            rm -f $TMPNAM $PS_NAM
```

and this is the table that is produced:

Outstanding orders on Tuesday 03-06-2003

Order Number	Order Date	Customer Name
342	2003-5-29	Hari Ghosh
97	2003-5-28	Kenaram Sau
43	2003-6-1	Chandidas Chingri
403	2003-5-30	Becharam Chatterjee

In a similar manner, `groff` can be used as a back end processor to produce diagrams, graphs, flow-charts and form letters. With a little programming using standard UNIX tools such as `bash`, `awk` (or `perl`) and `sed`, it is possible to create a mailmerge system for merging addresses (and, if necessary, salutations) from an address data base with form letters.

7.3. `Groff` and "Wysiwyg"

Yes! It's possible. I use it all the time. And I'll show you how. It depends partly on the text editor and partly on the Postscript viewer. In the next few paragraphs I shall use `vim` as the editor and `ghostview` as the Postscript viewer to illustrate the procedure.

Let's say I am editing our example file, `example.ms`. This file uses `pic`, `tbl` and `eqn` statements. Now, from *within* `vim` I enter the following 'colon command' (that is, a `:` followed by the command):

```
:map <F1> :w^M:!groff -ms -pet < example.ms > /tmp/ex.ps^M
```

What the `map` command does is to associate the entire string on the right with the F1 function key. In this command `<F1>` is got by pressing the F1 function key; and `^M` is the "Enter" character, obtained by the keystroke sequence "Ctrl+V+M".

With the map defined and `vim` in command mode, pressing the F1 key causes `vim` to interpret the character string mapped to F1 *exactly* as if it was typed from the keyboard: the `:w` followed by "Enter" causes the file to be written to disk and the `:!` causes the rest of the string to be passed to the shell to be interpreted and executed as a shell command—that is, the `groff` command gets executed, producing the `ex.ps` Postscript file.

In order for the next step to work—and the magic to begin—you must start `vim`, define the mapping of the F1 key, and press F1 at least once. Now, either open another terminal window, or in the present one stop `vim` temporarily with `^Z`. Just to make sure, check that the file `/tmp/ex.ps` is indeed there. Then start `ghostview` in the background with the `-watch` option (and, of course, your other favourite options):

```
ghostview -watch /tmp/ex.ps &
```

The `-watch` option tells `ghostview` to watch the file it is displaying (here `/tmp/ex.ps`), and each time that file changes, to load it afresh.

Now get back to editing. If you had stopped `vim` with `^Z`, you can resume it by typing `fg` at the shell prompt. Make some changes to the file and press F1 again. Bring the `ghostview` window to the top with a mouse click, and you should see your latest changes reflected there! Now, each time you press F1 from within `vim`, the latest changes will appear in the `ghostview` window. How's that for "wysiwyg"?

This method of making "wysiwyg" work is alright but, after the initial novelty wears off and you wish to make routine use of this facility, you begin to notice at least three areas which could be improved: having to type in the `map` colon command each time you start `vim`, having to temporarily stop `vim` in order to start `ghostview` and, having to close the `ghostview` window after your work is done. Wouldn't it be nicer if these steps were automatically taken care of?

We shall take up the first problem first. At the time of starting up, `vim` can be made to read a *startup file*, like this:

```
vim -u vimstart.rc example.ms
```

where the startup file `vimstart.rc` contains option settings that change `vim`'s behaviour in certain ways. Its possible that you are already using such a startup file named, say, `.vimrc` residing in your home directory.

Now, `map` commands can be written into the startup file and `vim` will read and interpret them in the usual way. The difficulty in *this* case is that the name of the `groff` document source file is likely to change from one invocation of `vim` to another, thus making it necessary to edit the startup file every time. We would effectively be trading one irritant for another. Let us therefore call `bash` to our aid through the following shell script named, say, `grin` (for *gr*off *in*teractive):

```
#!/bin/bash

# we shall expect this script to have only one parameter,
# which will be the name of the groff source file that
# vim has to edit. # is the shell variable that contains the
# number of keyword parameters passed to this script

if [ $# -ne 1 ]
then
    echo "Usage: $0 groff-source-file"

    exit 1;
fi

# we assume that the user already has a startup file, .vimrc
# we shall create a temporary startup file using our
# process-id (returned in the shell variable $)

TMPNAM=/tmp/${$}.vimrc

# into which we shall copy the existing .vimrc from the
# user's home directory

cp ~/.vimrc $TMPNAM

# to this we shall append the map command. Note that $1 is
# the value of the first argument to this shell script,
# which is just the name of the file that vim has to edit
# and ^M is Ctrl-V-M:

echo 'map #1 :w^M:!groff -ms -pet <' "$1" '> \
     /tmp/ex.ps^M' >> $TMPNAM

# and then start vim

vim -u $TMPNAM $1

# when vim exits, we remove the temporary vim startup file

rm -f $TMPNAM
```

Now if grin is declared an executable file:

