THE *UNIX*® SYSTEM DOCUMENT PREPARATION TOOLS: A RETROSPECTIVE

Brian W. Kernighan

Brian W. Kernighan *is head of the Computing Structures Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. His research focuses on document preparation software, programming languages, and software tools. Mr. Kernighan joined the company in 1969 and has a Ph.D. in electrical engineering from Princeton University.*

The family of programs in the UNIX system's document-preparation suite has grown over the years to encompass facilities for typesetting mathematics, tables, diagrams, and graphs on a variety of output devices. These programs—the `troff` formatter, the `eqn` language for mathematics, `tbl` for tables, `pic` and `ideal` for diagrams, `grap` for graphs—have been widely used, and some features have been emulated in other systems. In this paper, we will examine these programs, focusing on their most characteristic aspects, and on the lessons we have learned about both document preparation and software development. Although "retrospective" implies only a backward look, there will also be some description of work in progress.

Introduction

The UNIX system provides a family of programs for typesetting technical documents that contain mathematics, tables, figures, and diagrams. (Interested readers should also consult reference 1, an earlier version of this paper.)

The base on which everything else rests is a text formatter called `troff`,[2] developed by J. F. Ossanna around 1973. By itself, `troff` provides few high-level formatting operations beyond line filling and justification. In particular, page layout and document style are defined by packages of formatting requests. These are called "macro packages" because `troff` is a macro processor. (A *macro* is a small program or script that executes several separate commands as a single operation. Panel 1 defines terms and acronyms used in this paper.)

Several programs deal with specialized areas of document preparation. For example, `eqn` is a language for describing mathematical expressions.[3] The `eqn` program operates as a `troff` preprocessor; that is, a document that contains mathematics is passed through `eqn` before it is processed by `troff`. There are other preprocessors as well:

- `tbl` is a language for describing how to typeset tables.[4]
- `refer` fills in and formats bibliographic citations.[5]
- `pic` is a language for describing diagrams.[6]
- `ideal` is another language for diagrams that has capabilities for shading, opaquing, etc.[7]

Some of these programs, in turn, have preprocessors. For example, the `grap` language[8] describes a graphical display of numerical data, and is a `pic` preprocessor. That is, `grap`'s output, which is now in the `pic` language, is directed to the `pic` program, which in turn feeds the `troff` program. Several other programs produce `pic` language as output, including the S statistical analysis system[9] and interactive mouse-based editors for bit-map terminals.

Finally, the output from `troff`, however produced, can be further manipulated or printed on any number of devices, ranging from typesetters through laser printers to color displays and bit-map terminals.

This listing of multiple programs serves not only to introduce the *dramatis personae*, but also suggests several characteristics of what might be called the "UNIX system approach" to document preparation.

The first, and most obvious, characteristic is that there is a group of separate programs that each do one part of the job, rather than a single program that undertakes it all.

Second, we have described most of these tools as languages, rather than merely as programs. The user interface that each program provides is a language, specialized to that program's function.

A third point is the degree to which the development of these programs has been made possible by the software tools and other facilities of the UNIX system. As one instance, the languages for `eqn`, `pic`, `ideal`, and `grap` are all defined by context-free grammars that are translated into parsers by the `yacc` compiler-compiler. All except `eqn` use the `lex` lexical-analyzer generator to create their lexical-analysis routine.

Finally, the basic model of document preparation that these tools support is batch processing. That is, a previously prepared document, with embedded formatting information, is passed through one or more programs for further viewing. Although there are tools for fast previewing on bit-map terminals, the effect is not the same as interactive systems that integrate editing and formatting into "what you see is what you get" (WYSIWYG). For a survey of commercial WYSIWYG systems, see reference 10.

We will return to a discussion of these points after a description of the components.

Panel 1. Terms and Acronyms

ACM	Association for Computing Machinery
`awk`	pattern scanning and programming language
cache	holds recently computed values for reuse
CPU	central processing unit
`eqn`	language for describing how to typeset mathematical expressions
`grap`	language for describing and plotting graphs
`ideal`	language for describing and drawing diagrams
`lex`	a lexical-analyzer generator
macro	a single instruction that stands for a longer sequence of instructions
`pic`	language for describing and drawing diagrams
`refer`	tool to retrieve and format bibliographic citations
`tbl`	language for describing how to typeset tables
trap	mechanism that passes control to a specific macro when certain conditions are met
`troff`	text formatter
WYSIWYG	what you see is what you get
`yacc`	a compiler compiler or parser-generator; converts a set of grammar rules into a program that parses input according to the grammar

`Troff` and Friends

In this section, we will show just enough of each tool to convey the flavor or style of its language. For more details, see references 11 and 12.

