MONK: A HIGH-LEVEL TEXT COMPILER

Sharon L. Murrel and Thaddeus J. Kowalski

Sharon L. Murrel *and* ***Thaddeus J. Kowalski*** *are members of technical staff in the Computer-Aided Information Systems Research Department at AT&T Bell Laboratories in Murray Hill, New Jersey. Ms. Murrel's research interests include creating advanced systems for electronic textual communication, particularly focusing on publishing and computer conferencing. She is currently building an interactive system for designing tabular information using "layout-by-example" as well as developing new languages and tools for automated production of large publications, like this journal. She joined the company in 1977 and has a B.S. in computer science from New York University, and an M.A. and Ph.D. in experimental psychology from the State University of New York at Stony Brook. Mr. Kowalski is currently* (continued on page 60)

The `monk` document preparation system is a sophisticated set of computer programs that helps writers produce high-quality typeset output. It compiles high-level commands that appear in text into commands appropriate for a typesetting language. While `monk` is designed to simplify the preparation of documents, the main focus of our research has been the development of a new database language that facilitates the maintenance of precise, reusable typographic primitives. This article starts by providing a writer's view of a `monk` document and then presents the layout designer's view of the database hierarchy and composition. This separation of writer and designer views of document preparation is the keystone in the creation of attractive yet uniform documents.

Introduction

The `monk` document preparation system is a sophisticated set of computer programs that helps writers produce high-quality typeset output. It compiles English-like commands that appear in the input text into the precise typographical information that layout designers use for headers, footers, sections, figures, footnotes, references, cover sheets, tables of contents, indices, and more. `Monk` does this by consulting a database of document styles that contains the rules for compiling the commands into typeset output.

The database idea is the key to the power of `monk`. For document types already stored in the database, writers need not concern themselves with the details of typography. However, when a new document style is required, designers can quickly create one to fill the need.

The database concept for typography is not new—to the best of our knowledge it was first used at Carnegie-Mellon University in a system called Scribe.[1] We improved on Reid's idea by separating general layout from device-specific information; facilitating typographic design using a high-level language based on regular-expression string matching; and mapping document descriptions into an underlying

typesetting language.

Another document preparation system that was influenced by the Scribe approach is LATEX, which is built on a `troff`-like program, the TEX™ typesetting system.[2] (TEX is a trademark of the American Mathematical Society.) Lamport explained:[3]

"In turning TEX into LATEX, I have tried to convert a highly-tuned racing car into a comfortable family sedan. The family sedan isn't meant to go as fast as a racing car or be as exciting to drive, but it's comfortable and gets you to the grocery store with no fuss. However, the LATEX sedan has all the power of TEX hidden under its hood, and the more adventurous driver can do everything with it that he can with TEX."

`Monk` grew out of our efforts to add yet another document type to the UNIX® system's memorandum macros, `mm`.[4] From the beginning we conceived of `monk` as a set of software tools and filters that would work well with other UNIX system tools. (Panel 1 defines terms and tools.) More specifically, `monk` is a block-structured text compiler that produces `troff`[5–7] as its assembly language. Although this article will present `monk` as a preprocessor for `troff`, it is designed to support many different typesetting languages.

The relationship of `monk` and the macro packages to `troff` is similar to that of LATEX to the TEX typesetting system. Kernighan advised:[6]

"The single most important rule of using `troff` is not *to use it directly, but through some intermediary. In many ways, `troff` resembles an assembly language—a remarkably powerful and flexible one—but nonetheless such that many operations must be specified at a level of detail and in a form that is too hard for most people to use effectively."*

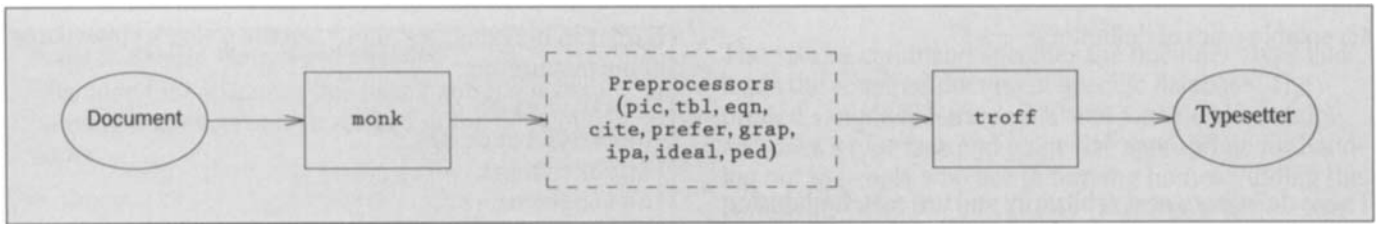
In 1977, the way to reduce contact with an assembly language was to write macro packages. This was as true for programming as it was for typesetting languages. Today, that reduction is done using high-level hierarchical languages, such as C for programming and `monk` for document preparation. `Monk` moves further away from assembly language by replacing the two character mnemonics used by the macro packages with English-like commands and replacing in-line `troff` with a library of primitives.

