AT&T

310-005
Issue 1

UNIX™ System V DOCUMENTER'S WORKBENCH™ Software Release 2.0

Technical Discussion and Reference Manual

NOTICE

The information in this document is subject to change without notice. AT&T assumes no responsibility for any errors that may appear in this document.

L-244067-24

Table of Contents

Introduction

The DOCUMENTER'S WORKBENCH programs provide tools of unusual precision for arranging and modifying text. The family of programs included in the workbench allows you considerable control over text and a wide range of functions. Without going through a maze of menus, you are offered a large collection of text formatting commands that will help you draft and revise text. You can also define and name your own formatting commands using conventions any beginner can quickly master.

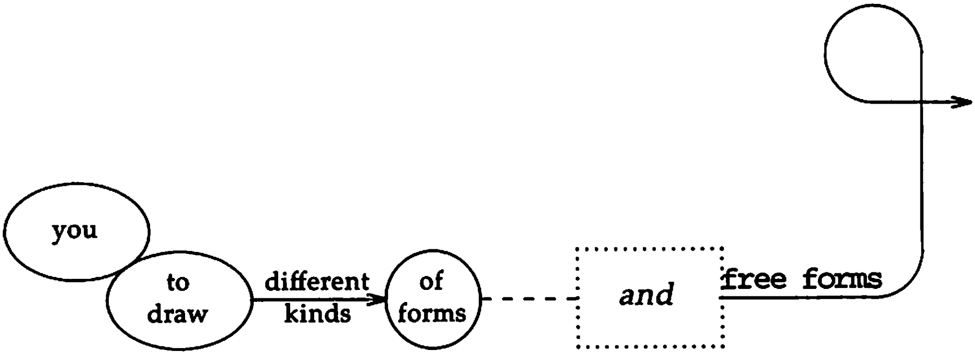
You can make your own shapes, characters, pictures, figures, and form memos and letters and store them to be quickly and easily used for other tasks. Most important, the DOCUMENTER'S WORKBENCH programs are an integral part of the UNIX operating system, which allows you to write a single script to edit several files at once, check spelling, analyze your writing style, communicate with other computers, and develop your own tools for sophisticated text processing. On the other hand, if you only need a few tools for writing simple memos and other uncomplicated text processing tasks, you can use the programs that way, too.

A few of the DOCUMENTER'S WORKBENCH system's tools are **tbl**, **pic**, and **eqn**. The table-drawing tool, **tbl**,

makes all sorts of tables,		
and	what	is
best	about	**tbl**
is	that	once
you	make	a
table	you	like,

you can save it for later use.

The picture drawing tool, **pic**, allows



eqn, the tool for preparing equations, enables you to present accurate mathematical notation:

$$\sum_{i=0}^{\infty} x_i = \frac{\pi}{2}$$

These tools are sensible and comfortable to use. The mathematical notation given above, for example, was produced from this input:

```
sum from i=0 to infinity x sub i = pi over 2
```

eqn allows you to describe equations as you would describe them to another person. You will find that a brief exposure to the **eqn** language will enable you to use it without referring to the tutorial.

Following a first draft, you can revise your copy with tools such as **diffmk** and **hyphen**, and you can check the formatting requests you have used with **checkmm**. When you have completed your document, you can compile a table of contents and an index.

Formatting Commands

To use the DOCUMENTER'S WORKBENCH programs, you need only know a few basic concepts. The primary ones you can guess by looking at the Sampler. Unlike a simple text processor, the DOCUMENTER'S WORKBENCH programs enable you to specify to a fine degree precisely what you want. You type in your text and intersperse formatting commands that state how you'd like your document to look. One glance at the Sampler shows the obvious difference between text lines and formatting commands (control lines). But it is worth emphasizing: control lines begin with a dot (.), and the dot must occupy the leftmost position on the line. Only control lines can begin with a dot because it tells the computer that what follows on the line is not text to be printed, but a formatting command.

There are two types of formatting commands: requests and macros. You will learn more about these as you read the tutorials. For now, you only need to know two things about them.

First, most request names are one or two lower-case letters, and most macro names are one or two upper-case letters. Both requests and macros, of course, are lines that begin with a dot.

Second, both requests and macros are "open to argument." That is, if you don't like the way they behave (or rather, make your text behave), you can change them. For example, somewhere in your text you need more than the one blank line that you get with the **.sp** request. You can make this happen by following the **.sp** request with an argument. If you need three and a half blank lines, you would follow the **.sp** with a single space and then an argument:

```
These lines are separated by
.sp 3.5
three and a half spaces.
```

The formatted version of these three input lines would look like this:

These lines are separated by

three and a half spaces.

Similarly, you can give a macro an argument. If you want a display of text centered in the middle of normal text, you would use the display macro .DS and follow it with the centering argument, C. Typing

```
.DS C
This text is centered.
.DE
```

will give you this:

This text is centered.

.DS and .sp will accept other arguments as well, effectively multiplying the number of fine-tuning knobs at your disposal.

Requests and macros can be followed by comments, which are frequently used as reminders. Like a request or macro, a comment does not appear in output. Here's how they look:

```
.if n .sp 3.5     \" if nroff is the formatter, give me three and a half inches
.if t .sp 1.75    \" but if troff is the formatter, I'd like half that much
```

The only things separating the request from the comment are spaces and the string \". Because the line begins with a dot, we don't need one preceding the \" string. The comment will be protected from machine interpretation all the same. Do not separate comments from requests with a tab; only spaces will be recognized as valid request or argument delimiters.

One additional formatting command will complete a basic knowledge of the DOCUMENTER'S WORKBENCH language: the escape sequence (so-called because it always includes the escape character, or backslash: \). Like requests and macros, an escape sequence controls all text that follows it, and some require arguments. The differences between an escape sequence and requests and macros are that an escape sequence does not appear on a line that begins with a dot, and no space intervenes between it and its arguments.

The escape sequence's advantage over the other formatting commands is its ability to get into tight places. If you want to change a single word or just one character without affecting anything else on a line, you can target just that word or character by preceding it with an escape sequence. Let's say you want to print a sentence using a bold font, but one word in the sentence needs to be in italic. You can do this by using the escape sequence together with requests and macros. You would type the following:

```
.ft B
These are bold words with an \fIitalic\fB word among them.
.ft R
```

The printed results would look like this:

> **These are bold words with an** *italic* **word among them.**

That is, after you used the font request (.ft) and its argument for bold (**B**), you can make an exception to bold by using the font escape sequence (**\f**) and its argument for italic typeface (**I**).

If you have special formatting requirements that the standard library of requests, macros, and escape sequences cannot satisfy, you can fashion your own macros and escape sequences and save them for later use. As you will see later in this tutorial, the conventions for doing this are easy to learn and use.

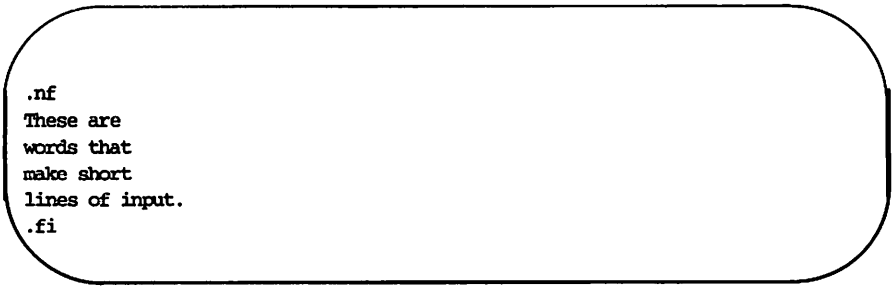
Line Filling

To this point you have seen before and after versions of files: the ones you type in and the ones you see come out in print. These two versions, as you may know, are called input and output. An important effect of processing (what happens to your input on its way to becoming output) is that the number of lines you get out is not always the same as the number you typed in. You know, for example, that all the control lines will disappear and that the spaces they once occupied will vanish. The escape sequences (and the spaces they once occupied) also will disappear when your input is processed. Similarly, the text in your input file will be formatted, line by line, from left to right, according to the format you specified.

This activity of filling the page with formatted text is called line filling. That is, a line will start filling from the left-hand margin and will continue until it gets to the right-hand margin where it will drop to the next line at the left margin, and so forth. When each line reaches the right-hand margin a number of things will be done, depending on your instructions. The right-hand margin will be aligned (adjustment) unless you ask for non-adjustment (**.na**). Words will be hyphenated according to rules you specify, or will not be hyphenated if that's what you want.

The practical point to be made about line filling is that you can turn it off or on with the requests **.fi** (fill) and **.nf** (no fill). If you would like to set up columns (or any other configuration of words for that matter) and have them come out exactly as you typed them in, you can request no fill mode.

For example, if you processed the following file:

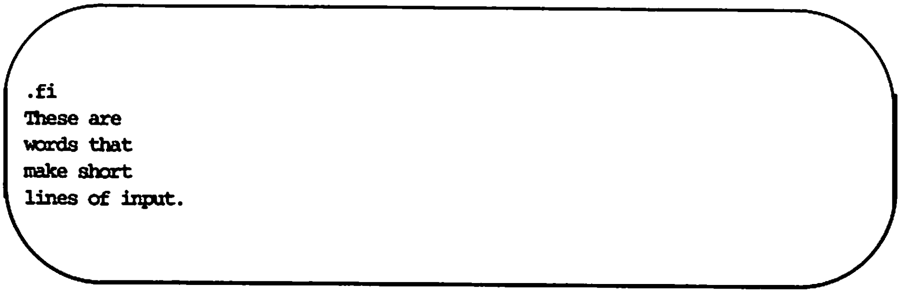


```
.nf
These are
words that
make short
lines of input.
.fi
```

the output would look like this:

These are
words that
make short
lines of input.

Using the fill request, however, would cause the formatter to automatically fill each line with as many words as will fit on it. Here is the contents of the input file:



```
.fi
These are
words that
make short
lines of input.
```

And now the formatted output:

These are words that make short lines of input.

Processing Input

Once you are ready to process your input files, you name the tools you used and your file or files, and the UNIX system will prepare them and send them to a printer. If you are preparing correspondence-quality text, you would be working with **nroff** (pronounced EN-roff) requests and escape sequences. If you were also using the **mm** (memorandum macros) macros, you would need to tell **nroff** that you had used them. The line you would type looks like this:

```
nroff −mm my.file
```

This is called a command line. The minus sign (−) tells the system that **mm** is a macro package it must use when it processes **my.file**. The system will then consult **nroff** and **mm** to fulfill the requests, macros, and escape sequences in your text. Once your files have been processed, you will have two versions: input and output. Processing never affects your original input file.

The output of files that are processed with **mm** or **nroff** can be sent to any printer. This is also known as piping the output to a printer, and is indicated in the command line by a vertical bar (|) between the command that formats the input file and the name of the printer.

```
nroff −mm my.file | printer
```

After entering this command line, your formatted copy will be output on the printer.

The files that are processed with **mmt** or **troff** are piped to a typesetter:

```
troff −mm my.file | typesetter
```

Only devices capable of producing typographical copy—phototypesetters, laser printers, or sophisticated dot-matrix printers—may be used to produce **troff** output.

The command line should reflect the formatting commands in your file. If you use requests that are unique to **troff** in your file, you must name **troff** in the command line. Think of the command line as a declaration of the activity in each file. The rules for how you make that declaration are loose in some cases and rigid in others. Each tutorial will list the appropriate command lines to process your text. The following table shows a selection of command lines to give you a feel for the rules.

	Command Lines	
Tools used	**Customary syntax**	**Alternative syntax**
mm macros	nroff −mm *file*	mm *file*
nroff	nroff *file*	*none*
nroff and mm	nroff −mm *file*	mm *file*
neqn, nroff and mm	neqn *file* | nroff −mm	mm −e *file*
neqn, nroff, tbl and mm	tbl *file* | neqn| nroff −mm	mm −e *file*
troff and mm	troff −mm *file*	mmt *file*
tbl, nroff, and mm	tbl *file* | nroff −mm	mm −t *file*
pic, troff, and mm	pic *file* | troff −mm	mmt −p *file*
eqn, pic, troff, and mm	pic *file* | eqn | troff −mm	mmt −e −p *file*
tbl, eqn, pic, troff and mm	pic *file* | tbl | eqn | troff −mm	mmt −e −p −t *file*
grap, troff, and mm	grap *file* | pic | troff −mm	mmt −g *file*

Software Notes

The following notes may help you avoid problems or to troubleshoot when problems do occur.

- The **checkmm** command will flag some business letter macros as possible errors, even though a file containing the letter macros will format properly.

- The **checkmm** command will flag as possible errors the macros that produce labeled footnotes, if they occur inside lists. A file containing such a sequence of macros will format properly, however.

- The chapter "The Preprocessor **eqn**" in the *User's Guide* suggests running the **eqn** preprocessor on the **/usr/lib/dwb/samples/eqn.stats** file. If you do this you will get the following warning:

  ```
  eqn warning: unquoted troff command
  ```

  The file **eqn.stats** will format properly, however.

- A **README** file is supplied with the binary version of the DOCUMENTER'S WORKBENCH Software for the 3B5 Computer and the 3B2 Computer. This file incorrectly states that there are only three macro packages included in the package, although it correctly names the four packages that are included: **mm, mv, mptx,** and **man.**

- Use of the **mv** macro package inhibits true constant width spacing in the output text. If you need to show constant width spacing in a viewgraph you can simulate it by entering the **troff** request

  ```
  .ss 12
  ```

  before the text you want to appear in constant width spacing. From that point, standard **troff** inter-word spacing will be used. You can restore the standard **mv** inter-word spacing by entering the request

  ```
  .ss 16
  ```

  at the point you want it to resume.

The DOCUMENTER'S WORKBENCH Software Sampler

The variety of text processing that the DOCUMENTER'S WORKBENCH Software offers can be seen in the samples that appear in the following pages. The input files shown in screens are those you type at the terminal. Each input file is named, and the command you would use to process each is given after its screen. On the facing page is the processed file: the version you would get from your printer or phototypesetter.

NOTE The examples of output in this sampler were processed for a standard 8.5x11-inch page and have been photo-reduced to fit on the 6.25x8.5-inch pages of this guide. As a result, the type sizes in the output examples will not correspond to the formatting instructions shown in the input examples.