`Troff`—The Text Formatter. `Troff` is descended from 1960s formatters that were derived from an original program (`RUNOFF`) created by J. E. Saltzer at the Massa-

Panel 2. This Paper's Manuscript Coded in -ms

The ms macros and text appear at the left; the comments on the right explain what the input means.

```
.TL                                          next line is title
THE \f2UNIX\fP\(rg SYSTEM DOCUMENT PREPARATION TOOLS ...
.AU                                          next line is author's name
Brian W. Kernighan
.AI                                          next line is author's institution
.MH                                          shorthand for "AT&T Bell Laboratories,
                                               Murray Hill, NJ 07974"
.AB                                          beginning of abstract
The family of programs in ...
.AE                                          end of abstract
.NH                                          next line is a heading to be numbered
Introduction
.PP                                          paragraph
The UNIX system provides ...
```


chusetts Institute of Technology. Formatting commands—a period (or dot) in column 1 followed by one or two characters—are interspersed with the text of the document that is being formatted. These commands are procedural, specifying typesetting operations in great detail. For example, a paragraph might be introduced by:

```
.sp 1
.ti 5
```

which causes a vertical line space and a temporary indent of five horizontal spaces. About 80 such commands exist. There are also about 40 in-line commands for changing size and font, inserting special characters or previously defined strings, and invoking various functions. (An *in-line command* is embedded within a text line, while a "dot" command—e.g., `.sp`—must be on a separate line.) For instance, the sequence `\fB` switches from the current font to the bold font; `\w'`*`string`*`'` computes the width of the *string*; and `\h'`*`n`*`'` and `\v'`*`n`*`'` cause horizontal and vertical motion, respectively.

However, `troff` differs from most procedural formatters. Many "normal" formatting operations are not built in, but programming features do exist from which such operations can be created. For example, `troff` does not break its output into pages automatically. There are no footnote commands, no paragraphs, no tables of contents. Instead, `troff` is programmable: it has variables, arithmetic, and conditional tests on many things, including the dimensions of processed text. This means a user can define a macro to encapsulate a sequence of operations or capture processed text. Arguments may be passed to macros. A user can also set a "trap" that causes control to be passed to a specific macro when a given amount of text has been processed. Traps are the mechanism for dividing output into pages, adding titles and page numbers, and printing multiple columns.

In summary, although `troff` is syntactically unappetizing, it is powerful. It enables a user to put any character at any point on the page and, with effort, program `troff` to do anything. But the language is truly

Panel 3. Sample `Tbl` Input

The `tbl` input appears on the left and explanations appear on the right. A tab Ⓣ separates data columns in the table. (See the text for the table this input produces.)

```
.TS
center, doublebox;                    center table, put double box around it
cfB s s                               center heading in bold, stretch across 3 columns
c c c                                 center 3 headings
^ c c                                 center 3 headings, justify 1st vertically
lp8 n n.                              one left justified column, size 8 point; 2 numeric columns
Program Sizes
Name Ⓣ Source Ⓣ Object bytes
Ⓣ Lines Ⓣ (text+data)
_                                     draw line across table here
TROFF Ⓣ 8681 Ⓣ 73136
EQN Ⓣ 1821 Ⓣ 34164
TBL Ⓣ 2581 Ⓣ 39936
REFER Ⓣ 3047 Ⓣ 32768
PIC Ⓣ 3760 Ⓣ 83968
IDEAL Ⓣ 4694 Ⓣ 72704
GRAP Ⓣ 2791 Ⓣ 58368
.TE
```

difficult, and dealing with it directly is to be undertaken only in the most dire circumstances. Indeed, one might argue that most of the remaining components of the document preparation family serve mainly to cover up bare `troff`.

Macro Packages. Because `troff` programming is complicated, there have always been packages of useful macros to provide formatting services like pagination, running titles, and paragraphs. (A *running title* is text that is printed at the top or bottom of each page.) The first such package, `-ms`,[13] is typical. Panel 2 shows how a document such as this one might begin, using `-ms` macros.

As you can see, all formatting details have been suppressed. Instead, the document is described in terms of its logical components (title, author's name and address, headings, paragraphs, footnotes, etc.), not the details of its typography. Those details are not specified until the document is actually formatted. When we run the document through `troff`, the particular macro package used determines the format of titles, how big the paragraph indent is, and so on. Thus, by using different packages, we can print the same document in different styles.

We use this feature extensively to typeset technical papers, first in our internal report style, then for external reports, and finally to provide camera-ready copy for a journal in that publication's format.

In effect, packages such as `-ms` change `troff` from a procedural language into a descriptive one. The intent of macro packages is the same as that of, for example, the Scribe formatter:[14] the separation of format from

content. However, troff constrains the syntax—i.e., all commands consist of one or two-letter names, perhaps followed by arguments—and this is not always esthetic or easy to use. The macro packages remain one of the least satisfactory components of the system.

The monk formatter described elsewhere in this issue[15] is an alternative that translates an input syntax modeled on the Scribe language into troff commands.

Eqn—The Equation Processor. Eqn defines a language for describing mathematical expressions. People with training in mathematics can learn the language in a few minutes; those without such background usually need an hour or so. For example, the input:

```
zeta (z) = sum from n=1 to infinity
1 over n sup z
```

produces:

$$\zeta(z) = \sum_{n=1}^{\infty} \frac{1}{n^z}$$

Eqn runs as a preprocessor: the entire document is passed through eqn before it goes through troff. Eqn copies most of the document untouched, but those parts marked as mathematics (e.g., placed between the commands .EQ and .EN) are translated into troff commands. The expression for the Riemann ζ-function above becomes 44 lines, or 1030 characters, of troff input.

Eqn was the first of the preprocessors, and most of the strengths and weaknesses of the preprocessor approach are to be seen in it. We shall return to this topic later.