Since `monk`'s creation, document style sheets have been developed for a wide variety of publications:

- Articles for the *AT&T Technical Journal* and the

Panel 1. Tools and Terms

`cite`	builds citations and cross references
`demonk`	deletes the `monk` commands in a file, leaving only the raw text
`eqn`	language for describing mathematical expressions
`grap`	language for describing and plotting graphs
`ideal`	language for describing and drawing diagrams
`index`	organizes and formats indexes
`ipa`	language for describing and producing the International Phonetic Alphabet
macro	a sequence of commands that are executed as a single operation
`mm/ms`	standard macro packages
`monk`	style-sheet based formatting program
`monkmerge`	replace a `monk` reference to an external file with the contents of that file
`monksample`	provides a sample document with the basic `monk` commands for that document type
`ped`	language for describing and drawing pictures
`pic`	language for describing and drawing pictures
`prefer`	retrieves and formats bibliographic citations
preprocessor	tool that processes a special-purpose language and generates `troff` input
`soelim`	replace the `troff` reference to an external file with the contents of that file
`spell`	displays possible spelling errors in a file
`tbl`	language for describing and drawing tables
`troff`	typesetting language and text formatter



Association for Computing Machinery

- Books for the Computer Science Press and Kluwer Academic Publishers
- AT&T Bell Laboratories technical memorandums and released papers
- Telephone books, letters, and research reports
- Song sheets for singing at nursing homes.

The breadth of these publication styles has encouraged us to develop a library of reusable typographic primitives that enables us to create most new document styles in a day or two. A single primitive can center a block of text or place a formatted reference list.

Figure 1 shows `monk`'s role in the documentation process. `Monk` replaces the `mm` macro package while retaining interfaces to task-specific preprocessors:[8–16] `tbl`, `eqn`, `pic`, `cite`, `grap`, `ipa`, `ideal`, `ped`, and `prefer`. Relative to the macro packages, `mm` and its sibling `ms`,[17] which have become the standard tools for document preparation at AT&T Bell Laboratories,[18] `monk` improves the interface with the writer and facilitates customization and maintenance. It also speeds document processing. Thus, `monk` compiles the user-level commands into the machine language that is directly executed by `troff`. This is faster than the macro commands, which must be run interpretatively by `troff`.

We will now provide a writer's view of a `monk` document, and then present the layout designer's view of the database hierarchy and its composition. This separation of the writer's and designer's views of document preparation is the keystone in the creation of attractive yet uniform documents. We also believe it aids writers by allowing them to concentrate on the structure of the text rather than its particular typography.

Figure 1. The documentation sequence using `monk`.

The Writer's Point of View

If Jeremy Bernstein was using `monk` to write *Three Degrees Above Zero*,[19] the writer's view of one of his pages might be:

```
|small(ON THURSDAY), 1 July 1948, several
notes of potential interest to radio
listeners and television watchers
appeared in ''The News of Radio'' column
of the |italics<New York Times>:

|begin(quotation)
A device called a transistor, which
has several applications in radio
where a vacuum tube ordinarily is
employed, was demonstrated |bold{. . .}
|paragraph
Along with these electronic
demonstrations, Bell's public relations
department had provided an eight-foot
model transistor on wheels, intended to
explain just how simple the whole thing
really was.
|end(quotation)
```

This example shows several `monk` commands. All `monk` commands begin with the character '|' (pronounced *pipe*). Walking through the example, the first command, `|small`, which can be abbreviated as `|s`, tells `monk` to reduce the size of the text. Here, the parentheses delimit the words to be shrunk; however, there are seven

acceptable pairs of delimiters:

```
(...), [...], <...>, {...},
"...", '...', `...'
```

These delimiters nest arbitrarily and are matched during parsing. The next command, |italics or |i, changes the type face to italic.

When a command operates on a long text segment, the commands |begin and |end serve as delimiters. The command quotation found in the standard database, indents text from both margins by a tasteful amount of white space. (The definition tailored for the *AT&T Technical Journal*, however, changes the type face to italics but does not change the margins.) The |bold or |b command works just like italics except it changes the type face to bold. Finally, the |paragraph or |p command starts a paragraph. We would like to draw your attention to three features:

- Monk provides abbreviations as well as English-like command names for the more frequently used commands. The standard command names are controlled by the layout designer and the database manager.
- Monk commands need not appear at the beginning of a line.
- Monk allows the writer to separate input text with white space, increasing its readability without affecting the output text.

Bernstein's view would be typeset as:

ON THURSDAY, 1 July 1948, several notes of potential interest to radio listeners and television watchers appeared in "The News of Radio" column of the *New York Times*:

A device called a transistor, which has several applications in radio where a vacuum tube ordinarily is employed, was demonstrated . . .

Along with these electronic demonstrations, Bell's public relations department had provided an eight-foot model transistor on wheels, intended to explain just how simple the whole thing really was.

Earlier in his book, we might see the writer's view of one of his pictures as:

```
|begin(here)
|begin(picture)
|insert(best.net)
|end(picture)
|picture_caption(Best Imaginary Network)
|save(best)
|end(here)
The length of the imaginary network
in Figure |remember(best) is
|e<( sqrt 3 ~+~ 1 ) a>, or |e<2.7 a>.
```

Here, we see the use of nested commands to produce a figure that contains a picture with its caption. This is done with the long forms of the monk commands here, picture, and |picture_caption. The monk command |insert provides a facility to insert other files in the input text; the file best.net contains the commands that draw the diagram. The figure number is saved as the symbol best using the monk command |save(best) and recalled using |remember(best). When the document is printed, |remember(best) is replaced by the "saved" number whether it appears before or after |save. Finally, we see the monk command |e for an in-line equation. The formatted output appears as Panel 2.