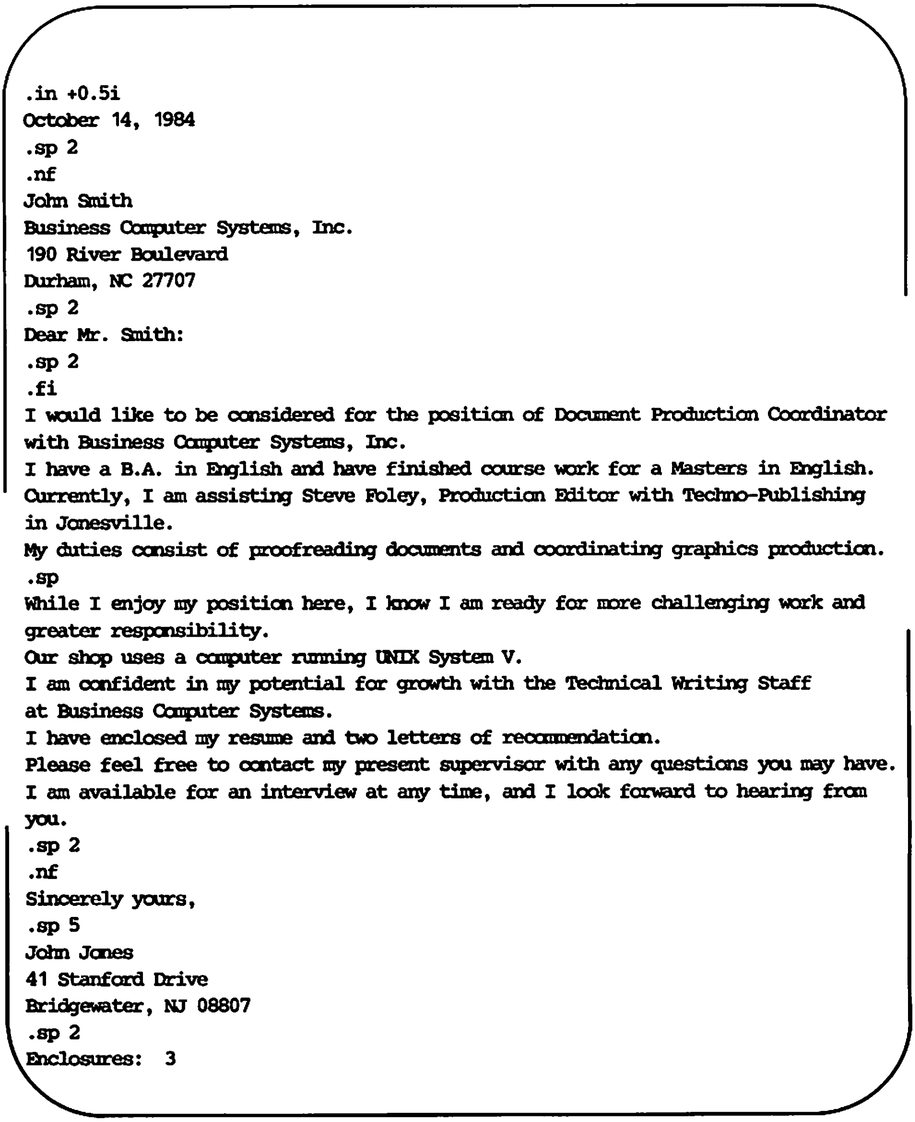
The input files are in **/usr/lib/dwb/samples**. You can copy any of the files into your own directory to use as the basis for your text processing activities. For example, the following sequence of commands copies the first sample, **nroff.letter**, into the directory **/usr/yourlogin**.

```
$cd /usr/yourlogin <RETURN>
$cp /usr/lib/dwb/samples/nroff.letter nroff.letter <RETURN>
```

Many printers are capable of producing most of these samples. Beginning with the **eqn.stats** file, however, producing the sample outputs will require sophisticated printers or phototypesetters that have multiple fonts and graphics capabilities. Because of the wide variety of printers, the commands that follow the screens of input do not name any specifically. You will have to check with your system administrator to determine the printers (and printer names) available on your system.

If you want to read more about the formatting commands used in the Sampler, refer to the alphabetical request index or summary beginning "The **nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual*. Some of these samples also appear in the *Handbook for New Users*, which gives more detail about the function of each macro or request in the input files.

File: nroff.letter



```
.in +0.5i
October 14, 1984
.sp 2
.nf
John Smith
Business Computer Systems, Inc.
190 River Boulevard
Durham, NC 27707
.sp 2
Dear Mr. Smith:
.sp 2
.fi
I would like to be considered for the position of Document Production Coordinator
with Business Computer Systems, Inc.
I have a B.A. in English and have finished course work for a Masters in English.
Currently, I am assisting Steve Foley, Production Editor with Techno-Publishing
in Jonesville.
My duties consist of proofreading documents and coordinating graphics production.
.sp
While I enjoy my position here, I know I am ready for more challenging work and
greater responsibility.
Our shop uses a computer running UNIX System V.
I am confident in my potential for growth with the Technical Writing Staff
at Business Computer Systems.
I have enclosed my resume and two letters of recommendation.
Please feel free to contact my present supervisor with any questions you may have.
I am available for an interview at any time, and I look forward to hearing from
you.
.sp 2
.nf
Sincerely yours,
.sp 5
John Jones
41 Stanford Drive
Bridgewater, NJ 08807
.sp 2
Enclosures:  3
```

Command line: **nroff nroff.letter** |*printer*

October 14, 1984

John Smith
Business Computer Systems, Inc.
190 River Boulevard
Durham, NC 27707

Dear Mr. Smith:

I would like to be considered for the position of Document Production Coordinator with Business Computer Systems, Inc. I have a B.A. in English and have finished course work for a Masters in English. Currently, I am assisting Steve Foley, Production Editor with Techno-Publishing in Jonesville. My duties consist of proofreading documents and coordinating graphics production.

While I enjoy my position here, I know I am ready for more challenging work and greater responsibility. Our shop uses a computer running UNIX System V. I am confident in my potential for growth with the Technical Writing Staff at Business Computer Systems. I have enclosed my resume and two letters of recommendation. Please feel free to contact my present supervisor with any questions you may have. I am available for an interview at any time, and I look forward to hearing from you.

Sincerely yours,

John Jones
41 Stanford Drive
Bridgewater, NJ 08807

Enclosures: 3

File: mm.report

```
.TL
Work Progress Report -- Second Quarter 1984
.AF "Business Computer Systems, Inc."
.AU "W. Williams" WW XF 665414 5398 7-123 bailey!www
.MT 0
.HU "Writing Assignments"
.P
I started work with the Technical Writing Staff on April 16.
My writing assignments are:
.BL
.LI
Documentation for the BCS Fortran compiler
.DL
.LI
I collected materials relevant to implementing programming languages
on the UNIX*
.FS *
Trademark of AT&T
.FE
system.
.LE
.LI
Documentation for the Distributed Transaction Processing System (DTPS) 2.0
.DL
.LI
I reviewed DTPS requirements, outstanding complaints about DTPS, and users'
suggestions for improving DTPS documentation.
.LE
.LE
.HU "Other Activities"
.P
On June 16, I went to a conference, "Writing About Computers," at Acme State
College.
.SG
```

Command line: **mm −Tlp mm.report** |*printer*

Business Computer Systems, Inc.

subject: Work Progress Report --
Second Quarter 1984

date: December 4, 1985

from: W. Williams
XF 665414
7-123 x5398
bailey!www

Writing Assignments

I started work with the Technical Writing Staff on April 16. My writing assignments are:

- Documentation for the BCS Fortran compiler
  - I collected materials relevant to implementing programming languages on the UNIX* system.
- Documentation for the Distributed Transaction Processing System (DTPS) 2.0
  - I reviewed DTPS requirements, outstanding complaints about DTPS, and users' suggestions for improving DTPS documentation.

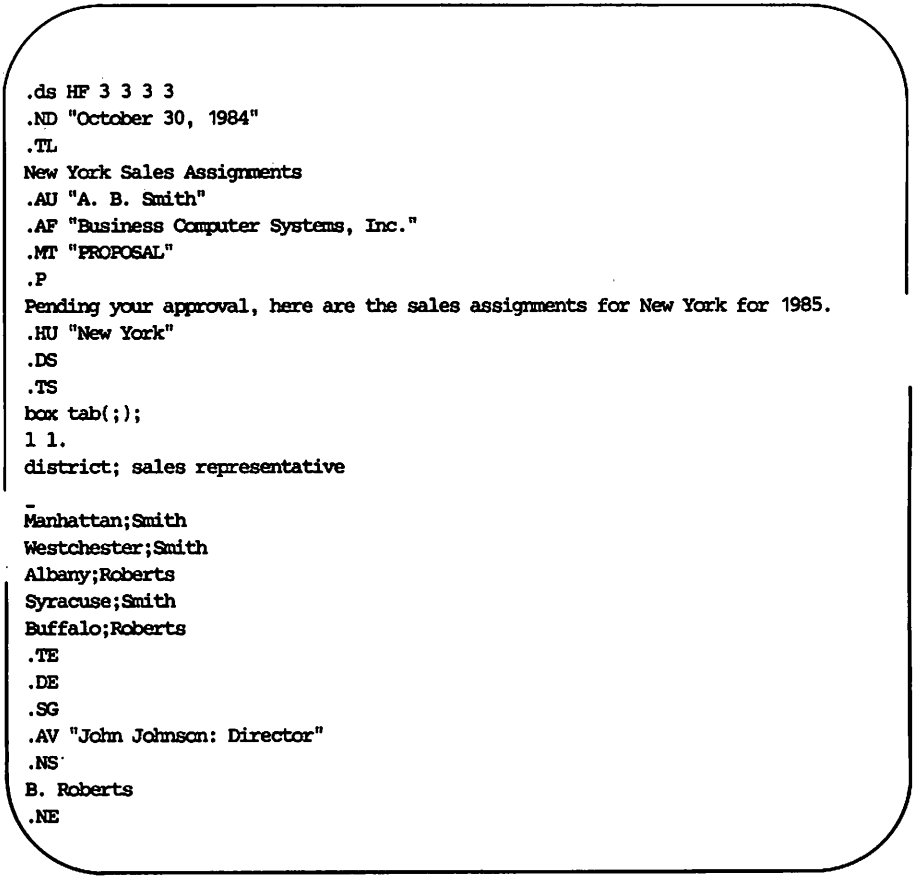
Other Activities

On June 16, I went to a conference, "Writing About Computers," at Acme State College.

W. Williams

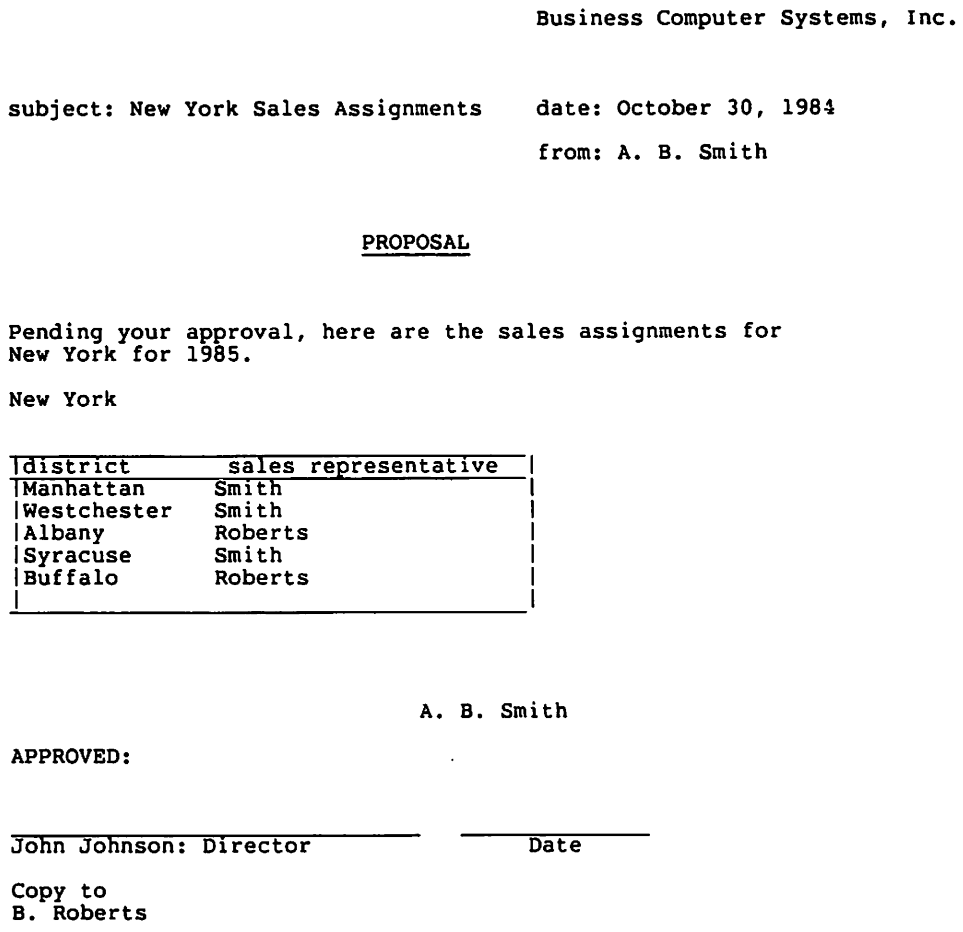
* Trademark of AT&T

File: mm.sales



```
.ds HF 3 3 3 3
.ND "October 30, 1984"
.TL
New York Sales Assignments
.AU "A. B. Smith"
.AF "Business Computer Systems, Inc."
.MT "PROPOSAL"
.P
Pending your approval, here are the sales assignments for New York for 1985.
.HU "New York"
.DS
.TS
box tab(;);
1 1.
district; sales representative
_
Manhattan;Smith
Westchester;Smith
Albany;Roberts
Syracuse;Smith
Buffalo;Roberts
.TE
.DE
.SG
.AV "John Johnson: Director"
.NS
B. Roberts
.NE
```

Command line: **mm −t −Tlp mm.sales** |*printer*



Business Computer Systems, Inc.

subject: New York Sales Assignments

date: October 30, 1984

from: A. B. Smith

PROPOSAL

Pending your approval, here are the sales assignments for New York for 1985.