Tbl—The Table Processor. The tbl preprocessor is, in concept, the same as eqn, although unrelated in detail. It provides a language specialized to formatting tables. As an example, Panel 3 shows the input that produces this table:

Program Sizes

Name	Source Lines	Object bytes (text+data)
TROFF	8681	73136
EQN	1821	34164
TBL	2581	39936
REFER	3047	32768
PIC	3760	83968
IDEAL	4694	72704
GRAP	2791	58368

Refer—A Reference Handler. Refer mechanizes the tedious but vital task of proving one's scholarship with long lists of references. Given a suitable database, it replaces an imprecise citation such as:

```
.[
furuta scofield shaw
.]
```

with a precise one:

11. R. Furuta, J. Scofield, and A. Shaw, "Document Formatting Systems: Survey, Concepts and Issues," *Computing Surveys*, Vol. 14, No. 3, September 1982, pp. 417-472.

formatted in the style defined by the macro package.

It also provides options for collecting and printing references at the end of the document, sorting by date or author name, identifying references in the text by superscript number or by author name and date, and various capitalization and formats for author names. At the user's convenience, the data for refer (i.e., the text for the reference cited) can be provided in-line (embedded in the text) instead of being extracted from a database.

Pic and Ideal—Picture Processors. Pic and ideal are graphics languages that provide for the inclusion of line drawings and diagrams in documents.

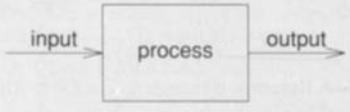
Pic is simpler; it involves placing primitive

Panel 4. Sample Pic Input and Typeset Result

Pic code:

```
.PS
arrow "input" above
box "process"
arrow "output" above
.PE
```

Typeset output:



objects (e.g., boxes, circles, lines, splines) at absolute positions or relative to previously placed objects. As an example, Panel 4 shows a diagram and the `pic` input that produced it. Each object may be followed by a list of attributes that control the object's size and position and provide the associated text strings. A macro facility allows a user to encapsulate and invoke common operations, perhaps with different parameters each time. There are also iteration and conditional constructs.

A recent version of `pic` created by M. L. Siemon produces PostScript® output.[16] (PostScript is a registered trademark of Adobe Systems, Inc.) This permits access to many of PostScript's graphics effects, including color and shading and arbitrary scaling and rotation.

`Pic` is often used as the *target language* (i.e., the output language) for other programs, serving as a relatively low-level graphics language that can be typeset. The main example is the graph-drawing language `grap`, described in the next section. Others include interactive tools like S, and languages for circuit diagrams, optical ray tracing, and chemical structure diagrams.

In `ideal`, points are defined in terms of simultaneous equations in complex coordinates. To draw the picture, `ideal` solves the constraint equations and connects the specified points with lines, circles, and splines. Panel 5 shows typical `ideal` input and the line art it produces.

`Ideal` also serves as the target language for other processors, notably a program[17] by H. W. Trickey for drawing graphs (i.e., diagrams that show nodes and arcs).

`Grap` and Related Programs. `Grap` provides a language for plotting data; it produces `pic` commands rather than feeding directly into `troff`. By default, `grap` plots input numbers as a scatter plot and automatically computes tick marks (interval marks on the graph's axes). Other commands provide control over labels, scaling, logarithmic coordinates, and the like.

Panel 6 shows the `grap` input that produced the graph in Figure 1. `Grap` also has the same macro processing and control-flow features as `pic` does.

Several very small, specialized language preprocessors also produce `grap` language as output. Most of these create statistical displays, such as dot charts (a form of histogram or bar chart), scatter-plot matrices, and box plots.[18]

`Troff` Postprocessors. The original version of `troff` produced output commands for a specific typesetter. The current version, called "device-independent `troff`," produces output in a form that can be translated readily into commands for any reasonable device. That means one driver program is required for each type of typesetter. But with the PostScript language emerging as a *de facto* standard, a driver that produces PostScript is often enough. This also is convenient for merging `troff` output with output from other sources (for example, digitized photographs).

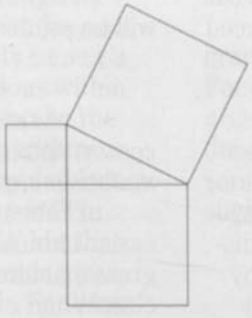
Page makeup—vertical justification, figure placement, avoidance of "widows"—is one area not satisfactorily handled by `troff` macro packages. (In publishing, a *widow* is a one- or two-word line at the end of a paragraph or a short line that spills to the top of the next

Panel 5. Sample `Ideal` Input and Typeset Result

`Ideal` code (`rect` is defined in a library):

```
.IS
...libfile rect
...width 1.25
main {
   put first: rect {          # print smallest rectangle
      ht = wd = 1;            # with height and width 1
      sw = 0;                 # and southwest corner at origin
   };
   put next: rect {
      nw = first.se;          # nw corner at se corner of first
      ht = wd = 2 * first.ht;
   };
   put last: rect {           # on the hypotenuse
      sw = first.ne;
      se = next.ne;
      ht = wd;
   };
}
.IE
```

Typeset result:



Panel 6. Grap Input for Figure 1

```
.G1
frame top invis right invis
label left "Time" "(in seconds)"
label bot "Olympic 400 Meter Run: Winning Times"
ticks left in at 45,50
ticks bot in at 1900,1950
draw solid
1896 54.2
1900 49.4
1904 49.2
...
1980 44.60
1984 44.27
arrow from 1924,51 to 1924,49
" ''Chariots of Fire''" ljust at 1924,51
.G2
```


page. Some people call the latter an *orphan*.) A recent program[19] deals with page makeup by postprocessing `troff` output. Extra information is added to the output to tell how various elements of the document should appear on the page. The postprocessor uses this information to make all pages the same height, prevent the creation of widow lines, place footnotes, and "float" figures. (That is, if a figure is too tall for the space left on the current page, text continues printing and the figure is placed on the next page.)