Monk provides a full complement of commands for:

- Floating or fixed blocks
- Captioned figures, pictures, tables, and equations
- Centered lines and centered blocks
- Numbered and indented headings and paragraphs
- Numbered, lettered, and variable lists
- Multicolumn output, references, and much more.

Its commands meet or exceed the facilities provided by the troff mm/ms packages and LATEX.

Monk automatically invokes the appropriate preprocessors, recognizing standard tbl, eqn, pic, and prefer, as well as the newer cite and index, whenever it sees their usage in the input text. This frees the writer from the responsibility of remembering to invoke

Panel 2. Sample Picture and Equation

The code that produces this figure and the accompanying text excerpt is described in the body of this paper.

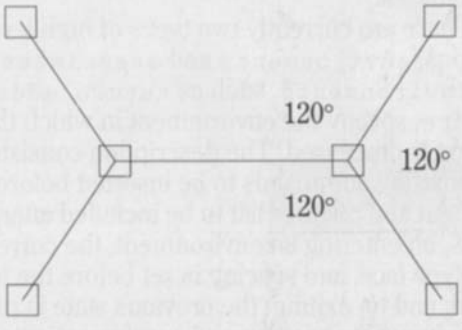


Figure 1. Best Imaginary Network

The length of the imaginary network in Figure 1 is $(\sqrt{3}+1)a$, or $2.7a$.

the necessary preprocessors in the correct order.

Monk also provides tools to help people organize and write documents. One such tool, `monksample`, provides sample documents that contain the basic commands associated with each document type. Thus, `monksample song` produces:

```
|make(song)
|title(Wassail Song)

|verse(|d(repeat following 4 times)
Here we come, a-wassailing,
   among the leaves so green;
Here we come, a-wandering,
   so fair to be seen:)

|refrain(|m(All:)Love and joy come to you,
...
And God send you a happy New Year.)
```

The `|make` command specifies the document type and loads the required document-specific database. The printed sample illustrates a format for title information and uses a type face and point size appropriate for handing out to people who live in nursing homes. Adding the command `|style(singers)` produces a large, clear Helvetica type face appropriate for singers:

Wassail Song

repeat following 4 times
Here we come, a-wassailing,
  among the leaves so green;
Here we come, a-wandering,
  so fair to be seen:

All: Love and joy come to you,
...
And God send you a happy New Year.

Many of the `monk` support tools allow the writer to extract and organize information from the document file to prepare a table of contents, index, and bibliography. All the special structures in `monk`—e.g., headers, figures, and tables—automatically generate information for a table of contents and for page makeup. Based on the writer's specifications, the preprocessor `toc` selects the correct subset to prepare the table of contents. The new postprocessor `pm`[20] adjusts vertical spacing, improves figure and footnote placement, and prevents widows. 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 next page or column.

The preprocessor `index` uses the writer's in-line `|index` commands to gather items for an index and format them. Similarly, the preprocessor `prefer` uses the writer's in-line `|reference` commands to gather references either directly from the text

or from a separate bibliography file. Each preprocessor expands the derived and formatted text, placing it where the writer specified using | `index_placement` and | `reference_placement`, respectively.

There are two tools that are helpful when using other programs to process `monk` files. The program `demonk` removes all the `monk` commands, leaving only the raw text. It is particularly useful with `spell`. The program `monkmerge` is the equivalent of `soelim`, which expands any external file references in place.

The Designer's Point of View

The typographical databases constitute the fundamental difference between `monk` and the macro packages. These databases are coded directly in a high-level language rather than built on top of `troff` assembly code. Information is organized hierarchically into high-level descriptions for writers, low-level translations into typesetting language, and support macros defined directly in that language. Global author-level commands (e.g., `quotation`), as well as those specific to a document type (e.g., a letter's `return address`), are defined using a set of English-like typesetting primitives, such as `indent` and `center_block`. Figure 2 clarifies the role of `monk` and its databases in compiling the writer's commands. `Monk` applies definitions from its high-level and primitive databases to translate the writer's input into preprocessor and `troff` input. The definition for `quotation` is translated into low-level primitives that, in turn, are translated into `troff` input.

The goal in developing the database language has been to move information up into the high-level descriptions, thereby reducing the complexity of the typesetting primitives and the remaining macros. This dichotomy is intended to support both flexibility and extensibility of the typographical database. `Monk` users can customize documents without needing to learn the underlying typesetting language. This not only simplifies the task of creating new document types, but also provides writers and designers with a powerful language for creating new commands. A writer can create new commands and redefine or rename existing commands by including definitions directly in the input text. Moreover, a computer center or a department can create a library of local definitions to modify or augment the standard `monk` databases.