New York

district	sales representative
Manhattan	Smith
Westchester	Smith
Albany	Roberts
Syracuse	Smith
Buffalo	Roberts

A. B. Smith

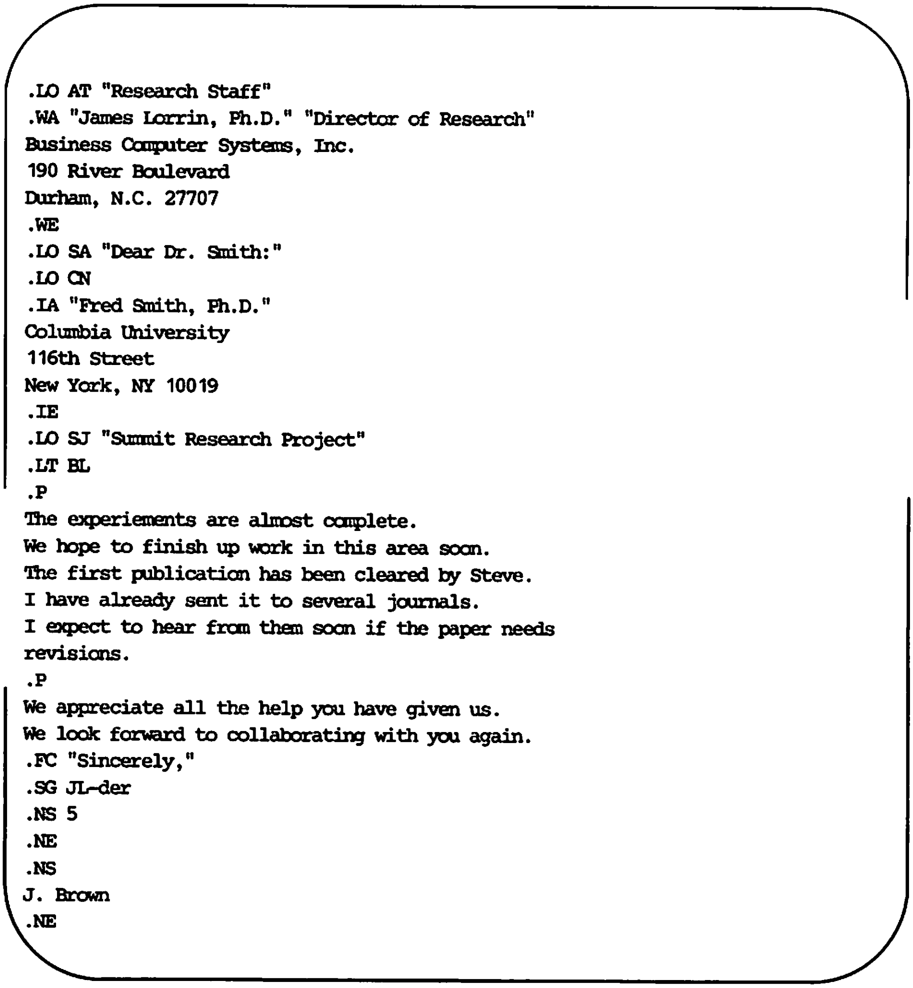
APPROVED:

John Johnson: Director

Date

Copy to
B. Roberts

File: mm.letter



```
.LO AT "Research Staff"
.WA "James Lorrin, Ph.D." "Director of Research"
Business Computer Systems, Inc.
190 River Boulevard
Durham, N.C. 27707
.WE
.LO SA "Dear Dr. Smith:"
.LO CN
.IA "Fred Smith, Ph.D."
Columbia University
116th Street
New York, NY 10019
.IE
.LO SJ "Summit Research Project"
.LT BL
.P
The experiements are almost complete.
We hope to finish up work in this area soon.
The first publication has been cleared by Steve.
I have already sent it to several journals.
I expect to hear from them soon if the paper needs
revisions.
.P
We appreciate all the help you have given us.
We look forward to collaborating with you again.
.FC "Sincerely,"
.SG JL-der
.NS 5
.NE
.NS
J. Brown
.NE
```

Command line: **mm −Tlp mm.letter** | *printer*

Business Computer Systems, Inc.
190 River Boulevard
Durham, N.C. 27707
December 4, 1985

CONFIDENTIAL

Fred Smith, Ph.D.
Columbia University
116th Street
New York, NY 10019

ATTENTION: Research Staff

Dear Dr. Smith:

SUBJECT: Summit Research Project

The experiements are almost complete. We hope to finish up work in this area soon. The first publication has been cleared by Steve. I have already sent it to several journals. I expect to hear from them soon if the paper needs revisions.

We appreciate all the help you have given us. We look forward to collaborating with you again.

Sincerely,

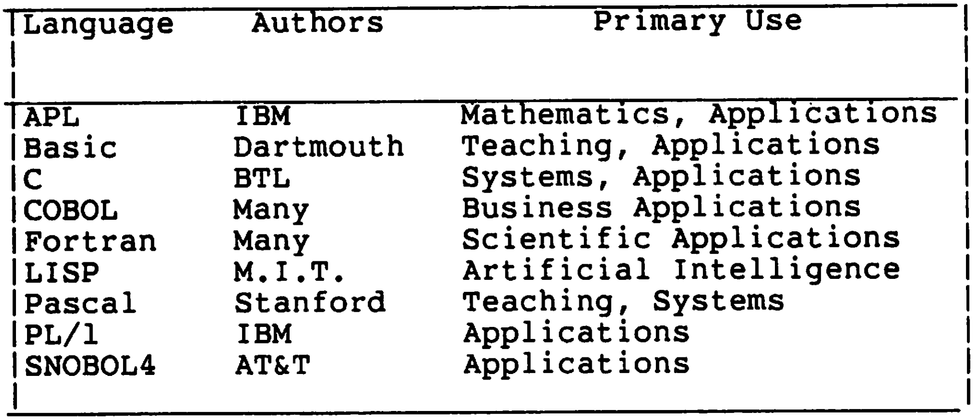
James Lorrin, Ph.D.
Director of Research

JL-der
Enc.
Copy to
J. Brown

File: tbl.language

```
.TS
box,center;
c c c
l l l.
Language<TAB>Authors<TAB>Primary Use
.sp
_
APL<TAB>IBM<TAB>Mathematics, Applications
Basic<TAB>Dartmouth<TAB>Teaching, Applications
C<TAB>BTL<TAB>Systems, Applications
COBOL<TAB>Many<TAB>Business Applications
Fortran<TAB>Many<TAB>Scientific Applications
LISP<TAB>M.I.T.<TAB>Artificial Intelligence
Pascal<TAB>Stanford<TAB>Teaching, Systems
PL/1<TAB>IBM<TAB>Applications
SNOBOL4<TAB>AT&T<TAB>Applications
.TE
```

Command line: **tbl −TX tbl.language | nroff −mm −Tlp | col |** *printer*



Language	Authors	Primary Use
APL	IBM	Mathematics, Applications
Basic	Dartmouth	Teaching, Applications
C	BTL	Systems, Applications
COBOL	Many	Business Applications
Fortran	Many	Scientific Applications
LISP	M.I.T.	Artificial Intelligence
Pascal	Stanford	Teaching, Systems
PL/1	IBM	Applications
SNOBOL4	AT&T	Applications

File: tbl.bridges

```
.TS
box,center;
c s s
c | c |.c
l | l | n.
Major New York Bridges
_
Bridge<TAB>Designer<TAB>Length
_
Brooklyn<TAB>J. A. Roebling<TAB>1595
Manhattan<TAB>G. Lindenthal<TAB>1470
Williamsburg<TAB>L. L. Buck<TAB>1600
_
Queensborough<TAB>Palmer &<TAB>1182
<TAB>  Hornbostel
_
<TAB><TAB>1380
Triborough<TAB>O. H. Ammann<TAB>_
<TAB><TAB>383
_
Bronx Whitestone<TAB>O. H. Ammann<TAB>2300
Throgs Neck<TAB>O. H. Ammann<TAB>1800
.TE
```

Command line: **mm −t −Tlp tbl.bridges** | *printer*

Major New York Bridges		
Bridge	Designer	Length
Brooklyn	J. A. Roebling	1595
Manhattan	G. Lindenthal	1470
Williamsburg	L. L. Buck	1600
Queensborough	Palmer & Hornbostel	1182
Triborough	O. H. Ammann	1380 383
Bronx Whitestone	O. H. Ammann	2300
Throgs Neck	O. H. Ammann	1800

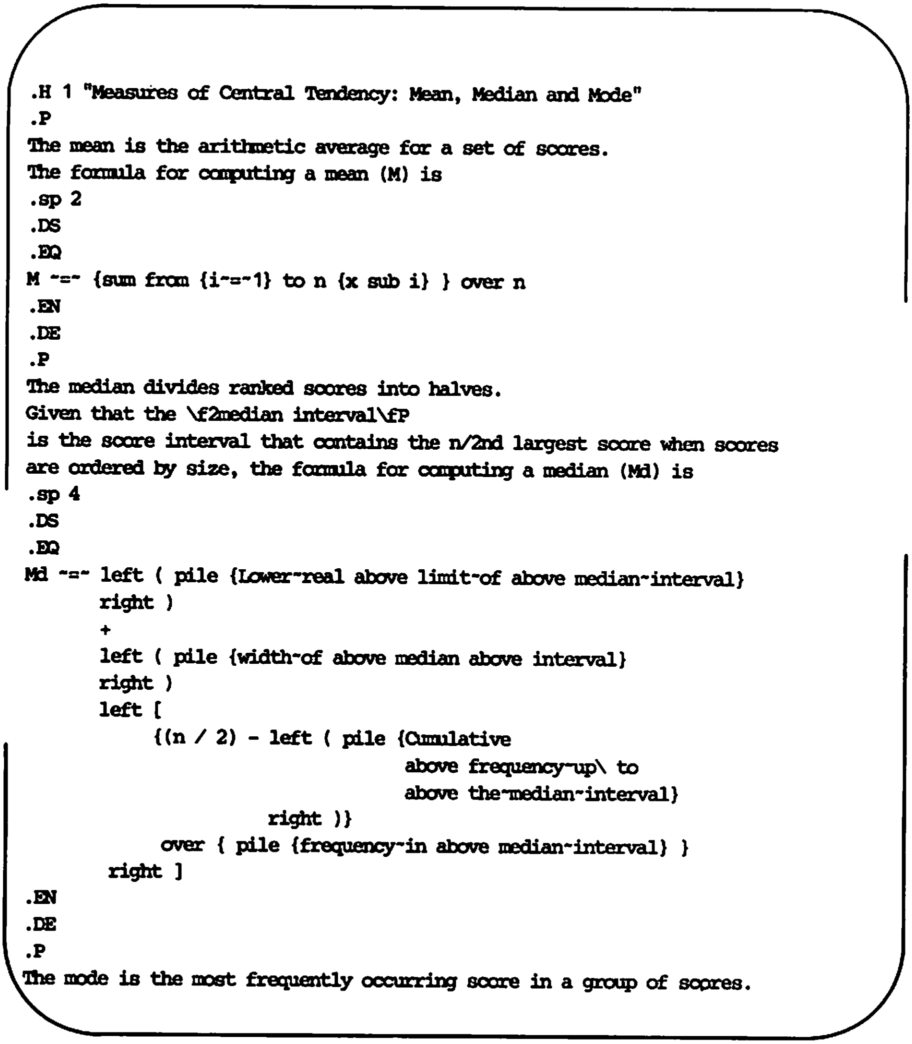
File: tbl.pres

```
.ad n
.TS
center box tab(;);
c s s s s
c | c | l | l | l
l | l | l | l | l.
American Presidents
=
Name;Party;Term;Election-Oppnts;Notes
_
T{
Gerald
R. Ford
T};Republican;1974-1977;Never elected;T{
Became V.P. when
Agnew resigned
T}
_
T{
Jimmy
Carter
T};Democratic;1977-1981;1976-Ford;T{
Negotiated
Camp David
treaties
T}
_
T{
Ronald
Reagan
T};Republican;1981-;T{
1980-Carter
     Anderson
1984-Mondale
T};T{
Oldest
President
T}
.TE
```

Command line: **mm −t −Tlp tbl.pres** | *printer*

American Presidents				
Name	Party	Term	Election-Oppnts	Notes
Gerald R. Ford	Republican	1974-1977	Never elected	Became V.P. when Agnew resigned
Jimmy Carter	Democratic	1977-1981	1976-Ford	Negotiated Camp David treaties
Ronald Reagan	Republican	1981-	1980-Carter Anderson 1984-Mondale	Oldest President

File: eqn.stats



```
.H 1 "Measures of Central Tendency: Mean, Median and Mode"
.P
The mean is the arithmetic average for a set of scores.
The formula for computing a mean (M) is
.sp 2
.DS
.EQ
M ~=~ {sum from {i~=~1} to n {x sub i} } over n
.EN
.DE
.P
The median divides ranked scores into halves.
Given that the \f2median interval\fP
is the score interval that contains the n/2nd largest score when scores
are ordered by size, the formula for computing a median (Md) is
.sp 4
.DS
.EQ
Md ~=~ left ( pile {Lower~real above limit~of above median~interval}
       right )
       +
       left ( pile {width~of above median above interval}
       right )
       left [
            {(n / 2) - left ( pile {Cumulative
                                   above frequency~up\ to
                                   above the~median~interval}
                      right )}
             over { pile {frequency~in above median~interval} }
        right ]
.EN
.DE
.P
The mode is the most frequently occurring score in a group of scores.
```

Command line: **eqn eqn.stats | troff –mm |** *typesetter*

1. Measures of Central Tendency: Mean, Median and Mode

The mean is the arithmetic average for a set of scores. The formula for computing a mean (M) is

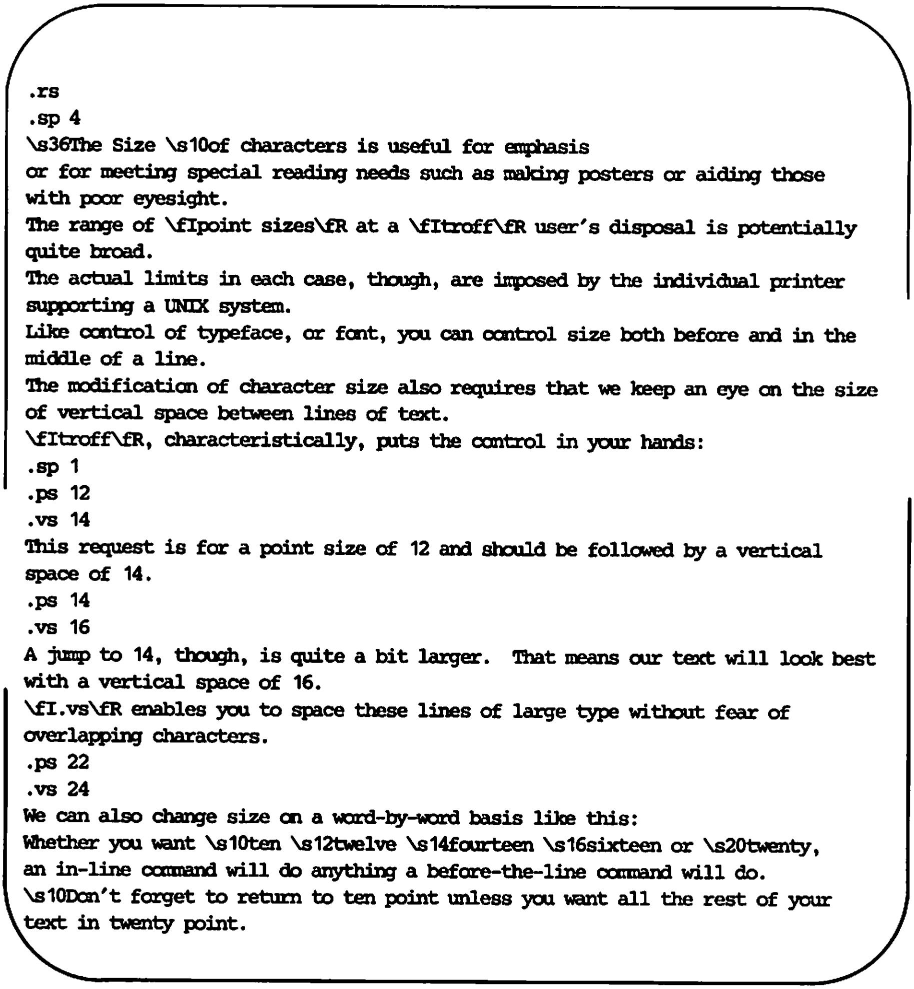
$$M = \frac{\sum_{i=1}^{n} x_i}{n}$$

The median divides ranked scores into halves. Given that the *median interval* is the score interval that contains the n/2nd largest score when scores are ordered by size, the formula for computing a median (Md) is

$$Md = \begin{Bmatrix} \text{Lower real} \\ \text{limit of} \\ \text{median interval} \end{Bmatrix} + \begin{Bmatrix} \text{width of} \\ \text{median} \\ \text{interval} \end{Bmatrix} \left[\frac{(n/2) - \begin{Bmatrix} \text{Cumulative} \\ \text{frequency up to} \\ \text{the median interval} \end{Bmatrix}}{\begin{matrix} \text{frequency in} \\ \text{median interval} \end{matrix}} \right]$$

The mode is the most frequently occurring score in a group of scores.