Preprocessors

The basic structure of the preprocessors is the same: input is copied unchanged to the output, except for the translation of marked portions into `troff`. In `eqn`, for example, there are two kinds of markers: one for displayed equations (i.e., separated from the text, often by blank lines), and one for in-line equations (i.e., embedded in the text). Two macros, `.EQ` and `.EN`, identify the start and end of a displayed equation. Lines between these macros are translated, while everything else is passed through. The `.EQ` and `.EN` are also copied through for further processing; most macro packages define them to center the equation and perhaps print an equation number. `Eqn`'s in-line mode uses two user-settable characters to delimit an in-line expression. So, for example, if the delimiters are both `@`, then

```
Let @alpha sub i@ and @beta sub j@ be ...
```

will be printed as

Let α_i and β_j be ...

As mentioned above, `eqn` was the first preprocessor, and most of the lessons learned from it apply, *mutatis mutandis*, to the other preprocessors as well.

The most important lesson is that it is much easier to build a preprocessor than to modify a big program that already exists and that belongs to someone else. When `eqn` was written, `troff` consisted of about 10,000 lines of intricate assembly language. But it would have been unthinkable to add the facilities of `eqn` to `troff`, even if the resulting program would have fit in

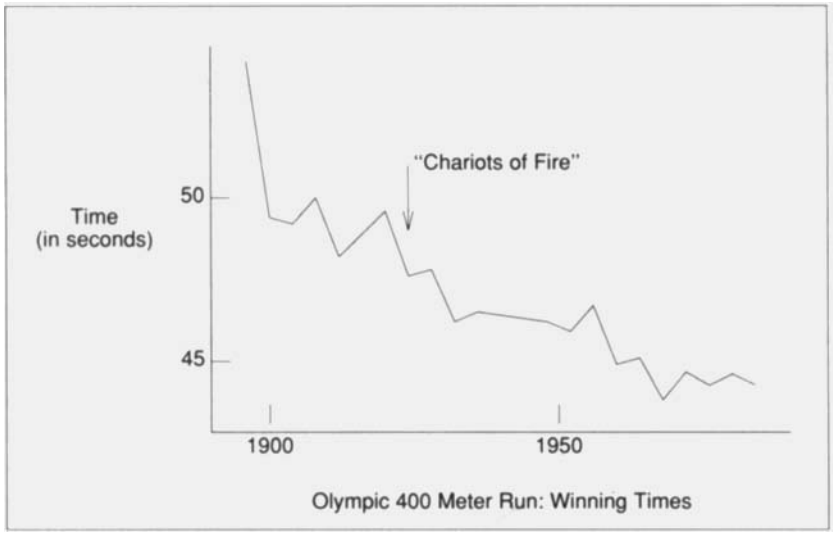


Figure 1. A typeset graph. See Panel 6 for the `grap` input.

the limited address space of the PDP™-11 minicomputer on which it ran at the time. (PDP is a trademark of Digital Equipment Corporation.) A separate program obviates these problems, and makes it possible to create a language that is suited to its particular task, unconstrained by the syntax of other languages. Viewed in this light, `eqn` becomes a small compiler, translating into a strange and specialized assembly language.

It is a tribute to Ossanna's original design for `troff` that no changes were needed in `troff` for `eqn` to do its job. `Eqn` generates code that exploits `troff`'s ability to include horizontal and vertical motions within text, store text and numbers in variables, compute the widths of arbitrarily complicated text, perform arithmetic, and take different actions depending on computed values.

This is an essential point. It is all very well to suggest preprocessors as a way to build tools, but there is an implicit requirement that the postprocessing program have enough functionality to do the job. For example, it would be difficult to write an `eqn` for a formatter that does not include the arbitrary positioning of text and storing of intermediate steps on which `eqn` relies.

Similarly, `troff` provides an in-line command that causes extra line space to be added before or after the output line that contains the `troff` command. `Eqn` uses this to ensure that a tall expression does not collide with the lines above and below it, a task that would be very difficult without the `troff` primitive.

Problems of Preprocessors. The benefits of breaking a big multifaceted job into a set of smaller, decoupled tasks are compelling, but there are negative aspects as well.

It is sometimes hard to remember all the programs and options involved in formatting a document. For complicated documents, a small shell program called `doctype` deduces the proper command line by scanning the document for formatting commands. For this manuscript, which is stored in a file called `retro`, `doctype` suggests (correctly)

```
refer -e retro | ideal | grap |\
  pic | tbl | eqn | troff -ms
```

as the command line. (The vertical bars "|" cause the output from the previous program to be passed or

"piped" to the next one. Such a command line is called a *pipeline.* Note that the backslash "\" at the end of the first line is used to continue a long line onto a second line.)

If a document is to be formatted repeatedly (while being written, for example), the proper sequence of commands might well be put into a shell file. Documents with complicated processing can also be controlled by the standard tool `make`, which is normally used for controlling the compilation of large programs.