```
chmod u+x grin
```

and invoked with the groff document source file as the only parameter:

```
grin example.ms
```

then vim will start with the map command already defined.

Grin may be improved so that it automatically starts ghostscript in "watch mode" and terminates it once vim ends.

8. Books Actually Published Using Troff/Groff

Finally! The question you've been dying to ask all this while. If groff (and/or troff) is not something only of interest to geeks and nerds, if it is not a fossil, if it is alive and kicking then someone, somewhere, must be using it to write articles and books with. Who, then? Where? When?

Some answers are provided by Corderoy[25] who lists about 50 such publications, the latest of which was in the year 2001. All Bell Labs publications use troff. That includes classics like *The C Programming Language* by Kernighan and Ritchie, *The UNIX Programming Environment* by Kernighan and Pike and *The C++ Programming Language* by Stroustrup. "But that's natural, since *they* wrote troff! Those don't count. Who are the *others*?", do I hear you say? Well then, many of O'Reilly's books were set with troff, particularly *Unix Text Processing*,[4] which is *about* troff. All of the late W. Richard Stevens' books (*The TCP/IP Illustrated* and *UNIX Network Programming* series among them) were set in GNU groff. Here is what Wright and Stevens have to say about the production of their book *TCP/IP Illustrated, Volume 2: The Implementation*:

> "Camera-ready copy of the book was produced by the authors. It is only fitting that a book describing an industrial-strength software system be produced with an industrial-strength text processing system. Therefore one of the authors chose to use the groff package written by James Clark, and the other author agreed begrudgingly."

Compiler Design in C by Holub is another example. And *Computer Networks, 3 Ed.* by Tanenbaum yet another. Indeed, I can't resist quoting Tanenbaum:

> "The book was typeset in Times Roman using troff, which, after all these years, is still the only way to go. While troff is not as trendy as 'wysiwyg' systems, the reader is invited to compare the typesetting quality of this book with books produced by 'wysiwyg' systems." (1996)[25]

McKusick, Bostic, Karels and Quarterman, authors of *The Design and Implementation of the 4.4BSD Operating System* write:

> "This book was produced using James Clark's implementations of pic, tbl, eqn and groff. The index was generated by awk scripts derived from indexing programs written by Jon Bentley and Brian Kernighan. Most of the art was created with xfig."[25]

Nor are books on UNIX software the only ones produced using groff and kin. *Collins English Dictionary & Thesaurus, 2nd Ed.* and *Collins German Dictionary, 4th Ed.* were also produced using troff.[25] And how's this for a marriage of XML and groff, in the words of Erik T. Ray, author of *Learning XML (Guide to) Creating Self-Describing Data*:

> "The print version of this book was created by translating the DocBook XML markup of its source files into a set of `gtroff` macros using a filter developed at O'Reilly & Associates by Norman Walsh. Steve Talbott designed and wrote the underlying macro set on the basis of the GNU `troff -mgs` macros; Lenny Muellner adapted them to XML and implemented the book design. The GNU `groff` text formatter Version 1.11.1 was used to generate PostScript output." [25]

Finally, *Accelerated C++* by Koenig and Moo is also produced using `troff`. This book came out in the year 2001.

9. The Summing Up

In this article I have made an attempt to introduce the novice to text processing with GNU `groff`. I have tried to show that, combined with the other tools of UNIX, `groff` can handle most text processing tasks.

This article may be taken to be a 'sampler' of the text processing capabilities of `groff`. As I have hinted often enough, what I have shown is by no means all that `groff` can do.

In trying to keep this article to a reasonable length, I have had to completely forgo any attempt at discussing 'pure' `groff` and its programmability, which is at the heart of its power. However, the interested reader can always refer to the articles listed in the bibliography and referred to in the text to learn more.

If this article serves to awaken the curiosity of its readers about `groff`, I shall consider my efforts amply rewarded.

References