File: troff.sizes



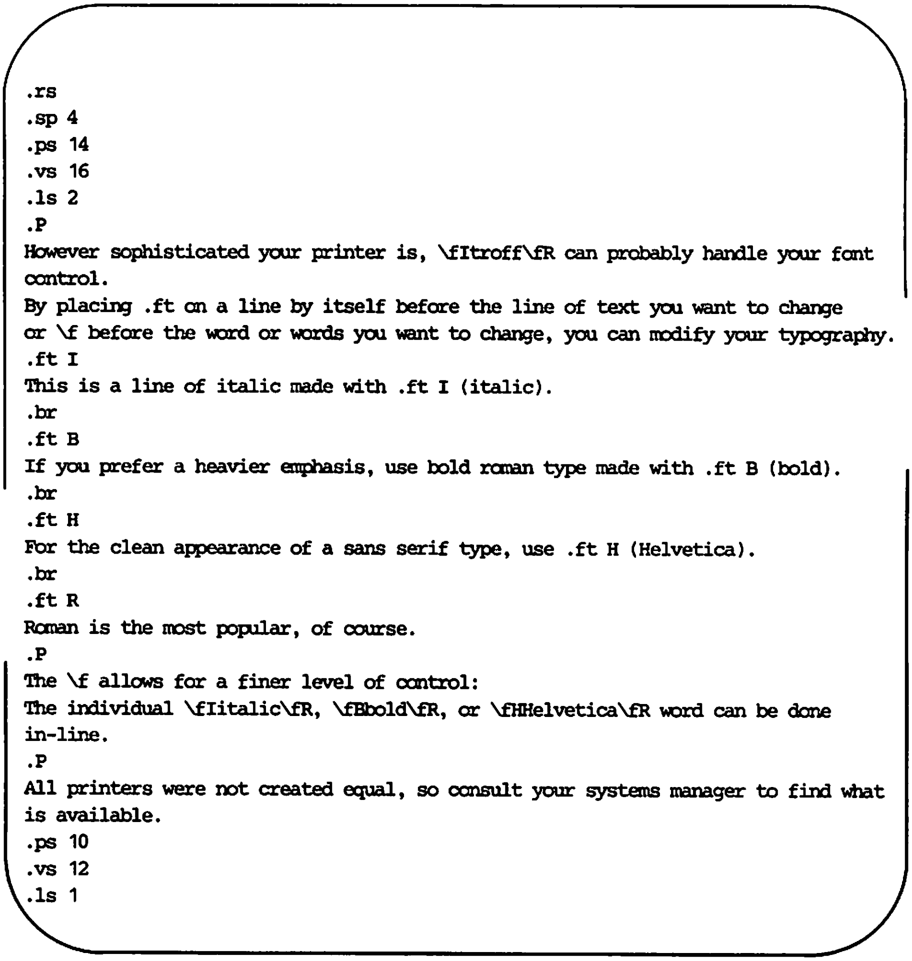
```
.rs
.sp 4
\s36The Size \s10of characters is useful for emphasis
or for meeting special reading needs such as making posters or aiding those
with poor eyesight.
The range of \fIpoint sizes\fR at a \fItroff\fR user's disposal is potentially
quite broad.
The actual limits in each case, though, are imposed by the individual printer
supporting a UNIX system.
Like control of typeface, or font, you can control size both before and in the
middle of a line.
The modification of character size also requires that we keep an eye on the size
of vertical space between lines of text.
\fItroff\fR, characteristically, puts the control in your hands:
.sp 1
.ps 12
.vs 14
This request is for a point size of 12 and should be followed by a vertical
space of 14.
.ps 14
.vs 16
A jump to 14, though, is quite a bit larger.  That means our text will look best
with a vertical space of 16.
\fI.vs\fR enables you to space these lines of large type without fear of
overlapping characters.
.ps 22
.vs 24
We can also change size on a word-by-word basis like this:
Whether you want \s10ten \s12twelve \s14fourteen \s16sixteen or \s20twenty,
an in-line command will do anything a before-the-line command will do.
\s10Don't forget to return to ten point unless you want all the rest of your
text in twenty point.
```

Command line: **troff troff.sizes** | *typesetter*

The Size of characters is useful for emphasis or for meeting special reading needs such as making posters or aiding those with poor eyesight. The range of *point sizes* at a *troff* user's disposal is potentially quite broad. The actual limits in each case, though, are imposed by the individual printer supporting a UNIX system. Like control of typeface, or font, you can control size both before and in the middle of a line. The modification of character size also requires that we keep an eye on the size of vertical space between lines of text. *troff*, characteristically, puts the control in your hands:

This request is for a point size of 12 and should be followed by a vertical space of 14. A jump to 14, though, is quite a bit larger. That means our text will look best with a vertical space of 16. *.vs* enables you to space these lines of large type without fear of overlapping characters. We can also change size on a word-by-word basis like this: Whether you want ten twelve fourteen sixteen or twenty, an in-line command will do anything a before-the-line command will do. Don't forget to return to ten point unless you want all the rest of your text in twenty point.

File: troff.fonts



```
.rs
.sp 4
.ps 14
.vs 16
.ls 2
.P
However sophisticated your printer is, \fItroff\fR can probably handle your font
control.
By placing .ft on a line by itself before the line of text you want to change
or \f before the word or words you want to change, you can modify your typography.
.ft I
This is a line of italic made with .ft I (italic).
.br
.ft B
If you prefer a heavier emphasis, use bold roman type made with .ft B (bold).
.br
.ft H
For the clean appearance of a sans serif type, use .ft H (Helvetica).
.br
.ft R
Roman is the most popular, of course.
.P
The \f allows for a finer level of control:
The individual \fIitalic\fR, \fBbold\fR, or \fHHelvetica\fR word can be done
in-line.
.P
All printers were not created equal, so consult your systems manager to find what
is available.
.ps 10
.vs 12
.ls 1
```

Command line: **troff −mm troff.fonts** | *typesetter*

However sophisticated your printer is, *troff* can probably handle your font control. By placing .ft on a line by itself before the line of text you want to change or \f before the word or words you want to change, you can modify your typography. *This is a line of italic made with .ft I (italic).*

If you prefer a heavier emphasis, use bold roman type made with .ft B (bold).

For the clean appearance of a sans serif type, use .ft H (Helvetica).

Roman is the most popular, of course.

The \f allows for a finer level of control: The individual *italic*, **bold**, or Helvetica word can be done in-line.

All printers were not created equal, so consult your systems manager to find what is available.

File: troff.ad

```
.rs
.sp 2
.ce 11
.ft B
.ps 24
Growing Computer Software Company Seeks
.sp 4
.fp 4 GS
.ft GS
experienced
.sp 2
\s28C Programmer
.sp 5
.fp 4 S
.ft B
\s24Knowledge of
.vs 30
.ps +1
\fHUNIX System V\fP
.ps -1
A Must
.vs
.sp 5
Opportunity For \fIRapid\fB Advancement
.sp 6
.ps 24
\v'-0.2i'Up\v'+0.2i'beat Working Environment
.sp 4
\s20Call 012-345-6789
.sp 2
or
.sp 2
Lo\h'-0.33m'ok In These Pages For Further Information
.ft R
.ps 10
```

Command line: **troff troff.ad** | *typesetter*

Growing Computer Software Company Seeks

experienced

C Programmer

Knowledge of

UNIX System V

A Must

Opportunity For *Rapid* Advancement

Upbeat Working Environment

Call 012-345-6789

or

L●k In These Pages For Further Information

File: troff.aeneid

```
.in 1.2i
.sp 3
.ps 36
.ce 1
}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\
\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\
\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\
\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\
\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\
\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{}\h'-0.5m'{
.sp 2
.ce 3
.I "\s24THE ARGUMENTES OF"
.sp 1
.I "\s22the thirteene bookes of Aeneidos,"
.sp 2
.B "\s11expressed in verse."
.sp 3
.ls 2
.nf
.ps 6
.nm 1 5
.ps 10
.br
.B
1. AENEAS, \f2in the \(first, to \f3Lyby \f2land arriueth well.\f3
2. \f2The fall of \f3Troy, \f2and wofull dole, \
y\v'-0.5'\h'-0.35m'\s6e\s0\h'0.35m'\v'0.5' second booke doth tell.\f3
3. \f2The thyrd of wandringes speakes, and father dead, and laid full low.\f3
4. \f2In fourth Queene \f3Dido \f2burnes, & \(flames of raginge loue doth show.\f3
5. \f2The fift declareth plaies, and how the \(fleete with fier was cought.\f3
6. \f2The sixt doth speake of ghosts, and howe deepe \f3Plutoes \f2reygne was sought.
7. \f2The seuenth booke, \f3Aeneas\f2 bringes vnto his fatall land.\f3
8. \f2The eight prepareth war, and power how foes for to withstand.\f3
9. \f2The ninth of battels telles, and yet the captaine is away.\f3
10. Aeneas \f2greeuous wrath \f3Mezentius, \f2in the tenth doth slay.\f3
11. \f2The eleuenth in vnequall \(fight \f3Camilla \f2castes to ground.\f3
12. \f2The twelfth with heauenly weapons giues to \f3Turnus \f2mortall wound.\f3
13. \f2The thirteenth weds A\h'-0.35'Eneas wife, and brings him to eternall life.
```

Command line: **troff −mm troff.aeneid** | *typesetter*

THE ARGUMENTES OF the thirteene bookes of Aeneidos,

expressed in verse.

1. **AENEAS,** *in the first, to* **Lyby** *land arriueth well.*

2. *The fall of* Troy, *and wofull dole, yͤ second booke doth tell.*

3. *The thyrd of wandringes speakes, and father dead, and laid full low.*

4. *In fourth Queene* **Dido** *burnes, & flames of raginge loue doth show.*

5. *The fift declareth plaies, and how the fleete with fier was cought.*

6. *The sixt doth speake of ghosts, and howe deepe* **Plutoes** *reygne was sought.*

7. *The seuenth booke,* **Aeneas** *bringes vnto his fatall land.*

8. *The eight prepareth war, and power how foes for to withstand.*

9. *The ninth of battels telles, and yet the captaine is away.*

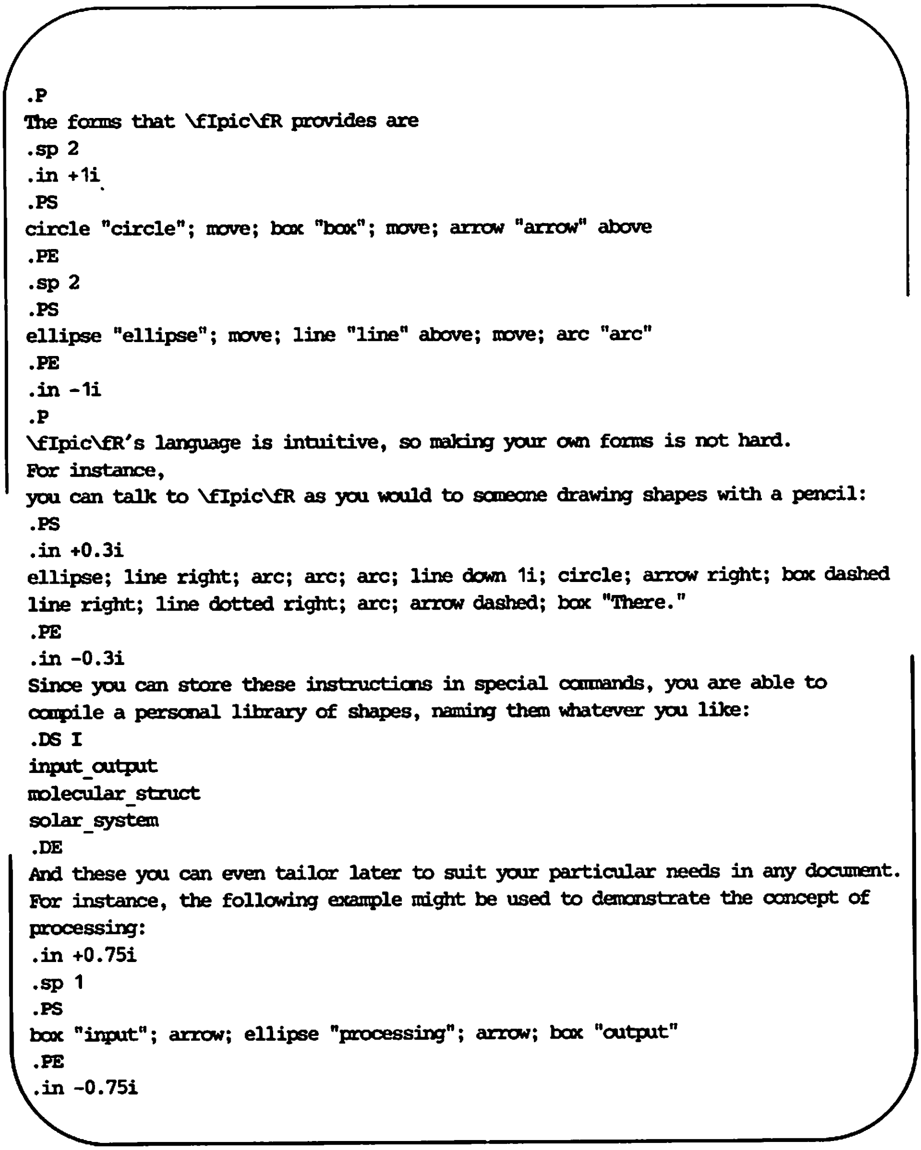
10. **Aeneas** *greeuous wrath* **Mezentius,** *in the tenth doth slay.*