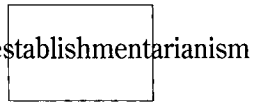
Error Messages. Preprocessors may produce error messages that bear no apparent relation to the user's input. As an example, at one point, the pipeline above resulted in the message:

```
Input: line 756: no specification
tbl quits
```

The "`756`" refers to a line in an input stream that is the result of four previous programs, not to a line in the original input file. Now, each preprocessor interprets and produces `troff` commands that maintain the user's notion of the line number and filename, so that error messages can be better related to the original input.

Communications problems. A more serious problem arises from the one-way nature of the communication between programs in a pipeline. As mentioned above, `eqn` generates conditional code because it cannot determine for itself the answers to questions like: *Is this string longer than that one?* Also to compute some motions properly, `eqn` needs to know the point size in which an expression is being set but, as a preprocessor, there is no way that it can ask `troff`. So, `eqn` makes an educated guess about the size and generates code that contains only relative sizes and motions; although imperfect, this works well enough over the normal range of sizes.

Neither `ideal` nor `pic` attempt to adjust the dimensions of boxes, etc., to fit the text they surround, in part because the `troff` code generation is too hard. It is left to the user to revise a picture like this:



by changing the default size for the box, or by moving, shrinking, or rewriting the text.

Target language. We have already mentioned that the target language for a preprocessor must provide the right primitives for the job at hand. But beyond that, there are language defects that show up only when a program (such as `troff`) is subjected to input prepared by a program instead of a person. For example, the `troff` language is irregular, often because it reflects idiosyncrasies of a typesetter long since abandoned. An in-line size change can be specified as `\s`*n*, where *n* is a single digit from the set 0, 4, 5, 6, 7, 8, 9, or as `\s`*nn*, where *nn* is two digits in the range 10 to 39. The input `\s39` causes a shift to size 39, while the input `\s40` causes the point size to become 4 and a zero to be printed! It is unfortunate that we have to wire such rules into preprocessors.

`Troff` also inherits syntax problems from its original implementation on the 16-bit PDP-11 computer. Two-character names for commands and variables are the most obvious of a large number of such defects, many of which are exacerbated by machine-generated input. Recent versions of `troff` provide alternative syntax that permits more regular specification of input, including longer names for characters.

The target language may be missing some feature, the absence of which complicates the preprocessor's task. At one time, for example, the `grap` preprocessor was forced to print numbers in F format (i.e., `1000000`) instead of exponential notation (`1E6`) because, through lack of imagination by the implementer, `pic` did not recognize the latter. Fortunately, this was easy to fix.

Translating the preprocessor's language into `troff` code is complicated because `troff` does not maintain a stack of recent point sizes or fonts. This

forces preprocessors to manipulate sizes and fonts explicitly instead of simply pushing down a previous state.

Efficiency. Another characteristic of machine-generated input is simply that there is often a lot of it. For example, the graph shown in Figure 2 of the field lines of a differential equation is about 460 bytes of `grap` input, which becomes 7870 bytes of `pic` input, 6710 bytes of `eqn` input, 8600 bytes of `troff` input, and 5900 bytes of `troff` output. `Pic` pictures have been known to exceed 50,000 elements, which requires, perhaps, 5 megabytes of internal storage and produces a lot of output.

The processing time required for running multiple programs, each expanding the size of its input, is a concern for complicated documents. For example, the manuscript for this paper takes about 18.7 seconds to process (producing 138,100 output bytes) on a Digital Equipment Corporation VAX™-8550 computer:

Program	Bytes of input	CPU time (seconds)
`doctype`	44500	0.2
`refer`	44500	0.5
`ideal`	45900	0.4
`grap`	46750	1.3
`pic`	56300	1.7
`tbl`	58800	0.6
`eqn`	67000	0.9
`troff -ms`	71850	12.1

Remember that each preprocessor copies the whole input, even though it processes only a small part. For contrast, it requires 9.2 seconds simply to run the document through `troff` with `-ms` but without the preprocessors.

In general, there are stresses in two areas: fixed-size tables are never big enough, and simple algorithms (e.g., linear searches) are often not fast enough. The first change that was made to `troff` after `eqn` began to be

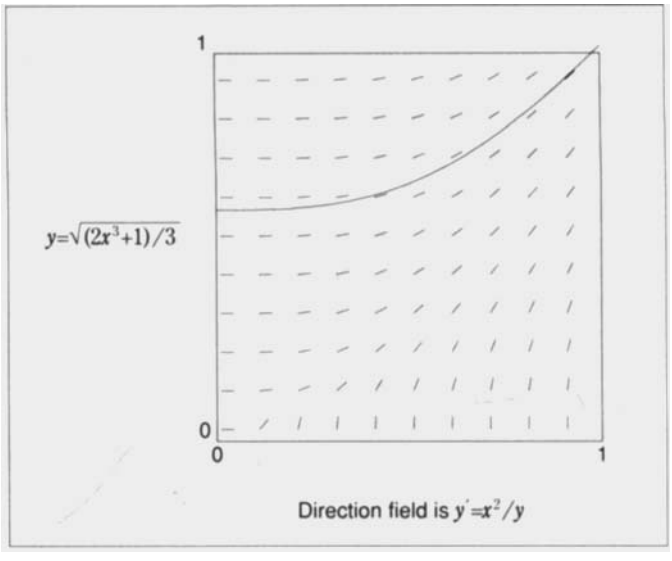


Figure 2. Field lines of a differential equation. This graph was produced using `for` loops. See Panel 8 for the `grap` input.

heavily used was a complete revision of the string-handling code; the original mechanism was an order of magnitude too slow. More recent changes to `troff`—hash tables for macro names and number registers, a cache for width computations—have also been motivated by efficiency concerns. (*Hashing* is a technique for locating information quickly, by using the name as an address. For example, the name of a macro is used to compute where the macro's definition is stored. A *cache* holds recently computed values so they can be used again instead of being recomputed.)