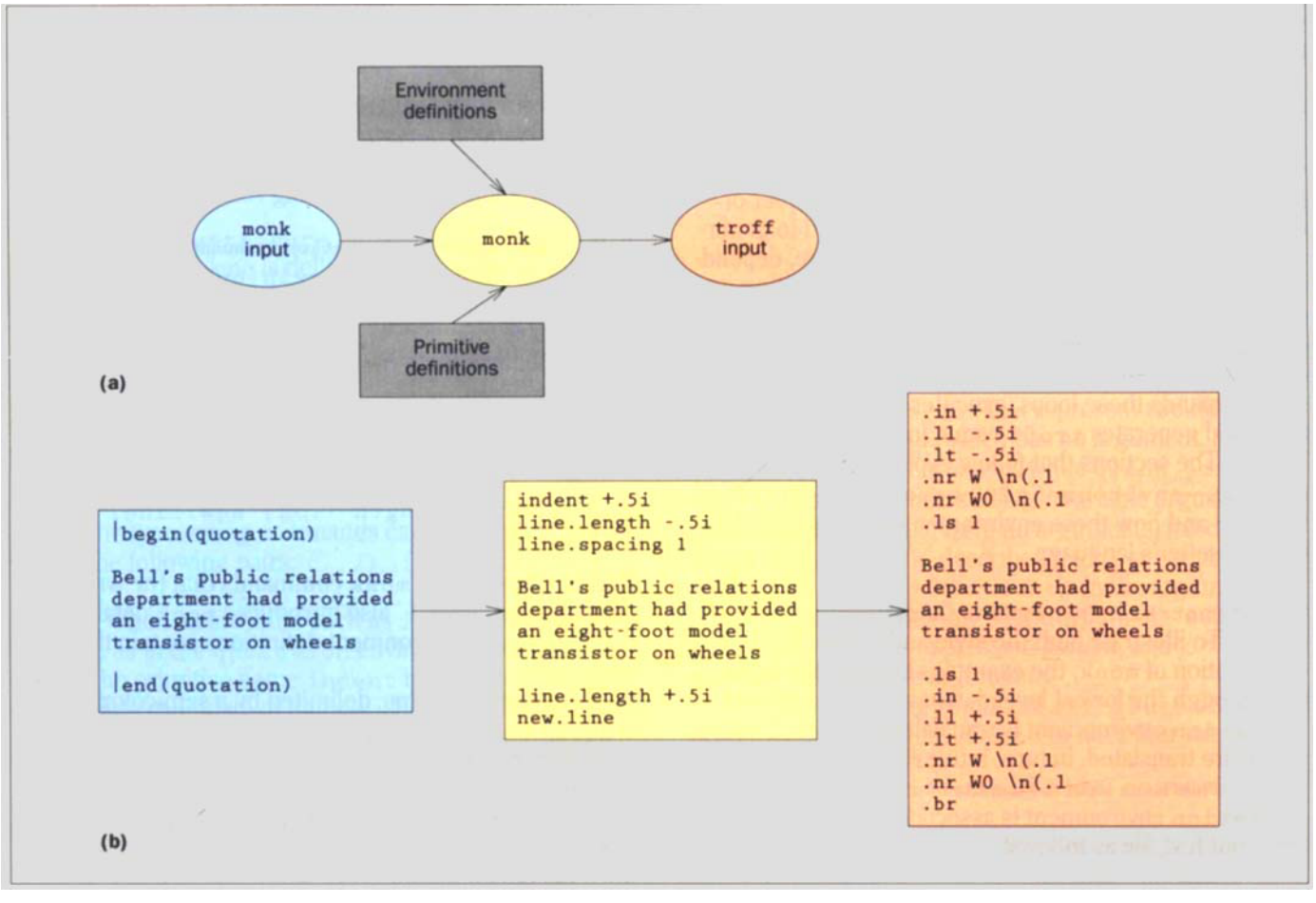
There are currently two types of high-level descriptions: `environments` and `associates`.

`Environments`, such as `return address` and `figure`, specify the environment in which the delimited text is processed. The description consists of a list of formatting commands to be inserted before the text segment and another list to be included after the text. Thus, on entering an environment, the correct page position, type face, and spacing is set before the text is processed; and on exiting, the previous state is restored and any appropriate position and spacing information is added.

`Associates`, such as `author` and `document`, allow pieces of information supplied either by name or by position to be labeled and linked. In a memo, the `author` command provides `name`, `initials`, `location`, `dept`, `room`, and `extension`. The description consists of a list of relevant information, a list of formatting commands to prepend and append to the associate itself, and separate lists to prepend and append to each item of information provided.

Database elements `environment` and `associate` are defined using a set of 63 typesetting primitives and 9 macros. These, in turn, are defined using typesetting command sequences. Currently, both are encoded only in the `troff` language. However, the syntax is intended to support definitions in any typesetting language and the document descriptions themselves are intended to continue producing attractive documents when coupled with appropriate primitive and macro definition files. Only the primitive and macro databases contain typesetting commands.

Many primitives use a variable number of arguments to select from a number of cases. In matching



arguments, `monk` recognizes fixed and variable arguments and compares regular-expression patterns.

The nine macros are written directly in `troff`. Macros cannot be entirely eliminated because, when `monk` is processing input, nothing is yet known about line or page breaks (they must be determined by the underlying typesetter). Macros handle initialization, the tops and bottoms of pages, floating displays, footnotes, multiple columns, and the end of the input text. Future

Figure 2. Compilation of the `monk` environment `quotation`. (a) Relationship of `monk` and its databases. (b) The compilation: `monk` input; the expansion of the environment for quotations; and the expansion of the primitives for indents, line length, and spacing.)

versions of monk will provide an expanded set of primitives that will be used to encode these macros.