11. *The eleuenth in vnequall fight* **Camilla** *castes to ground.*

12. *The twelfth with heauenly weapons giues to* **Turnus** *mortall wound.*

13. *The thirteenth weds Æneas wife, and brings him to eternall life.*

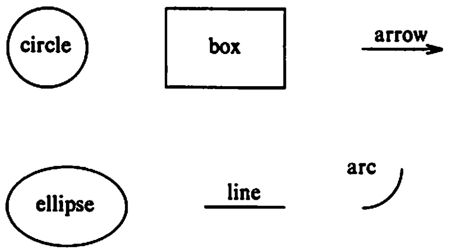
File: pic.forms



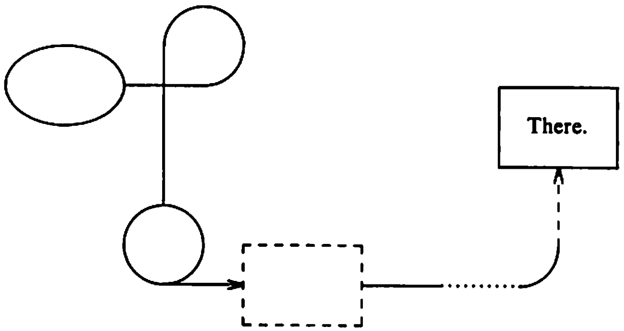
```
.P
The forms that \fIpic\fR provides are
.sp 2
.in +1i
.PS
circle "circle"; move; box "box"; move; arrow "arrow" above
.PE
.sp 2
.PS
ellipse "ellipse"; move; line "line" above; move; arc "arc"
.PE
.in -1i
.P
\fIpic\fR's language is intuitive, so making your own forms is not hard.
For instance,
you can talk to \fIpic\fR as you would to someone drawing shapes with a pencil:
.PS
.in +0.3i
ellipse; line right; arc; arc; arc; line down 1i; circle; arrow right; box dashed
line right; line dotted right; arc; arrow dashed; box "There."
.PE
.in -0.3i
Since you can store these instructions in special commands, you are able to
compile a personal library of shapes, naming them whatever you like:
.DS I
input_output
molecular_struct
solar_system
.DE
And these you can even tailor later to suit your particular needs in any document.
For instance, the following example might be used to demonstrate the concept of
processing:
.in +0.75i
.sp 1
.PS
box "input"; arrow; ellipse "processing"; arrow; box "output"
.PE
.in -0.75i
```

Command line: **pic pic.forms | troff −mm |** *typesetter*

The forms that *pic* provides are



pic's language is intuitive, so making your own forms is not hard. For instance, you can talk to *pic* as you would to someone drawing shapes with a pencil:



Since you can store these instructions in special commands, you are able to compile a personal library of shapes, naming them whatever you like:

input_output
molecular_struct
solar_system

And these you can even tailor later to suit your particular needs in any document. For instance, the following example might be used to demonstrate the concept of processing:

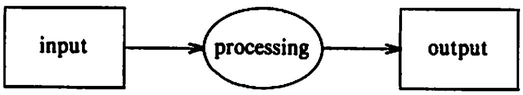


Table of Contents

Introduction

This tutorial shows you how to use **mm**, a collection of macros used to format letters, reports, memoranda, papers, manuals, and books.

You should be familiar with the following terms and tools to benefit fully from the pages ahead:

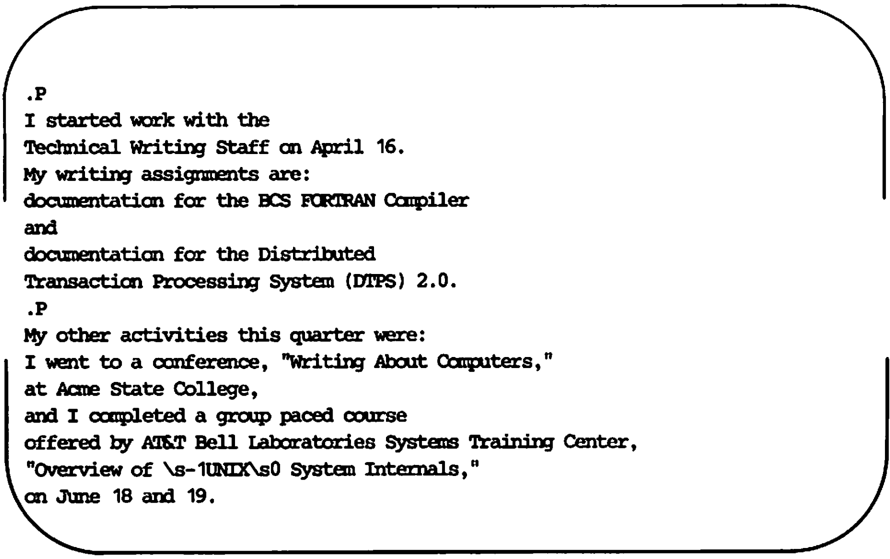
- You should know what a file and directory are and how to create them. See the *UNIX System V User Guide*.
- You should know how to use a UNIX system text editor (for example, **ed** or **vi**) to create and change your own documents. See the *UNIX System V User Guide*.
- You should know how to run programs with options and arguments. See the *UNIX System V User Guide*.

For a detailed description of the **mm** macros, refer to the "**mm** Technical Discussion." For more about the principles of text formatting, see the tutorial "The Formatter **nroff**."

You should use **mm** as you read through this tutorial, so that when you finish, you will be able to format documents with **mm** using its defaults. For some macros, you will be able to refine how they work with arguments. This tutorial provides several examples of lines before and after formatting; closely compare the input lines to the output lines to solidify your understanding of how the **mm** macros work.

Formatting the Body of Your Document

Suppose your file named **report.in** contains the following lines:



```
.P
I started work with the
Technical Writing Staff on April 16.
My writing assignments are:
documentation for the BCS FORTRAN Compiler
and
documentation for the Distributed
Transaction Processing System (DTPS) 2.0.
.P
My other activities this quarter were:
I went to a conference, "Writing About Computers,"
at Acme State College,
and I completed a group paced course
offered by AT&T Bell Laboratories Systems Training Center,
"Overview of \s-1UNIX\s0 System Internals,"
on June 18 and 19.
```

To format this file, type

mm -Tlp report.in > report.out

This command line formats **report.in** using the **mm** package and puts the result in a file named **report.out**. The option "**-Tlp**" prepares the result for a wide range of output devices. Ask your system administrator the device name of your local printer.

The **.P** appearing in **report.in** is an **mm** macro that creates a new paragraph. The dot in column one cues the formatter to the presence of a line that should be executed rather than printed, that is, a control line. The upper-case letter P that follows the dot specifies the control that you want the formatter to exert. After you use the formatter, **report.out** should look something like this:

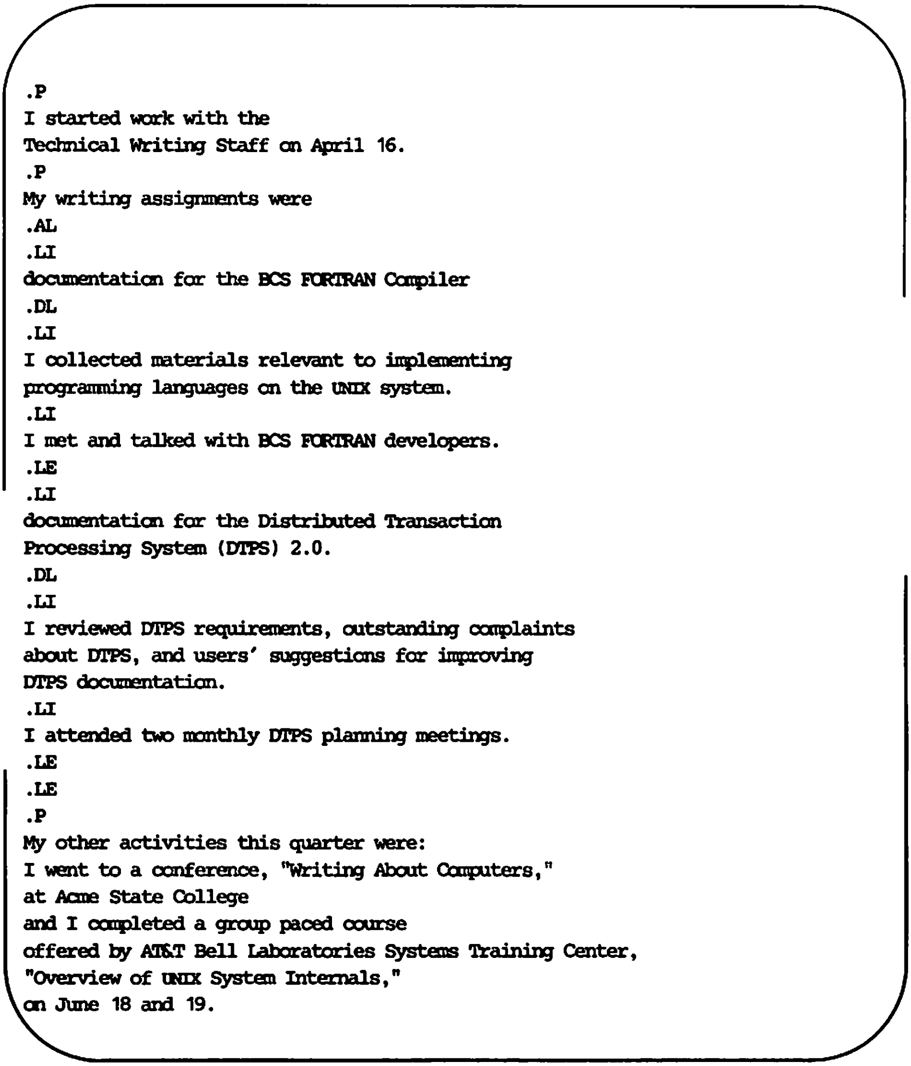
```
                              - 1 -


     I started work with the Technical Writing Staff on April 16.
     My writing assignments are: documentation for the BCS
     FORTRAN Compiler and documentation for the Distributed
     Transaction Processing System (DTPS) 2.0.

     My other activities this quarter were: I went to a
     conference, "Writing About Computers," at Acme State College
     and I completed a group paced course offered by AT&T Bell
     Laboratories Systems Training Center, "Overview of UNIX
     System Internals," on June 18 and 19.
```

Constraints that are imposed by your terminal or printer may put more or less text on a given line than is shown here. The important thing to notice is that **report.out** does not look the same as **report.in**: there's a page number at the top, and lines are filled out from the left-hand margin, continuing until the right-hand margin is reached.

Notice that **.P** left-justifies paragraphs by default. You can give **.P** the argument **1** (type **.P 1**) to indent the first line of a paragraph five spaces. Try **.P 1** to change the paragraph style in **report.out**.

Formatting Lists (.BL, .DL, .AL)

Suppose that you want to list the work done for your writing assignments. Change your file named **report.in** to look like this:

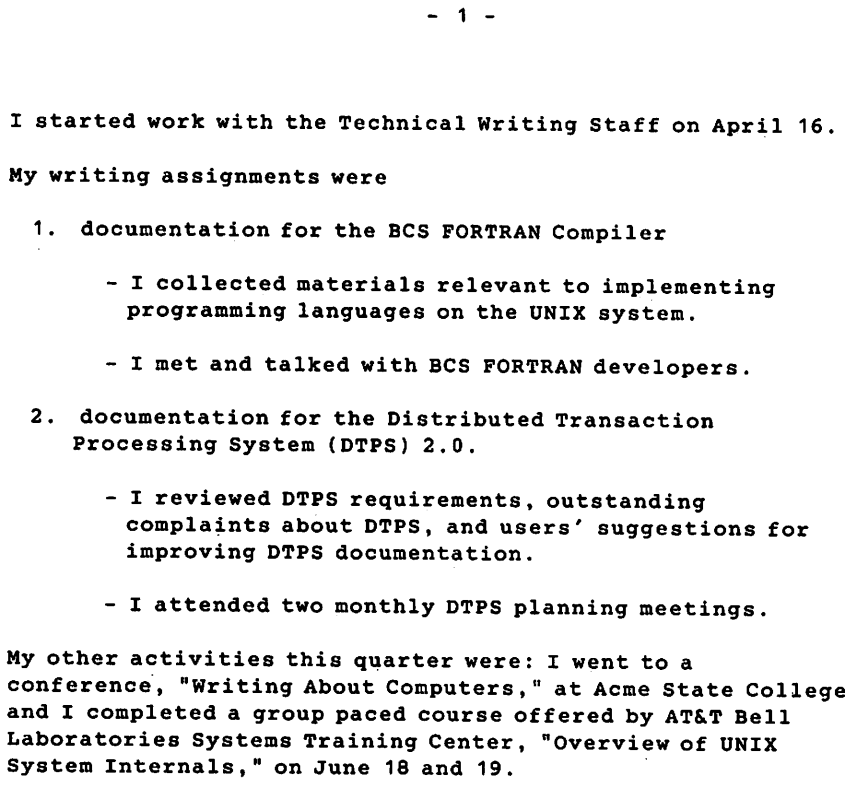


```
.P
I started work with the
Technical Writing Staff on April 16.
.P
My writing assignments were
.AL
.LI
documentation for the BCS FORTRAN Compiler
.DL
.LI
I collected materials relevant to implementing
programming languages on the UNIX system.
.LI
I met and talked with BCS FORTRAN developers.
.LE
.LI
documentation for the Distributed Transaction
Processing System (DTPS) 2.0.
.DL
.LI
I reviewed DTPS requirements, outstanding complaints
about DTPS, and users' suggestions for improving
DTPS documentation.
.LI
I attended two monthly DTPS planning meetings.
.LE
.LE
.P
My other activities this quarter were:
I went to a conference, "Writing About Computers,"
at Acme State College
and I completed a group paced course
offered by AT&T Bell Laboratories Systems Training Center,
"Overview of UNIX System Internals,"
on June 18 and 19.
```