Similarly, the use of `grap` has forced a change in `pic`. All arrays now grow dynamically until memory is exhausted.

Combining preprocessors. For several problems, the existing programs do not provide good solutions. One problem is how to mix the various components in arbitrary ways. In the current implementations, "nesting"

(e.g., placing a table within a table) is not permitted but pictures, graphs and tables may all contain mathematics. Pictures inside tables, and vice versa, do not work, and there are some clumsy mechanisms to ensure cooperation between `eqn` and other preprocessors.

For example, if we want to include a tall equation in a picture, then the `eqn` input must include a command to suppress automatic addition of space before and after the equation. Otherwise, the equation positioning would interfere with the line spacing that `pic` or `ideal` is doing. `Tbl` must duplicate `eqn`'s command for setting in-line delimiters so users can prevent `tbl` from splitting equations.

Languages

Most of the tools in the UNIX system's document preparation family implement languages. There are advantages to thinking in terms of "language" rather than "program." Languages are not *ad hoc* collections of commands, but a set of entities and grammar rules that state precisely how the entities may be combined into larger structures. With some forethought, artificial languages can have regular structure, and avoid the silly restrictions and special cases that often characterize programs not designed from an explicit linguistic view. Building programs in terms of languages also encourages the use of tools to mechanize various parts of the job, particularly parser-generators for defining the grammar and lexical-analyzer generators for defining the lexical tokens. (A *parser* is a routine that analyzes the grammatical structure of a program—rather like diagramming a sentence. A *lexical analyzer* is a routine that groups the characters of a program into names, operators, etc. The generators create these routines from compact descriptions.)

In the document preparation tools we are discussing, the languages are separate from each other and from the underlying formatter. This means that each one may be tailored to its domain of application, with no need to fit the linguistic model of any other. By contrast, the sublanguages within other formatting systems are constrained by the linguistic rules of their host language in a way that can make them less natural.

Two examples illustrate the point. In the Scribe formatter,[14] mathematics must be expressed in a functional notation because that is the only notation the language provides. For example, the expression:

$$\sum_{i=1}^{n} f(x_i)\,\Delta x$$

is written in Scribe as:

```
@Sum(From i=1, To n) f#(x@down[i]) Delta x
```

while in `eqn`, it is:

```
sum from i=1 to n f(x sub i ) DELTA x
```

Similarly, although "math mode" in the TEX™ formatter[20] has some separate semantic rules, the syntax is consistent with the rest of the TEX language. (TEX is a trademark of the American Mathematical Society.) For example, *all* reserved words begin with \ (i.e., with a backslash). The result is that the mathematical parts of a TEX document seem more cluttered than their `eqn` counterparts, although it can be argued that the TEX rules are more systematic.

Consider the sample matrix and the code excerpts in Panel 7. In the TEX language, column alignment is specified first, then the elements are entered by rows. In the `eqn` language, the alignment is defaulted and the elements are entered by columns. (`Eqn` automatically aligns the columns and centers each element; however, the user can specify different handling.)

It is time to think in terms of a language when a task becomes repetitive and well enough understood that the primitive operations are clear. Our experience with the `grap` language for typesetting graphs may be instructive. J. L. Bentley had used `pic` to produce graphs similar

Panel 7. Comparison of TEX and Eqn Syntax

Sample matrix:

$$A=\begin{bmatrix} x-\lambda & 1 & 0 \\ 0 & x-\lambda & 1 \\ 0 & 0 & x-\lambda \end{bmatrix}$$

TEX code for matrix:

```
A=\left(\vcenter{
   \halign{$\ctr{#}$\quad
      Ⓣ $\ctr{#}$\quad
      Ⓣ $\ctr{#}$\cr
   x-\lambda Ⓣ 1 Ⓣ 0\cr
   0 Ⓣ x-\lambda Ⓣ 1\cr
   0 Ⓣ 0 Ⓣ x-\lambda\cr}}\right)
```

Equivalent `eqn` code:

```
A = left (
    matrix {
       col {x - lambda above 0 above 0}
       col {1 above x - lambda above 0}
       col {0 above 1 above x - lambda}
    }
    right )
```

to the one shown in Figure 1. He found it feasible—`pic` provided the necessary facilities—but tedious and repetitive. I had previously modified the UNIX system program `graph` to produce `pic`, in an unsatisfactory attempt to produce typeset graphs. After a conversation revealed this mutual interest, Bentley and I designed a language that we felt captured the basic notions of graphs: a frame with optional labels and tick marks, data plotted as lines or points, and automatic scaling of data to fit the frame. The original design took about two afternoons' worth of discussion, and I implemented a first version of the program, using `yacc`, `lex`, and some code from `pic`, in a single (long) evening.