In addition to these two environment types and the set of primitives, the monk database supports several programming constructs. It offers for loops, as well as conditional tests based on the status of primitives. Loops permit sophisticated handling of a variable number of authors and arguments. Conditionals are used to generate different typesetting commands for a figure, depending on the current width; for paragraphs, depending on global and local numbering styles; and for type face changes, depending on whether line breaks in the input are retained or lines are filled to the full page width. Monk expands these loops, executes the conditional tests, and generates troff commands.

The sections that follow explain how database elements—environment and associate—are coded in monk, and how these environments are translated into the typesetter's language.

Tracing Quotation from Input to Output

To illustrate both the style and hierarchical organization of monk, the examples below trace input text through the lowest level database. The high-level quotation environment is translated into primitives, which are translated, in turn, into troff commands.

From Monk Input to Formatted Output. The quotation environment is associated with the quote in the input text file as follows:

```
|begin(quotation)
Along with these electronic
demonstrations, Bell's public
relations department had provided
an eight-foot model transistor on
wheels, intended to explain just how
simple the whole thing really was.
|end(quotation)
```

Alternatively, the short form |quotation(Along with ...was.) also specifies the scope of the command. In either case, monk and troff together produce:

Along with these electronic demonstrations, Bell's public relations department had provided an eight-foot model transistor on wheels, intended to explain just how simple the whole thing really was.

Definition of the Quotation Environment. When monk finds the command |begin(quotation) or |quotation in the input text, it applies the most recent definition of quotation. The standard definition available in all document types is:

```
|environment(quotation;
    indent +.5i, line.length -.5i,
    line.spacing 1, new.line;
    line.length +.5i, new.line)
```

Besides introducing monk commands that the writer uses, the character '|' also introduces monk database commands. All environment definitions contain three segments:

- The command name, delimited by a semicolon (quotation in this example)
- The initialization list, again delimited by a semicolon
- The exit list, seen as the last line in this example.

In applying this definition, the formatter enters a new environment, processes text, and then restores the previous environment. When monk enters a quotation, the primitive indent +.5i indents half an inch; line.length -.5i shortens the line by half an inch; line.spacing 1 selects single-spacing; and new.line forces a new line. After processing the text, monk restores the indentation automatically (control of this stack will be discussed below) and forces a new line. Thus, the quote appears separated from the text body on single-spaced lines with wide margins.

Definition of the Indent Primitive. The following monk command defines a stacking primitive named

indent, whose initial value is 0 and whose invocation template takes only a single variable argument:

```
|attribute(indent; stack, default 0;
    $ [
.in $
])
```

The command |attribute introduces the definition of indent, whose name is delimited by the first semicolon. Type information, also delimited by a semicolon, specifies that monk maintain a last-in, first-out stack of indent values, beginning with 'indent 0', so that it can restore the previous state when exiting an environment. This definition provides a template with only one case: a single variable argument named '$'. The clause within the square brackets specifies the associated typesetting commands to be issued. Just as with a writer's monk command, the typesetting commands can be delimited by any of the following pairs: (...), [...], <...>, {...}, "...", '...', `...'.

When an attribute such as indent +.5i appears, as in the quotation environment defined above, the primitive name indent is used to locate an appropriate candidate definition. Then, the templates are examined to determine what specific definition applies to the form. For example, the character '$' matches any single argument. Substituting the matched argument '+.5i' into the typesetting commands delimited by '[' and ']' produces the troff command:

```
.in +.5i
```

Similarly, to translate the environment quotation, monk processes the primitives for line length, line spacing, and new lines to produce the troff input shown in the rightmost box in Figure 2b.

Modifying the Quotation Environment

The current definition of any environment can be changed by copying the original and modifying it. The new definition can be placed in a shared or private file of style information, or it can be placed in-line in the document. The following redefinition of quotation inserts one blank line before a quote and two after it:

```
|environment(quotation;
    indent +.5i, line.length -.5i,
    line.spacing 1, blank.lines;
    line.length +.5i, blank.lines 2)
```

Definition of the blank.lines Primitive. When an environment definition uses a primitive, monk must first find the current definition and then select the case that matches the arguments given. The two instances of blank.lines in quotation represent two different cases: blank.lines, which has no argument, requires a case that takes no argument, while blank.lines 2 requires a case that takes a single variable argument. In general, primitives can have no arguments, fixed arguments (like nobreak, described later), or variable arguments (like the +.5i for indent). In the argument list, '$' or '$*name*' matches any single argument, '$$*pattern*' matches the regular expression *pattern*, and '$*' matches one or more arguments.

The following command attribute defines the primitive named blank.lines to have four cases:

```
|attribute(blank.lines; nostack;
    [
.sp 1     \" one blank line
]     $ [
.sp $     \" variable blank space
]     nobreak [
'sp 1
]     nobreak $ [
'sp $
])
```

Each case specifies the argument list followed by the corresponding typesetting commands delimited by square brackets. The first instance of blank.lines in