Type

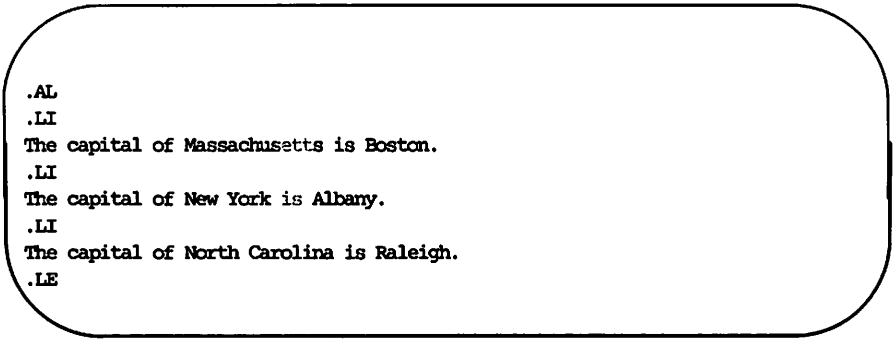
mm −Tlp report.in > report.out

to put the result of formatting the modified text into a file:



```
                              - 1 -


    I started work with the Technical Writing Staff on April 16.

    My writing assignments were

      1.  documentation for the BCS FORTRAN Compiler

            - I collected materials relevant to implementing
              programming languages on the UNIX system.

            - I met and talked with BCS FORTRAN developers.

      2.  documentation for the Distributed Transaction
          Processing System (DTPS) 2.0.

            - I reviewed DTPS requirements, outstanding
              complaints about DTPS, and users' suggestions for
              improving DTPS documentation.

            - I attended two monthly DTPS planning meetings.

    My other activities this quarter were: I went to a
    conference, "Writing About Computers," at Acme State College
    and I completed a group paced course offered by AT&T Bell
    Laboratories Systems Training Center, "Overview of UNIX
    System Internals," on June 18 and 19.
```

All **mm** lists share the same general structure: they begin with a list-initialization macro such as **.AL**; they specify items with the list-item macro, **.LI**; and they end with **.LE**, which is the list-end macro. Here's an example of an automatically incremented list:



```
.AL
.LI
The capital of Massachusetts is Boston.
.LI
The capital of New York is Albany.
.LI
The capital of North Carolina is Raleigh.
.LE
```

If you were to type these lines into a file called **my.file**, and then typed **mm my.file**, your processed file would look like this:

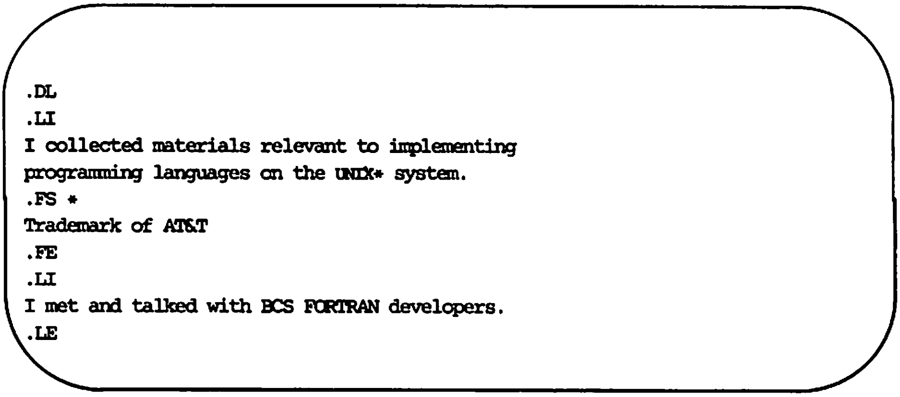
1. The capital of Massachusetts is Boston.
2. The capital of New York is Albany.
3. The capital of North Carolina is Raleigh.

The list-initialization macro that you choose usually determines the mark that appears before each list-item. For example, the **.AL** macro produces a numbered list by default.

The file **report.in** uses two types of lists: a dash list (starting with **.DL**) nested inside an automatically incremented list (**.AL**). Before the **.LE** associated with **.AL** occurs, **.DL** and associated **.LIs** appear twice. Whenever the formatter encounters a list-initialization macro, it puts everything (including other lists) on hold and attends to that list. When the **.LE** that ends the first dash list appears, the formatter picks up where it left off; notice that the list item between the dash lists becomes an automatically incremented item.

Formatting Footnotes (.FS, .FE)

In your report, you should acknowledge that UNIX is a trademark of AT&T. To do this, you can use a footnote.



```
.DL
.LI
I collected materials relevant to implementing
programming languages on the UNIX* system.
.FS *
Trademark of AT&T
.FE
.LI
I met and talked with BCS FORTRAN developers.
.LE
```

Two macros delimit the text of a footnote: **.FS** signals the beginning, and **.FE** signals the end. These two macros are called a macro pair, since you cannot use one without the other. Think of a macro pair as you think of parentheses; it is incorrect to use the open parenthesis without the close, and vice versa.

There are two ways to label a footnote in your document:

1. You can choose your own label by giving the **.FS** macro an argument. The label that you use in the document should be the label you use with **.FS**, to avoid confusion. In the example above, an asterisk is used as the footnote label, producing a footnote that looks like this:

```
----------
* Trademark of AT&T
```

2. Rather than use a label, you can number footnotes automatically with the characters *F.

```
.LI
I collected materials relevant to implementing
programming languages on the UNIX\*F system
.FS
Trademark of AT&T
.FE
```

This format produces a footnote that looks like this:

```
----------
1. Trademark of AT&T
```

if this is the first time you use *F in the document. If it's the second time, your footnote is

```
----------
2. Trademark of AT&T
```

and so on.

In the example above, an asterisk was used to label a footnote, but you can use any label or more than one (for example **.FS *****). Your document may contain both labeled and automatically numbered footnotes. If you use ***F** in the document, do *not* give the associated **.FS** macro a label, or you will get results that are hard to sort out.

NOTE

Labeled footnotes do not affect the incrementation of numbered footnotes.

Creating Numbered Headings (.H)

There are two categories of activities described in **report.in**: writing assignments and everything else. To emphasize these categories in **report.out**, use section headings. Use .H with an argument to create numbered section headings:

```
.H 1 "Writing Assignments"
.P
I started work with the Technical Writing Staff on April 16.
.BL
etc. etc.
.H 1 "Other Activities"
.BL
etc etc
```

The first argument to .H provides the numbered heading level and the second argument becomes the heading text. Enclose the heading text in double quotes if it contains spaces (for example "Second Level Heading"). For example

```
.H 1 First
.H 2 "Second Level Heading"
.H 2 "Another Second Level Heading"
.H 3 "Third Level Heading"
.H 3 "Another Third Level Heading (you can use up to seven levels)"
.H 1 "Another First Level Heading"
.H 2 "Second Level Heading"
```

produces headings like the following:

```
1.   First

1.1   Second Level Heading

1.2   Another Second Level Heading

1.2.1   Third Level Heading

1.2.2   Another Third Level Heading (you can use up to seven levels)


2.   Another First Level Heading

2.1   Second Level Heading
```

Creating Unnumbered Headings (.HU)

If you do not want to number your headings, you would use .HU:

```
.HU "Other Activities"
```

.HU acts the same as .H except that no heading mark is printed. When you use .H and .HU together, .HU increments the counter for level 2:

```
.H 1 First
.H 2 "Second Level Heading"
.HU "First Unnumbered Heading"
.HU "Second Unnumbered Heading"
.H 2 "Second Level Heading"
```

produces headings like these:

```
1.   First

1.1   Second Level Heading

First Unnumbered Heading

Second Unnumbered Heading

1.4   Second Level Heading
```

Displays (.DS, .DE, .DF)

mm gives you two ways to keep text blocks together: static displays and floating displays. Use the macro pair .DS and .DE to delimit static displays, which appear in the same relative position in your output as they do in your input. You can give .DS an argument to indent the whole display (by default approximately two spaces), as in the following example:

```
.DS I
FORTRAN --  a programming language
used for scientific applications,
and in academia for a variety of applications, including:
circuit analysis systems,
statistical packages,
and applications for engineers.
.DE
```

produces output like this:

```
    FORTRAN --  a programming language
    used for scientific applications,
    and in academia for a variety of applications, including:
    circuit analysis systems,
    statistical packages,
    and applications for engineers.
```

Notice that displays leave text exactly as you typed it, unlike the paragraph macro, which fills in the line from the left-hand margin to the right (unless you give .DS a second argument; see the "The Macro Package **mm**:

Technical Discussion" in the *Technical Discussion and Reference Manual*). If you use **.DS C** instead of **.DS I** in the example above, each line of text in the display will be centered:

```
                 FORTRAN --  a programming language
                 used for scientific applications,
         and in academia for a variety of applications, including:
                     circuit analysis systems,
                       statistical packages
                  and applications for engineers.
```

If you want to center the entire block of text, rather than each line, use **.DS CB**:

```
FORTRAN --  a programming language
used for scientific applications,
and in academia for a variety of applications, including:
circuit analysis systems,
statistical packages
and applications for engineers.
```

A Floating display, delimited by the **.DF/.DE** macro pair, "floats" through the input text to the top of the next page if there is not enough room for it on the current page; thus text that follows a floating display in the input file might precede it in the output file. The display text appears as you typed it, like the static display.

You cannot nest displays and footnotes, in any combination, and you cannot put section headings within displays or footnotes.

Creating Headers and Footers (.PH, .EH, .OH, .PF, .EF,.OF)

Another handy set of **mm** macros formats page headers and footers. All header and footer macros take an argument of the form:

"'left-part'center-part'right-part'"

This is a single argument, enclosed in double quotes and consisting of three parts, each part surrounded by delimiters (here, single quotes). In your output, these three parts are left-justified, centered, and right-justified. For example,

```
.PH "' W. Williams' Technical Writing Staff' Page \\\\nP' "
```

produces a page header like the following:

```
W. Williams                Technical Writing Staff                Page 1
```

at the top of the first page of the document. The page number changes appropriately for pages that follow.

You might wonder how **\\\\nP** in the input became **1** in the formatted file. **P** is a number register that contains the page number (do not confuse the number register **P** with the macro **.P**). Specifying **\\\\nP** in the argument to the page header macro tells **mm** to print the contents of register **P** in the page header. This is not how you usually access number registers, but that's beside the point for now. You can read about this number register in the **mm** technical discussion in the *Technical Discussion and Reference Manual.*

If need to use apostrophe (') within part of the header, use another character for the part delimiter:

```
"*Let's put this left*This center*Let's put this right*"
```

If you don't want to specify all three parts of a header, you don't have to. For example, leave out the left-part and center-part like this:

```
.PH "''' \\\\nP'"
```

If you don't specify a page header, the page number, enclosed by hyphens, appears in the center of your page.

The other header macros work the same way that .PH does: .EH prints a line at the top of each even-numbered page immediately after the page header, and .OH does the same for odd-numbered pages.

Footer macros work like header macros: use .PF for lines at the bottom of all pages, .EF for even-numbered pages, and .OF for odd-numbered pages. If you don't specify a page footer, you get a blank line. You can specify headers and footers anywhere in your file; they go into effect as soon as you use the appropriate macro.

Formatting the Beginning of a Formal Memorandum

You can show the title and the author's name at the beginning of **report.in**. This style, the formal memorandum, involves a special sequence of macros at the beginning of your input file:

```
.TL
Work Progress Report -- Second Quarter 1984
.AF "Business Computer Systems, Inc."
.AU "W. Williams" WW NY HDQT 1234 4-321 unix!ww
.MT 0
.P
I started work with the
Technical Writing Staff on April 16. . .
```

If you use any of the formal memorandum macros above, you must use them in the order shown to avoid a formatting error. Also, do not put any text or blank lines before .TL, or you will get a formatting error (parameter setting macros and requests are all right).

If at the beginning of your file you use these macros and then specify page headers, the header that you specify appears on the second and following pages, not the first. Page footers that you specify at the beginning of the file appear on the first and following pages.

The example above produces a mast for **report.out**:

```
                                AT&T Business Computer Systems, Inc.


    subject: Work Progress Report --       date: November 1, 1985
             Second Quarter 1984
                                       from: W. Williams
                                             NY HDQT
                                             4-321 x1234
                                             unix!ww


    I started work with the Technical Writing Staff on April 16. . .
```

Titles and Authors (.TL, .AU)

Anything after the title macro .TL appears beside the word **subject:** in the formatted report. If your paper has more than one author, use a separate .AU macro for each one, for example

```
.AU "W. Williams" WW NY HDQT 1234 4-321 unix!ww
.AU "J. Foley" JF XF 665415 6666 7-321 machine_6!jf
```

.AU is followed by the author's name (W. Williams), initials (WW), company location (NY), department (HDQT), telephone number (1234), office room number (4-321), and machine address for electronic mail (unix!ww). If you need to, you may specify the author's title with .AT, which immediately follows .AU for the given author. For example:

```
.AT Supervisor "Technical Writing Staff"
```

This title will appear in the signature block, which is discussed later.

The Name of Your Firm (.AF)

After your name (.AU and .AT), format the name of your firm with .AF. The default for .AF is "AT&T Bell Laboratories," which will appear when you do not use .AF. (You can change this default by asking your system administrator to edit the file **/usr/lib/macros/strings.mm**. See the "The Macro Package **mm**: Technical Discussion" in the *Technical Discussion and Reference Manual.*) This macro puts its argument in bold letters in the upper right-hand corner of the page.

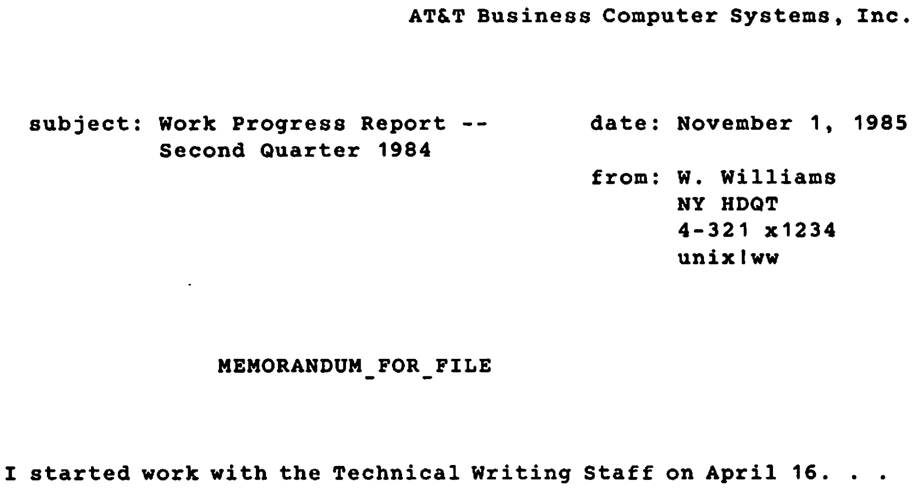
```
.AU "W. Williams" WW NY HDQT 1234 4-321 unix!ww
.AT Writer "Technical Writing Staff"
.AF "Business Computer Systems, Inc."
```

If you specify more than one author and more than one firm name, the last firm that you specify appears in the upper right-hand corner of the page.

If you do not give .AF an argument, you suppress printing the name of your firm and the labels **subject: date:** and **from:**, which are associated with formal memoranda. However, the information you provide .TL and the other formal memorandum macros will appear in their usual positions at the top of the page.