Since then, the language has grown substantially as we have attacked more complicated graphs, and as our user population has made suggestions. In its current form, it provides:

- Automatic scaling of data in multiple coordinate systems (linear or logarithmic on either axis)
- Automatic or manual generation of labeled ticks and grid lines
- Labeling of axes
- Plotting of data as points or as lines in several styles.

It has a macro processor, file inclusion with automatic processing through a macro, a `for` loop and `if-then-else`, variables, arithmetic expressions, and a complement of built-in functions. (A `for` loop consists of a condition—e.g., does a variable have a specific value—that is tested on each iteration and operations that are executed if the condition is met. An initial state for the variable and incrementing for each iteration may also be specified. An `if-then-else` specifies operations to be executed if a condition is met, and another set of operations if it is not.) Yet `grap` remains easy to use for the simplest cases. Given a column of numbers, it will provide abscissae and plot the data with tick marks at sensible locations.

Language features. We have found that several language features are useful almost universally. For example, macros provide a way to abbreviate a construct and use it over and over; macros with arguments are even better. `Pic` and `grap` provide macros with arguments, and this facility has been added to `eqn`, which previously permitted only parameterless macros. Although macro processors are usually slower than compiled code, they are flexible and their implementation is easy. `Ideal` has a very general procedure mechanism, which serves many of the same purposes.

The ability to take input from a file is convenient in general; it is mandatory in `grap`, which typically pro-

cesses data created by other programs. The `grap` notion of copying a file through a macro is especially convenient:

```
.G1
copy "file" through { macro body }
.G2
```

copies the file. Each line in the file is treated as one invocation of the unnamed macro, with each field on the line treated as an argument.

The ability to evaluate arithmetic expressions and store the results in variables is necessary in a language that deals with numbers. Languages for graphics should have at least built-in functions for logarithms, exponentials, and trigonometry. We have also found it valuable to have loops and conditionals.

One facility that is needed in all the preprocessors is an escape to the underlying languages, because there is no reasonable way for the preprocessor to provide everything. The simplest example is the ability to pass something to `troff` without interpretation. This capability is heavily used in `eqn`, where any quoted string is copied through intact. The quoted strings provide access to `troff`'s set of special characters and to its in-line commands for horizontal and vertical motions, which can be used to adjust spaces or define built-up characters. `Ideal`, `pic`, and `grap` also provide this quoting mechanism and, as well, copy through any line that begins with a period, under the assumption that this is a `troff` command.

In all these cases, the onus is on the user to be sure that the passed-through commands do not interfere with the code being generated by the preprocessor itself. For example, if a `pic` user changes the `troff` line spacing in the middle of a picture, chaos ensues.

We also permit escapes to the shell from the various tools. In this way, a program can be run to produce some data that is then included by file copy, while keeping the document a self-contained entity rather than a collection of pieces.

It is unclear just how far to go toward making each language "complete." Does `eqn` need a `for` loop, for example? That seems unlikely, but things are not always so clear. The original version of `grap` provided no looping facility. But since the `for` statement was added, many uses for it have been found, including making multiple frames of a "movie" to illustrate the evolution of a function, and making the picture in Figure 2 of the field lines of a differential equation. Panel 8 shows the input for the graph in Figure 2. Notice that other trappings of programming languages find their way in as well, right down to comment conventions.

Tools

The UNIX system programming environment is our basic tool. Without that, it is doubtful that we would have come to develop these programs, particularly in the way that we have. The notion of pipes, and of separate programs cooperating to get something done, is central to the system. Furthermore, programming languages like C and `awk`, and general-purpose tools for file comparison, pattern searching, and editing all help to make the task easier.

`Eqn` was among the first applications of `yacc`, and its first use for a nonconventional programming language. The `yacc` compiler-compiler (or, more accurately, parser-generator) has made it possible for us to define new languages and adapt them quickly as experience dictates, in a way that would be hard with conventional handwritten parsers. Furthermore, the resulting languages really are languages, with well-understood grammatical properties.

The `lex` lexical-analyzer generator serves much the same purpose for lexical analysis as `yacc` does for syntax, and provides many of the same benefits, particularly the ease of changing the lexical rules. For example, if we modify the `pic` language to permit input numbers in E format as well as F, all we need do is change a single

Panel 8. `Grap` Code to Produce the Graph in Figure 2

This example illustrates `grap`'s programming language features, e.g., `for` loops and built-in functions.

```
.G1
frame ht 2 wid 2
coord x 0,1 y 0,1
label bot "Direction field is $y sup prime = x sup 2 / y$"
label left "$y = sqrt {(2x sup 3 +1)/3}$" right .6
ticks left in 0 at 0,1
ticks bot in 0 at 0,1
len = .035
for tx from .01 to .91 by .1 do {
  for ty from .01 to .91 by .1 do {
    deriv = tx*tx/ty
    scale = len/sqrt(1+deriv*deriv)    # keep lines same length
    line from tx,ty to tx+scale,ty+scale*deriv
  }
}
draw solid
for tx = 0 to 1 by .05 do {
  next at tx, sqrt((2*tx*tx*tx+1)/3)
}
.G2
```


regular expression from

```
[0-9]+(".")?[0-9]*|"."[0-9]+
```

to

```
([0-9]+(".")?[0-9]*|"."[0-9]+)\
([eE][+-]?[0-9]+)?
```

In our laboratory, we use bit-map terminals, which provide a high-resolution screen and a multiple-window environment where each window is associated with a separate process. For document preparation, one normally has a window for editing, another for running programs, and a third for viewing the output. The effect is not quite "what you see is what you get," but it is close enough for most purposes, while still retaining the advantages of the batch-oriented formatting tools.