Panel 3. One Case of the `minimum.lines` Attribute

```
|attribute(minimum.lines; nostack;
      $ [
'\"      minimum space mechanism
'\"      :B amount of accumulated blank space
'\"      :D amount of accumulated blank space in diversion
'\"      :E position of last minimum space in diversion
'\"      :N position of last minimum space
'\"      }D name of last minimum space diversion
'\"      ;4 amount to space TEMP
.br
|ifvalue diversion on <
.if !'\n(.z'\*(}D' .rr :D :E
.nr ;4 $v
.if !(\n(.d=\n(:D) .nr :E 0      \" different place for sure
.nr ;4 -\n(:Eu     \" remove previous accumulation
.if \n(;4 \{\
.sp \n(;4u
.nr :E +\n(;4u \}
.rm ;4
.ds }D \n(.z
.nr :D \n(.d
>
|ifnotvalue diversion on <
.rr :D :E          \" remove the in-diversion registers
.nr ;4 $v
.if !(\n(nl=\n(:N) .nr :B 0      \" different place for sure
.nr ;4 -\n(:Bu     \" remove previous accumulation, if any
.if \n(;4 \{\
.sp \n(;4u         \" space
.nr :B +\n(;4u \}                \" recompute accumulation
.if !(\n(nl=\n(:N) .nr :B 0     \" different place for sure
.if !(\n(nl=\n(:N) .nr :B 0     \" different place for sure
.rm ;4
.nr :N \n(nl
>)
```

quotation matches the first attribute case. It takes no arguments and always inserts one blank line on output using the troff code:

```
.sp 1
```

The troff code produced is exactly what appears within the square brackets, as there are no arguments to replace.

The second attribute case takes one variable argument and allows the designer to specify exactly the amount of space desired. The instance of blank.lines 2 in quotation matches this case because the single argument '2' matches the special character '$'. Monk replaces each '$' in the troff template with the matched argument '2', and generates:

```
.sp 2
```

The third case takes one fixed argument nobreak, and the fourth takes that fixed argument as well as a variable argument. These two cases generate spacing without forcing a line break.

Definition of the Minimum.lines Primitive. The new quotation definition should not really use the simple primitive blank.lines to control spacing. It needs to use a primitive named minimum.lines, which collapses successive requests for space.

It is not necessary to stare at the intricacies of the definition in Panel 3; its sole purpose is to clarify that the primitives are not simply new names for equally succinct troff commands. Many of the monk primitives translate into complex troff code.

The code for minimum.lines assures that only the maximum amount of space requested is output, rather than the sum of all spacing requested. The single case presented in Panel 3 illustrates the use of the monk conditional tests, ifvalue and ifnotvalue. Conditional tests examine the current state of any stacking primitive and output the bracketed text only if the condition is true. Other examples of conditional tests appear in definitions for footnotes and displays, which are formatted differently depending on the current width, and in the definitions for section numbering, which may include the numbers of parent headings.

Tracing Author From Input to Output

Author is an example of the second type of high-level description: associates. It determines the processing of the author's name, initials, and location.

Information about the two authors of this paper appear in the monk input text as:

```
|author(name "S. L. Murrel",
    initials SLM, location MH)
|author(initials TJK,
    name "T. J. Kowalski", location MH)
```

Here, each category of information about an author is named. Double quotes surround any value that contains embedded spaces (so the information is treated as one argument), and a comma separates each name-value pair.

Author simply saves all this information in a set of strings that are later used when the document's title and author information is printed. It does this by providing typesetting command lists to prepend and append to the environment and to all the name-value pairs within the scope of the environment.

```
|associate(author;
    incr authornum,
    clear string author(authornum)
        initials(authornum)
        location(authornum);

    [name $;
        set string author(authornum) $;]
    [initials $;
        set string initials(authornum) $;]
    [location $;
        set string location(authornum) $;]
```

When `author` is invoked, the first command list is generated; i.e., the author counter is incremented, and the corresponding strings are cleared. The primitives `incr` and `clear` take a variable number of arguments. Here, `incr` has one variable argument, the counter name; `clear` has one fixed argument, `string`, which specifies the type of item, and three variable string arguments to be cleared.

Then, the text within the scope of `author` is read and the associates are matched. `Associates` themselves parallel `environments` in that they provide initialization and exit lists. However, they resemble primitives in that they match and pass primitives in the same way that primitives pass arguments. When an associate—e.g., `name`, `initial`, `location`—is encountered with the correct number of arguments, its initialization and exit lists are inserted. Here, each associate has one variable argument, denoted by the character '`$`'. When a match is found, the corresponding string is set to the value of the argument. Thus, the text `location MH` becomes the primitive

```
set string location(authornum) MH
```

which is later translated into typesetting commands.

Although the definition given above requires that each information segment be named, most database definitions also support information that is specified by position. The standard definition for the `author` associate is more complicated because it does support the entry of information by name and by position. As it reads information, it keeps track of the number of values read, stores the values in a temporary array, and on completion copies them into the appropriate arrays. The definition below illustrates how this is done:

```
|associate(author;
  incr authornum, clear tmpnum,
  clear string tmp(1) tmp(2) tmp(3);
  store string author(authornum) tmp(1),
  store string initials(authornum) tmp(2),
  store string location(authornum) tmp(3);

  [name $;
     incr tmpnum, set string tmp(1) $;]
  [initials $;
     incr tmpnum, set string tmp(2) $;]
  [location $;
     incr tmpnum, set string tmp(3) $;]
  [$;
     incr tmpnum,
     set string tmp(tmpnum) $;])
```

If all the entries are named, then the order remains unimportant; the value is stored in the current array element. However, unnamed entries are identified by position, stored in a temporary array, and on completion copied into the current arrays. For example, the following `monk` input is also correct:

```
|author("S. L. Murrel", SLM, MH)
|author("T. J. Kowalski",
    initials TJK, MH)
```

Here, the first unnamed entry is treated as the author name, the second as the initials, and the third as the location. Moreover, the two styles can be freely mixed, as in:

```
|author(location MH, SLM,
    name "S. L. Murrel")
```

However, if the input is inconsistent, then some information may be lost. For example,

```
|author(MH, SLM, name "S. L. Murrel")
```

first sets the name string to `MH` and then resets it to `S. L. Murrel`.