Choosing the Memorandum Style (.MT)

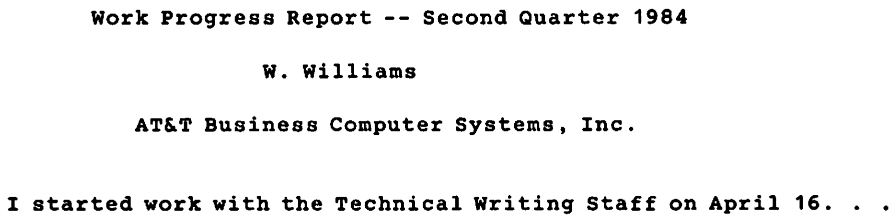
.MT specifies the formal memorandum style (versus the business letter style, described below). If you use the formal memorandum style, you may use an argument to specify one of three types: the memorandum, the released-paper, or the external letter. The mast above was produced with .MT 0, which corresponds to the memorandum type. If you do not provide an argument to .MT, or if you type .MT 1, you will produce a slightly altered memorandum type mast; *MEMORANDUM FOR FILE* will appear a few lines after the last line of information about the author (unix!ww).



```
                                    AT&T Business Computer Systems, Inc.


    subject: Work Progress Report --         date: November 1, 1985
             Second Quarter 1984
                                             from: W. Williams
                                                   NY HDQT
                                                   4-321 x1234
                                                   unix!ww


                    MEMORANDUM_FOR_FILE


  I started work with the Technical Writing Staff on April 16. . .
```

Thus by giving .MT different arguments, and changing nothing else, you make the same beginning macros (.TL, .AU, etc.) generate slightly different masts for the memorandum type. The **mm** technical discussion in the *Technical Discussion and Reference Manual* lists all the arguments available for .MT.

You get an entirely different mast, associated with the released paper type, by specifying .MT 4 instead of .MT 0. The released paper mast looks something like this:



```
             Work Progress Report -- Second Quarter 1984

                          W. Williams

               AT&T Business Computer Systems, Inc.


     I started work with the Technical Writing Staff on April 16. . .
```

You can get the mast associated with the external letter type by specifying .MT 5. The external letter mast looks something like this:

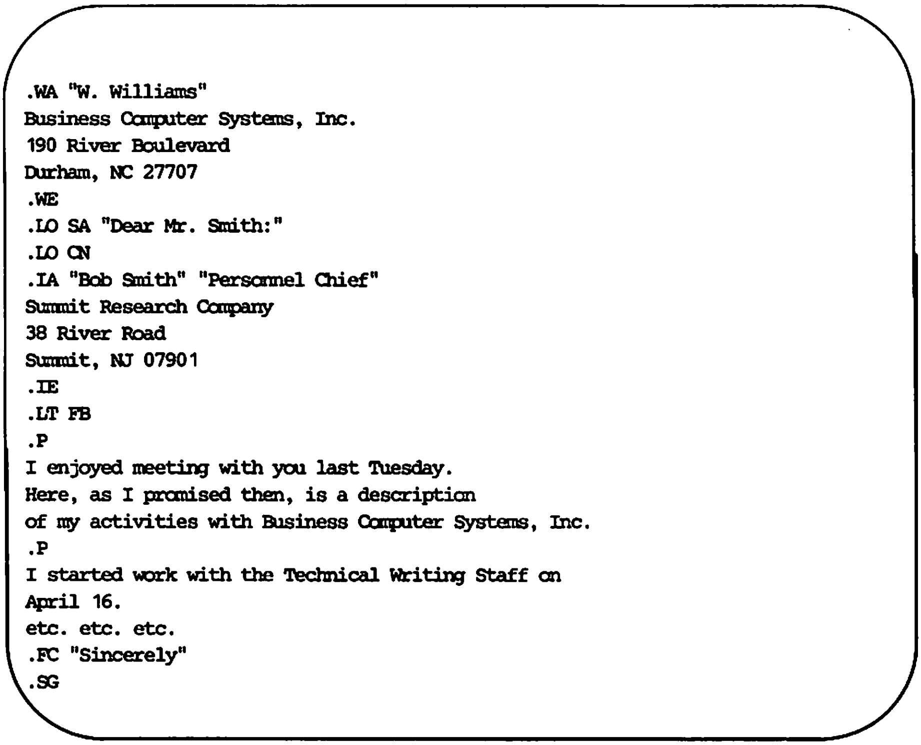
```
Work Progress Report --
Second Quarter 1984




                               November 1, 1985

I started work with the Technical Writing Staff on April 16. . .
```

The Sampler in this book shows memoranda laid out with formal memorandum macros, before and after formatting.

Formatting a Business Letter

In contrast to the formal memorandum style produced with .MT, you can obtain an entirely different style using a set of macros designed to produce common business letters.

Suppose that you wanted to include information from **report.in** in a full-blocked letter (the addresses, salutation, and so on, are left-justified). Instead of using the sequence .TL, .AU, .MT, use the following lines:



```
.WA "W. Williams"
Business Computer Systems, Inc.
190 River Boulevard
Durham, NC 27707
.WE
.LO SA "Dear Mr. Smith:"
.LO CN
.IA "Bob Smith" "Personnel Chief"
Summit Research Company
38 River Road
Summit, NJ 07901
.IE
.LT FB
.P
I enjoyed meeting with you last Tuesday.
Here, as I promised then, is a description
of my activities with Business Computer Systems, Inc.
.P
I started work with the Technical Writing Staff on
April 16.
etc. etc. etc.
.FC "Sincerely"
.SG
```

Letter Type (.LT)

mm offers four types of business letters; choose one by giving .LT (the letter type macro) an argument:

.LT FB	produces the full-blocked type (as above)
.LT SB	produces the semi-blocked type
.LT BL	produces the blocked type
.LT SP	produces the simplified type

The **mm** technical discussion in the *Technical Discussion and Reference Manual* explains these letter types.

Addresses (.WA, .WE, .IA, .IE)

The macro pair .WA and .WE formats the writer's address, and the pair .IA and .IE formats the "inside" or recipient's address. You must give an argument (the writer's name) to the macro .WA, but with .IA, this argument is optional. You also have the option of specifying a title (for example **Personnel Chief**) as a second argument to both macros.

Letter Options (.LO)

Use the letter-options macro .LO to format several common components of a business letter; the example above prints a salutation (.LO SA **"Dear Mr. Smith:"**) and the line "CONFIDENTIAL" (.LO CN). You can produce other components of a letter by giving other arguments to .LO. For example, .LO SJ prints a subject line.

```
SUBJECT:
```

If you specify the line .LO SJ **"Description of Activities"** for any business letter type besides the simplified type, the line prints the following on the second line below the salutation:

```
SUBJECT: Description of Activities
```

If you use the simplified type, this control line produces:

DESCRIPTION OF ACTIVITIES

You must use .WA and .WE, .IA and .IE, and .LT in that order, or you get a formatter error. Also, if you use any of these letter macros in the same file you that use formal memorandum macros (for example, .TL or .MT), you will get confusing results. The Sampler in this book shows a business letter using these macros, before and after formatting.

Formatting the End of Your Document

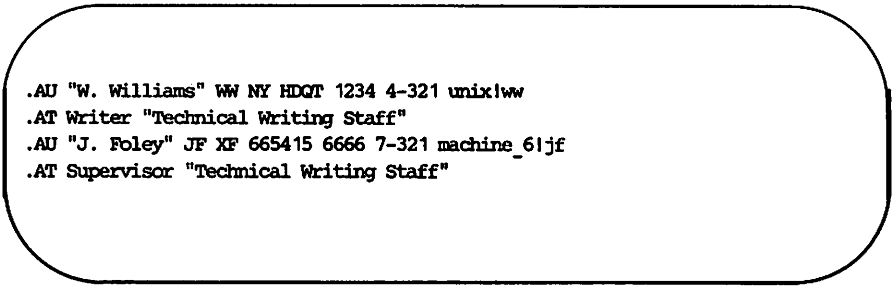
Most macros to format the end of a document work with both formal memoranda and with business letters.

Formal Closing (.FC) and Signature Block (.SG)

To close **report.in** with "Yours very truly," use .FC after the body of the document. If this closing seems too formal, specify an argument to .FC for a different closing:

```
.FC "Sincerely yours,"
.SG
```

.SG (for signature line) prints each author's name after the formal closing; otherwise each name appears a few spaces down from the last line of the body of the memo. The formatter collects the author's name from .AU (or .WA) for .SG to use. For example, the following lines were used to specify authors above:



```
.AU "W. Williams" WW NY HDQT 1234 4-321 unix!ww
.AT Writer "Technical Writing Staff"
.AU "J. Foley" JF XF 665415 6666 7-321 machine_6!jf
.AT Supervisor "Technical Writing Staff"
```

Now, if you use .FC **"Sincerely yours,"** and .SG at the end of your report (as above), the following signature block will appear centered in the output:

```
                              Sincerely yours,



                              W. Williams
                              Writer
                              Technical Writing Staff



                              J. Foley
                              Supervisor
                              Technical Writing Staff
```

Three blank lines are left above each name for an author's signature. You can use either .FC or .SG by itself.

Approval Line (.AV)

If your memorandum requires a line for a signature signifying formal approval, use .AV:

```
.AV "Todd Doe"
```

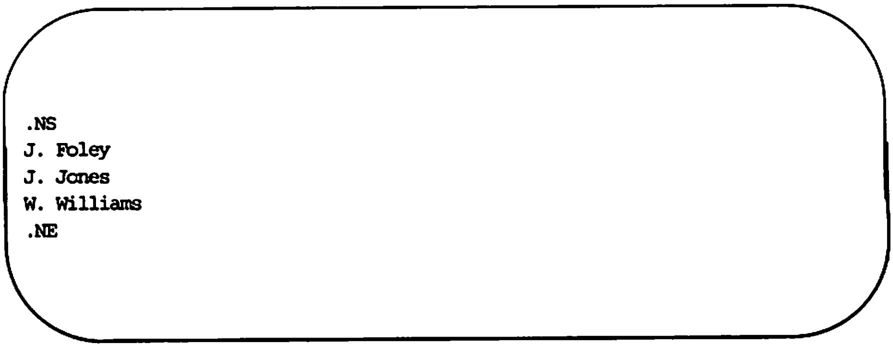
produces

```
APPROVED:


--------------------------------      ---------------
Todd Doe                              Date
```

You can use .AV anywhere on the page, but it is most commonly used between the signature block and the "Copy to" list.

Copy to Lists and Other Notations (.NS, .NE)

Use the macro pair .NS and .NE to print notations for lists of attachments or "Copy to" lists after the signature block. For example:



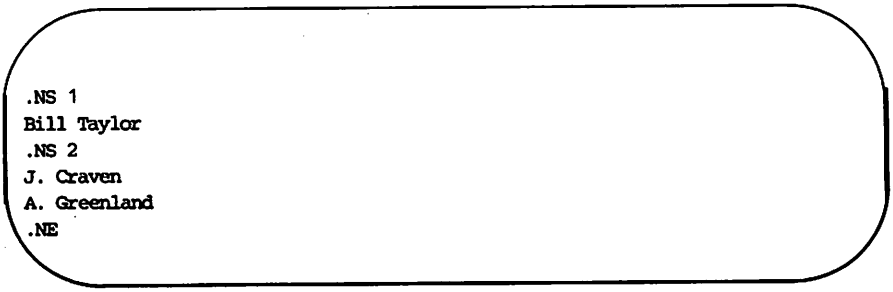
```
.NS
J. Foley
J. Jones
W. Williams
.NE
```

prints

```
Copy to
J. Foley
J. Jones
W. Williams
```

List the recipients of your document between .NS and .NE. This macro pair provides proper spacing and breaks notations properly across pages.

Use arguments to .NS to get more specific "Copy to" lists. For example:



```
.NS 1
Bill Taylor
.NS 2
J. Craven
A. Greenland
.NE
```

produces

```
Copy (with att.) to
Bill Taylor

Copy (without att.) to
J. Craven
A. Greenland
```

Some examples of memoranda and business letters formatted with **mm** appear in the Sampler at the beginning of this *User's Guide.*

Moving On

You now have a good grasp of the formatting power of **mm**. This tutorial was not intended to teach you everything about **mm**; it was meant to sketch out enough for you to start using **mm** right away. Read "The Macro Package **mm**: Technical Discussion" in the *Technical Discussion and Reference Manual* to learn the technical intricacies of **mm**.

Index

Table of Contents

Introduction

nroff is a text formatter for typewriter-like printers and terminals. A text formatter manipulates text by interpreting special commands that you place between the lines for which you want formatted output. **nroff** allows you to format a variety of documents, including letters, reports, and books. You will learn the basics of **nroff** by using this tutorial, following the examples that it provides.

The prerequisites to benefit from this tutorial are as follows:

- You should know what a file and directory are and how to create them. See the *UNIX System V User Guide.*
- You should know how to use a UNIX system text editor (for example, **ed** or **vi**). See the *UNIX System V User Guide.*
- Some experience with **mm** is helpful, but not necessary, in making the most of this tutorial. See the tutorial "The Macro Package **mm**: A Tutorial" in this book.

NOTE

"The Formatter **troff**: a Tutorial" in this book describes a related but more powerful set of requests that you can use to prepare text for phototypesetters. For details about using **nroff**, refer to the "**nroff** Technical Discussion" in the *DOCUMENTER'S WORKBENCH Software Technical Discussion and Reference Manual.*

After reading this tutorial, you will be able to control many attributes of your formatted document, including the margins, indentation, hyphenation, characters per line, and lines per page. You also will know how to use number registers, define strings, and create simple personal macros. This tutorial provides several examples of lines before and after formatting; closely compare the input lines to the output lines to solidify your understanding of **nroff**'s workings.

Requesting Space (.sp) and Indentation (.ti, .in)

Suppose you have a file named **file.in** that contains these lines:

```
I'm glad I'm not a pair of ragged claws, scuttling across the floor
of seafood restaurants.
This is the last line of this paragraph.
.sp 2
.ti +5m
This is the first line of a new paragraph.
This is the second line.
This is the third line.
This is the fourth line, and so on.
```

Besides text, **file.in** contains two requests: **.sp** and **.ti**. **.sp 2** requests two lines of space between text lines, and **.ti +5m** requests that the next text line be indented five ems (one em is about the width of the letter *m*). The dot (.) in column one alerts **nroff** to the presence of a line that should be executed rather than printed, that is, a control line. The two lower-case letters that follow this dot specify the control that you want **nroff** to exert. Arguments to requests, like **2** to **.sp** and **+5m** to **.ti**, refine how they work.

Requests are the simplest control lines that the DOCUMENTER'S WORKBENCH Software offers. Each request performs a single formatting task. Macros, in contrast, combine requests in special ways, and thus do several formatting chores. Later in this tutorial, you will learn how to create and use macros in addition to those that are already available with **mm**.