Conclusions

It would be rash to claim that the tools described here are the "best" way to do document preparation, but they have worked well for us. In particular, the strategy of building separate programs has been especially productive for research, because it means that anyone with a good idea can try it out, without a huge effort, and without worrying about how to squeeze new ideas into an old program.

We have always developed our tools for our own

use, not for our perception of what someone else might need. Accordingly, as new programs become available, they are stressed and refined under criticism by colleagues. Of course, the needs of people writing document preparation languages have had a positive effect on our language development tools, from the earliest experience with `yacc`.

Document preparation is and will remain an excellent area for research. It is a microcosm of computer science, from theory through algorithms to compilers and parallel processing to user interfaces. But the problems involved are of manageable size, and better solutions find immediate application.

Acknowledgments

I am grateful to Jon Bentley, Tom Cargill, Doug McIlroy, Peter Nelson, and Chris Van Wyk for many helpful criticisms of this paper.

References

1. B. W. Kernighan, "The UNIX™ Document Preparation Tools—A Retrospective," *Proceedings of Protext I*, Text Processing System International Conference, Dublin, Ireland, October 1984, pp. 12-25.
2. J. F. Ossanna, "NROFF/TROFF User's Manual," *UNIX™ Time-Sharing System: UNIX Programmer's Manual*, Seventh Edition, Volume 2, Bell Laboratories, Murray Hill, New Jersey, Holt, Rinehart and Winston, New York, January 1979, Section 12, pp. 196-229.
3. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Communications of the ACM*, Vol. 18, No. 3, 1975, pp. 151-157.
4. M. E. Lesk, "TBL—A Program to Format Tables," *UNIX™ Time-Sharing System: UNIX Programmer's Manual*, Seventh Edition, Volume 2, Bell Laboratories, Murray Hill, New Jersey, Holt, Rinehart and Winston, New York, January 1979, Section 10, pp. 157-174.
5. M. E. Lesk, "Some Applications of Inverted Indexes on the UNIX System," *UNIX™ Time-Sharing System: UNIX Programmer's Manual*, Seventh Edition, Volume 2, Bell Laboratories, Murray Hill, New Jersey, Holt, Rinehart and Winston, New York, January 1979, Section 11, pp. 175-195.
6. B. W. Kernighan, "PIC—A Language for Typesetting Graphics," *Software Practice & Experience*, Vol. 12, January 1982, pp. 1-20.
7. C. J. Van Wyk, "A High-Level Language for Specifying Pictures," *ACM Transactions on Graphics*, Vol. 1, No. 2, 1982, pp. 163-182.
8. J. L. Bentley and B. W. Kernighan, "GRAP—A Language for Typesetting Graphs," *Communications of the ACM*, Vol. 29, No. 8, August 1986, pp. 782-792.
9. R. A. Becker, J. M. Chambers, and A. R. Wilks, *The New S Language*, Wadsworth & Brooks/Cole, Pacific Grove, California, 1988.
10. M. E. Walter, Jr., "Technical Documentation Systems," Seybold Publications, Media, Pennsylvania, 1988.
11. R. Furuta, J. Scofield, and A. Shaw, "Document Formatting Systems: Survey, Concepts and Issues," *Computing Surveys*, Vol. 14, No. 3, September 1982, pp. 417-472.
12. B. W. Kernighan and M. E. Lesk, "UNIX Document Preparation," *Document Preparation Systems*, J. Nievergelt, G. Coray, J.-D. Nicoud, and A. C. Shaw (eds.), North-Holland Publishing Co., New York, 1982, pp. 1-20.
13. M. E. Lesk, "Typing documents on the UNIX system: using the –ms macros with troff and nroff," *UNIX™ Time-Sharing System: UNIX Programmer's Manual*, Seventh Edition, Volume 2, Bell Laboratories, Murray Hill, New Jersey, Holt, Rinehart and Winston, New York, January 1979, Section 8, pp. 125-145.
14. B. K. Reid, "Scribe: A Document Specification Language and Its Compiler," Ph.D. thesis, Report No. CMU-C5-81-100, Computer Science Department, Carnegie-Mellon University, October 1980.
15. S. Murrel and T. J. Kowalski, "Monk: A High-Level Text Compiler," *AT&T Technical Journal*, Vol. 68, No. 4, July/August 1989, pp. 45-60.
16. Adobe Systems Inc., *PostScript Language Reference Manual*, Addison-Wesley, Reading, Massachusetts, 1985.
17. H. W. Trickey, "Drag—A Graph Drawing System," *International Conference on Electronic Publishing, Document Manipulation and Typography*, Nice, France, April 1988, J. C. Van Vliet (ed.), Cambridge University Press, pp. 171-182.
18. W. S. Cleveland, *The Elements of Graphing Data*, Wadsworth & Brooks/Cole, Pacific Grove, California, 1985.
19. B. W. Kernighan and C. J. Van Wyk, "Page Makeup by Postprocessing Text Formatter Output," *Computing Systems*, Vol. 2, No. 2, 1989, pp. 103-132.
20. D. E. Knuth, *TEX and METAFONT, New Directions in Typesetting*, Digital Press, Bedford, Massachusetts, 1979.

(Manuscript received May 1, 1989)