Definition of the `Store` Primitive. The primitive `store` illustrates the use of:

- The special character '$*' that matches one or more arguments. Thus, it can match a single argument or expand to match four string names.
- A `for` loop that iterates, replacing variables within its scope on each iteration. The loop `|for i in 1 2 3` iterates three times, replacing all instances of the variable `$i` by 1, 2, and 3 in turn. When the special character '$*' matches one or more arguments, the loop iterates once for each argument matched.
- The value '$$*pattern*' that compares arguments to the regular expression *pattern*. This allows explicit matching of single-character variables or those that contain a particular substring.

The primitive `store` supports cases that store any number of values or strings. It provides two cases for values, followed by two cases for strings. That is, the first case in each category uses the regular pattern '.' (pronounced *dot*) to match a single character, while the second matches any other variable. Because `troff` requires a different syntax for handling one- and two-character values and strings, separate cases must be provided. Rather than develop a special syntax, we handled this oddity by using the support for regular-expression matching:

```
|attribute(store; nostack;
      $* $$. [
|for i in $* {
.nr $i \n$$.
}]    $* $number.register [
|for i in $* {
.nr $i \n($number.register
}]    string $* $$. [
|for i in $* {
.ds $i \*$$.
}]    string $* $string.register [
|for i in $* {
.ds $i \*($string.register
}])
```

The argument '$*' in the typesetting commands is replaced by the list of arguments matched. The examples that appear in the definition of `author` match the fourth case (with '$*' matching only one variable) and are translated into a single typesetting command that stores a value from the temporary array into the correct array. Using a macro processor, the convenient array and variable names that appear in the `monk` databases are transformed to the two-letter names required by `troff`.

The expansion of the arguments in `for` loops can be seen in the primitive instance:

```
store string a1 a2 a3 p
```

These arguments match the third case (i.e., '$*' matches `a1`, `a2`, and `a3`; and '$$.' matches `p`). `Monk` produces the following sequence:

```
.ds a1 \*p
.ds a2 \*p
.ds a3 \*p
```

The order of the cases in these definitions is convenient for the database manager. When testing for matches, `monk` applies precedence rules; i.e., it examines the cases with the fixed argument `string` first, so that the expanding argument '$*' does not match the fixed argument `string`.

Programmability: Using `Author` Information

This section illustrates different ways to format the same title and author information. This flexibility is the essence of the document styles provided by `monk`. Here, we discuss simplified first-page formats for a memo and for an *AT&T Technical Journal* article.

As can be seen in Panel 4, the definition for `titlebox` uses the `for` loop construct and the primitive `if` to handle a variable number of author names and locations.

Panel 4. Definition That Uses Loops and Conditionals

```
|environment(titlebox;;

  new.page, fill off, inline off, size +2, space +2,
  font bold, center on, macro title_string,
  font roman, size -2, space -2, inline on,
  fill on, blank.lines,
  |for i in 1 2 3 4 5 6 7 8 9 [
  if begin GE_NUMBER(authornum, $i),
    new.line, size +1, font italics,
    string author($i), font roman, size -1,
    if begin UNEQUAL_STRING_REGISTERS(
     location($i), location($i+1)),
      if begin EQUAL_STRING(location($i), MH),
        text "AT&T Bell Laboratories", new.line,
        text "Murray Hill, NJ 07974-2070", new.line,
      if end,
    if end,
  if end, ]
  center off,

  |ifvalue sawabstract on {
    blank.lines, center on, size +1,
    text italics "ABSTRACT", size -1,
    center off, blank.lines,
    indent +.5i, line.length -.5i, line.spacing 1,
    macro abstract_string, line.length +.5i, indent -.5i,
  }
  minimum.lines)
```

The command `titlebox` does not require any input text; it uses information stored by other `monk` commands. It begins a new page and prints the title string. The title is emboldened, centered, and enlarged by two point sizes; any line breaks in the input are preserved. `Titlebox` then skips a line, restores the point size, and prints the author information.

The `for` loop can handle up to nine authors. `Monk` unrolls the loop: for each author, it prints the name and address, if the address differs from the next author's address or if there is no next author. The `if` that compares the author locations is a primitive that depends on the existence of an `if`-test in the underlying typesetting language.

Finally, the definition centers the header "ABSTRACT," skips a line, increases the margins, sets single spacing, and prints the abstract. The database definition needs to restore all primitives to their previous

Panel 5. Titlebox Tailored for This Publication

```
|environment(titlebox:
  |ifvalue sawtitlebox off {
    spacing on, fill off, font.family techtitle, font bold,
    if else begin LT_NUMBER(title_height,3),
      size 24, space 19,
    if end,
    if else begin,
      size 20, space 17,
    if end,
    indent 8P, blank.lines |.675i,
    macro title_string, indent 0,

    size 9, space 11.5,
    blank.lines |1.5i, line 8P-6p, indent 8P,
    blank.lines |1.505i, putauthors,
    blank.lines |1.5i,
    horizontal.motion number author_width()u+4p,
    line |35P } ;

  |ifvalue sawtitlebox off {
    sawtitlebox on,
    blank.lines |2.2i+.5p,
    column 1 7.6P 8P+PAGE_OFFSET,
    column 2 25P 17P+PAGE_OFFSET,
    column on})
```

values, if they are not automatically restored on exiting an environment.