1. Joseph F. Ossanna and Brian W. Kernighan, *Nroff/Troff User's Manual,* `http://netlib.bell-labs.com/cm/cs/cstr/54.ps.gz`. Postscript format. Describes `troff` completely and in detail. Has few examples. Should be accompanied by Kernighan's Tutorial.
2. Brian W. Kernighan, *A TROFF Tutorial,* `http://www.kohala.com/start/troff/v7man/trofftut/troff-tut.ps`. Postscript format. Many examples.
3. Dean Provins, *Groff and Friends HOWTO,* `http://www.ucal-gary.ca/~dprovins/groff.ps.gz` (2001). Postscript format.
4. Dale Dougherty and Tim O'Reilly, *Unix Text Processing,* `http://www.oreilly.com/openbook/utp/`.
5. Peter Schaffter, `http://www.ncf.ca/~df191/mom.html`. Homepage for the `mom` macros.
6. Brian W. Kernighan and Rob Pike, "CHAPTER 9 Document Preparation" in *The UNIX Programming Environment,* Prentice Hall of India Private Limited, New Delhi (1987). Very readable introduction to `troff`. Also, readily and cheaply available in most bookshops.
7. James Clark's homepage is at `http://www.jclark.com/bio.htm`. James Clark originally wrote the `groff` suite of programs.
8. M. E. Lesk, *Using the -ms Macros with Troff and Nroff,* `http://www.kohala.com/start/troff/v7man/msmacros/msmacros.ps`. Postscript format.
9. Larry Kollar, *Using groff with the -ms macros,* `http://tylx.tri-pod.com/ms.ps.gz`.
10. Eric P. Allman, *Writing Papers With NROFF Using -me.* `Groff` source format. Available with `groff` source.
11. Eric P. Allman, *-Me Reference Manual.* `Groff` source format. Available with `groff` source.
12. Jeffrey Copeland and Jeffrey Haemer, *Jeffreys Copeland & Haemer's "Work" Columns,* `http://www.alumni.caltech.edu/~copeland/work`. Miscellaneous take-offs on UNIX and `groff`. Some of them should make interesting reading for the `groff` enthusiast.
13. Brian W. Kernighan and Rob Pike, "CHAPTER 3 Using the Shell" in *The UNIX Programming Environment,* p. 94, Prentice Hall of India Private Limited, New Delhi (1987).
14. Brian W. Kernighan, *PIC—A Graphics Language for Typesetting (Revised User Manual),* `http://netlib.bell-labs.com/cm/cs/cstr/116.ps.gz`. Postscript format.
15. Eric S Raymond, *Making Pictures with GNU PIC,* `http://www.kohala.com/start/troff/gpic.raymond.ps`. Postscript format.

16. W. Richard Stevens, *Examples of* `pic` *Macros.,* `http://www.kohala.com/start/troff/troff.html. The pic` macros that Stevens used to write his books such as *UNIX Network Programming* and *TCP/IP Illustrated*. Postscript format.

17. W. Richard Stevens, *Turning PIC Into HTML,* `http://www.kohala.com/start/troff/troff.html`. Programs and scripts to convert `pic` into HTML. Postscript format.

18. Lorinda L. Cherry and M. E. Lesk, *Tbl—A Program to Format Tables,* `http://cm.bell-labs.com/cm/cs/doc/76/tbl.ps.gz`. Postscript format.

19. Brian W. Kernighan and Lorinda L. Cherry, *A System for Typesetting Mathematics,* `www.kohala.com/start/troff/v7man/eqn/cacm.ps`. Postscript format.

20. Brian W. Kernighan and Lorinda L. Cherry, *Typesetting Mathematics, User's Guide (Second Edition),* `http://www.kohala.com/start/troff/v7man/eqn/eqn2e.ps`. Postscript format.

21. M. E. Lesk, *Some Applications of Inverted Indices on Unix Systems,* `http://www.kohala.com/start/troff/v7man/refer/refer.ps`. I could not locate the article by Bill Tuthill that describes `refer`. This document is only partly devoted to `refer`. Postscript format.

22. J. L. Bentley, L. W. Jelinski, and B. W. Kernighan, "CHEM—A Program for Typesetting Chemical Structure Diagrams," *Computers and Chemistry,* http://netlib.bell-labs.com/cm/cs/cstr/122.ps.gz, Bell Labs (April 1986.). Another "small language", that is useful for drawing molecular structures of chemicals. In Postscript.

23. J.L. Bentley, B.W. Kernighan, and Tools for Printing Indexes, `http://netlib.bell-labs.com/cm/cs/cstr/128.ps.gz`, Bell Labs (October 1986). A set of tools to work with `troff` and print indexes of documents. Postscript.

24. Michael Haardt, *Style and Diction,* `http://www.gnu.org/software/diction/diction.html`. Diction and style are two standard UNIX commands. Diction identifies wordy and commonly misused phrases. Style analyses surface characteristics of a document, including sentence length and other readability measures.

25. Ralph Corderoy, *Publications that use* `troff`, `http://troff.org/pubs.html`. This webpage lists about 50 publications that use troff. The list includes books writeen by people from Bell Labs (naturally), as well as, among others, W. Richard Stevens.