The tailored definition in Panel 5 uses the same title and author information to prepare the top of an *AT&T Technical Journal* article. After the definition changes to the special font family used by the *Journal*, the title is printed in 24-point type if it is two lines or less, and in 20-point type if it is three lines long. Then, the horizontal rules are drawn, and a separate definition, `putauthors`, places the author names. Finally, all the new-column information for the first page is initialized: a narrow column 1 for the biography, and a wide column 2 for the abstract. The conditional tests on the primitive `sawtitlebox` allow the designer to format text correctly in spite of missing or extra elements.

Future Work

Future expansion of the databases will involve both the remaining macros and page layout.

The set of primitives needs to be enhanced so that the macros can also be defined by primitives, rather than directly in the formatting language. Though this requires primitives of increasing complexity, the benefits of high-level macro definitions justify it. With this

addition, a novice who is unfamiliar with the underlying formatter could change the macros themselves.

Page layout is currently embedded in the other definitions. It is important to separate the description of the page from the description of section formatting. This will make sophisticated and flexible page layout easier.

References

1. B. K. Reid, "Scribe: A Document Specification Language and its Compiler," CMU-CS-81-100, Department of Computer Science, Carnegie-Mellon University, October 1980.
2. D. E. Knuth, *The TEXbook*, Addison-Wesley Publishing Company, Reading, Massachusetts, June 1986.
3. L. Lamport, *A Document Preparation System LATEX*, Addison-Wesley Publishing Company, Reading, Massachusetts, September 1986.
4. AT&T, *UNIX™ System V DOCUMENTERS WORKBENCH™ Software Release 1.0 Macro Package Reference*, CIC No. 307-152, AT&T Customer Information Center, Indianapolis, Indiana, 1984.
5. J. F. Ossanna, "NROFF/TROFF User's Manual," *UNIX Time-Sharing System: UNIX Programmer's Manual*, Seventh Edition, Vol. 2, Holt, Rinehart and Winston, New York, January 1979, pp. 196-229.
6. B. W. Kernighan, "A TROFF Tutorial," *UNIX Time-Sharing System: UNIX Programmer's Manual*, Seventh Edition, Vol. 2, Holt, Rinehart and Winston, New York, January 1979, pp. 230-244.
7. B. W. Kernighan, "A Typesetter-Independent TROFF," Computer Science Technical Report 97, AT&T Bell Laboratories, Murray Hill, New Jersey, 1981.
8. AT&T, *UNIX™ System V DOCUMENTERS WORKBENCH™ Software Release 1.0 Preprocessor Reference*, CIC No. 307-153, AT&T Customer Information Center, Indianapolis, Indiana, 1984.
9. M. E. Lesk, "TBL—A Program to Format Tables," *UNIX Time-Sharing System: UNIX Programmer's Manual*, Seventh Edition, Vol. 2, Holt, Rinehart and Winston, New York, January 1979, pp. 157-174.
10. B. W. Kernighan and L. L. Cherry, "A System for Typesetting Mathematics," *Communications for the ACM*, Vol. 18, No. 3, March 1975, pp. 151-157.
11. B. W. Kernighan, "PIC—A Language for Typesetting Graphics," *Software Practice & Experience*, Vol. 12, January 1982, pp. 1-20.
12. *UNIX Programmer's Manual*, Ninth Edition, Vol. 1, M. D. McIlroy (ed.), AT&T Bell Laboratories, Murray Hill, New Jersey, September 1986.
13. 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.
14. C. J. Van Wyck, "A High-Level Language for Specifying Pictures," *ACM Transactions on Graphics*, Vol. 1, No. 2, January 1982, pp. 1-20.
15. T. Pavlidis, "PED: A Distributed Graphics Editor," *Proceedings of the Graphics Interface*, May 1984, pp. 75-79.
16. *UNIX Programmer's Manual*, Tenth Edition, Vol. 2, A. G. Hume (ed.), AT&T Bell Laboratories, Murray Hill, New Jersey, to be published, 1989.
17. 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, Vol. 2, Holt, Rinehart and Winston, New York, January 1979, pp. 125-145.
18. AT&T, *UNIX™ System V DOCUMENTERS WORKBENCH™ Software Release 1.0 Text Formatters Reference*, CIC No. 307-151, AT&T Customer Information Center, Indianapolis, Indiana, 1984.
19. J. Bernstein, *Three Degrees Above Zero*, Charles Scribner's Sons, New York, 1984.
20. 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.

Biographies (continued)

examining how programmer productivity can be increased. His interests also include artificial intelligence, operating systems, computer-aided design, text-processing environments, and real-time systems. Mr. Kowalski joined the company in 1978. He has a B.S.E. from the University of Michigan and holds the M.S.E.E. and Ph.D. degrees in electrical engineering from Carnegie-Mellon University.

(Manuscript received May 1, 1989)