To format **file.in** and to put the results into a new file, type this command line:

```
nroff file.in > file.out
```

You can look at **file.out** on your terminal after the shell prompt returns, or you can send **file.out** to a printer with

```
cat file.out | lp
```

or send it directly with

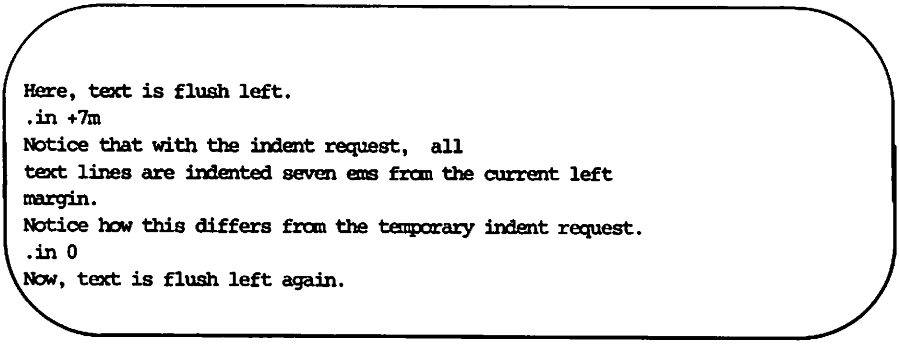
nroff file.in | lp

A printed version looks like this:

```
I'm glad I'm not a pair of ragged claws, scuttling across
the floor of seafood restaurants.  This is the last line of
this paragraph.


          This is the first line of a new paragraph.  This
is the second line.  This is the third line.  This is the
fourth line, and so on.
```

Your printer or terminal may insert more or less text on a given line than this example shows. The important thing to notice is that **file.out** does not look the same as **file.in**. Using **nroff**, you have the power to make **file.out** look precisely the way you want. You can command **nroff** to insert more (or fewer) blank lines or to indent more (or fewer) spaces by changing the arguments to these two requests. Try putting three spaces between the paragraphs, and indenting the new paragraph seven ems.

The **.ti** request (temporary indent) indents only the line that follows it; the second and following lines of text return to the left margin. This request is helpful when formatting paragraphs. To move all output lines to the right seven spaces, use **.in** instead of **.ti**; then the indent is more than temporary. The contents of **file.in** looks like this:



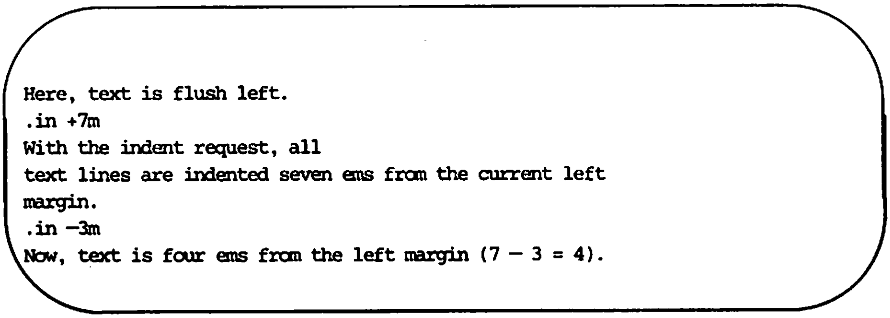
```
Here, text is flush left.
.in +7m
Notice that with the indent request,  all
text lines are indented seven ems from the current left
margin.
Notice how this differs from the temporary indent request.
.in 0
Now, text is flush left again.
```

A printed copy of **file.out** appears as follows:

```
Here, text is flush left.
               Notice that with the indent request, all text
               lines are indented seven ems from the current
               left margin.  Notice how this differs from the
               temporary indent request.
Now, text is flush left again.
```

Here, you indent lines in output seven ems until you set them flush left by typing **.in 0**. Typing **.in** without an argument sets indentation where it was before you last used **.in**. Initially, **nroff** sets indentation to 0 (flush left), so in the example above, **.in** would work the same as **.in 0**.

Instead of resetting the indentation to 0, you may want to set it to another value. Consider the contents of **file.in**:



```
Here, text is flush left.
.in +7m
With the indent request, all
text lines are indented seven ems from the current left
margin.
.in -3m
Now, text is four ems from the left margin (7 - 3 = 4).
```

A printed copy of **file.out** follows:

```
Here, text is flush left.
               With the indent request, all text lines are
               indented seven ems from the current left mar-
               gin.
        Now, text is four ems from the left margin
        (7 - 3 = 4).
```

Line Filling (.nf, .fi) and Line Breaks (.br)

Notice that the number of lines you type in is not the number **nroff** puts out. Besides interpreting control lines and making them disappear from the output, **nroff** rearranges your text, filling the page with tightly formatted output.

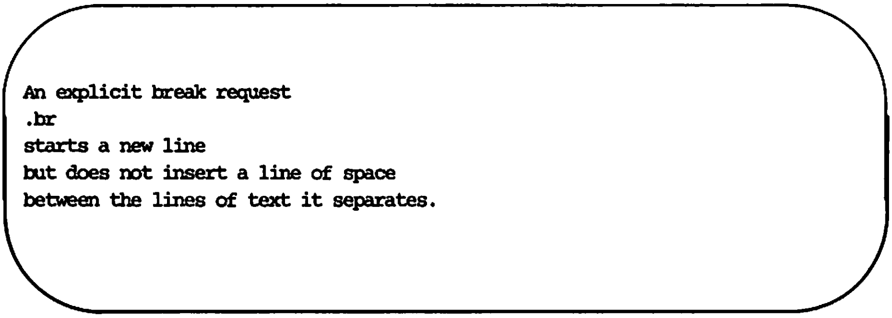
nroff fills lines automatically. When line filling is on, words accumulate in a line buffer until it is full, and then the buffer is flushed. **file.in** is as follows:

```
This means that words fill a line,
regardless
of their position on the page as they are input.
.nf
The "no-fill" request
turns off line filling,
making output the same as input.
.fi
Line filling stays off
unless you turn it back on with the fill request.
```

A printed copy of **file.out** appears as follows:

```
This means that words fill a line, regardless of their posi-
tion on the page as they are input.
The "no-fill" request,
turns off line filling,
making output the same as input.
Line filling stays off unless you turn it back on with the
fill request.
```

nroff also flushes the line buffer when it finds a line break. As you may have noticed, **.sp** forces a line break and produces lines of space. The **.br** also forces a new line. Consider this version of **file.in**:



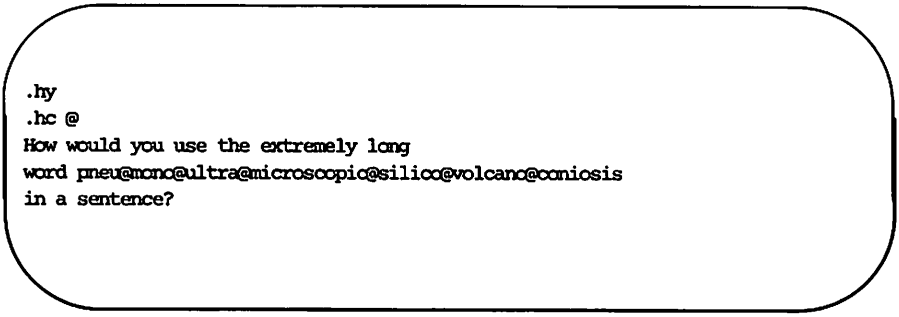
```
An explicit break request
.br
starts a new line
but does not insert a line of space
between the lines of text it separates.
```

file.out prints as follows:

```
An explicit break request
starts a new line but does not insert a line of space
between the lines of text it separates.
```

Hyphenation (.hy, .nh, .hc, .hw)

Notice that when **nroff** fills lines, it only puts out whole, never hyphenated, words. By default, **nroff** does not hyphenate. To turn on hyphenation anywhere in your text, use **.hy**. When you switch on hyphenation, you may put a hyphenation indicator in a text word to specify places where the word should be hyphenated if need be. Set this hyphenation indicator with **.hc**. **file.in** looks like this:



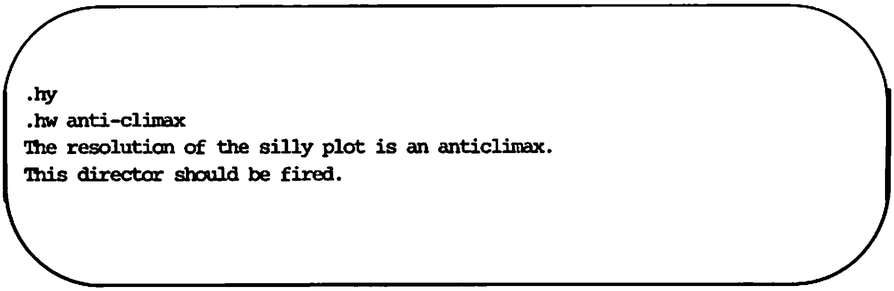
```
.hy
.hc @
How would you use the extremely long
word pneu@mono@ultra@microscopic@silico@volcano@coniosis
in a sentence?
```

The formatted, **file.out** follows:

```
How would you use the extremely long word pneumonoultra-
microscopicsilicovolcanoconiosis in a sentence?
```

From this point on, **nroff** interprets the character "@" as an acceptable place to put a hyphen, if needed. After inserting the hyphen, **nroff** flushes the line buffer and starts a new line. **nroff** hyphenates words not containing the hyphenation indicator wherever it wants. Do not use **.hc** without turning on hyphenation with **.hy**.

If you want to specify particular words to be hyphenated in a particular way, use the request **.hw**:



```
.hy
.hw anti-climax
The resolution of the silly plot is an anticlimax.
This director should be fired.
```

Now, every time **nroff** finds "anticlimax" at the end of a buffer, it tries to hyphenate it the way that you specified, not "an–ticlimax" or "anticli–max." If the word cannot fit on the line the way you have specified it, **nroff** does not try to hyphenate it.

Centering (.ce)

.ce centers as many text lines as its argument specifies. With no argument, .ce centers one line. Here's **file.in**:

```
.nf
The centering command is effectively a "no-fill" command,
except that output lines
are centered instead of flush left.
If you use the no-fill command with
.ce 4
the centering command, centering takes charge.
The next three lines and the line preceding are centered.
.fi
But now you have turned on line filling.
What happens to the centering?
The answer is that centering has priority over filling.
```

The printed version, **file.out**, looks like this:

```
The centering command is effectively a "no-fill" command,
except that the output lines
are centered instead of flush left.
If you use the no-fill command with
       the centering command, centering takes charge.
   The next 3 lines and the line preceding are centered.
         But now you have turned on line filling.
              What happens to the centering?
The answer is that centering has priority over filling.
```

Justification, Unpaddable Space (.ad, .na)

nroff ordinarily gives you even (justified) left and right margins. (**mm** gives you a ragged right margin by default.) To change margin justification, use **.ad**, as this version of **file.in** demonstrates:

```
.ad l
Here, you've given the adjustment request the argument "l".
This tells nroff to justify only the left margin.
Many people prefer a ragged right margin.
.ad b
With the "b" argument, .ad justifies both left and right margins.
When there is an even right and left margin,
nroff pads the line by expanding spaces.
This may produce text alignment unpleasing to the eye.
.na
The "no adjustment" request turns right justification off (that is, the
left margin is justified, but the right margin is not).
```

The printed versions appear as follows:

```
Here, you've given the adjustment request the argument "l".
This tells nroff to justify only the left margin.  Many peo-
ple prefer a ragged right margin.  With  the  "b"  argument,
.ad  justifies  both  left   and  right  margins.   When  there
is  an  even  right  and  left  margin,  nroff  pads  the  line  by
expanding   spaces.   This   may   produce   text      alignment
unpleasing  to the eye.  The "no adjustment" request turns
right justification off (that is, the left margin is justi-
fied, but the right margin is not).
```

One way to adjust the right margin and maintain a line pleasing to the eye is to specify a space that **nroff** cannot expand during justification. To do this, type a backslash followed by a space "\ " , an unpaddable space, at places **nroff** had padded. The backslash is an escape character; you will find out more about this below.

An alternative to using unpaddable spaces is to request that some seldom-used character, such as a tilde (~), be translated into a space on output. To do this, use the "translate" request

```
.tr ~ 
```

(that is, dot tr space tilde space). If you find that you need a tilde later in the output, turn it back into a tilde it by inserting this line:

```
.tr ~~
```

(dot tr space tilde tilde). Later, you may restore the tilde as an unpaddable space by repeating **.tr ~**, but only after a line break or after **nroff** outputs the line containing the tilde.

What do you suppose happens to the text below when you format it?

```
.ad b
.ce 2
What request wins out?
Will there be adjustment, or centering?
```

Remember what happened when **.fi** competed with **.ce**? Here as there, lines after **.ce** are centered, despite the request for adjustment. After that, left and right margins will be adjusted until you change adjustment or until you use **.na**.

Setting Tabs (.ta)

nroff automatically sets tab stops every eight ens from the current indent, but you can change these stops with **.ta**. Here's **file.in**:

```
.ta 0.5i 1.5i 2.5i 3.0i
The next line contains tabs; the tab request
places the tab stops at particular places:
         Here     is       a        line with tabs.
.ta 1.5i 2.5i 3.0i 3.5i
The next line also contains tabs, but the tab request places the
stops differently from above:
         Here     is       a        line with tabs.
```

The file comes off the printer looking like this:

```
The next line contains tabs; the tab request places the tab
stops at particular places:
     Here           is         a      line with tabs.
The next line also contains tabs, but the tab request places
the stops differently from above:
                Here         is     a      line with
tabs.
```

These tab stops are left-justified, but you can set up right-justified tab stops or centered tab stops, too. For details about how to do this, refer to the chapter "**nroff** Technical Discussion" in the *Technical Discussion and Reference Manual.*

If you want to position numbers, or if you need more complicated columnar layout, use **tbl**, which is described in the chapter "The Preprocessor **tbl**" in this guide.

Selecting a Font (.ft)

nroff is frequently used with mechanical printers like daisy-wheel printers, which produce documents by striking pre-cast characters as they turn on a wheel. Using such a printer, **nroff** can provide three distinctions among fonts. It can provide a regular font by default (**.ft R** or **\fR**); it can represent an italic font by having the printer underline (**.ft I** or **\fI**); finally, it can provide a bold version of the regular font by having the printer back up and overstrike characters (**.ft B** or **\fB**). **nroff** thus understands three fonts—regular, italics, and bold—even when it is used with a basic mechanical printer. When **nroff** is used with a more advanced printer, such as a laser printer or sophisticated dot matrix printer, it can provide a more pleasing version of regular (or roman), italics, and bold:

abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

To switch fonts, use the **.ft** request: **.ft B** for bold, **.ft I** for italic, and **.ft R** for roman. To return to the previous font, whatever it was, use either **.ft P** or **.ft** with no argument. Once you change fonts with **.ft** though, **nroff** uses the font that you specify until you change fonts again.

Another way to italicize text is to use the **.ul** request. Depending on your printer, **.ul** underlines the next input line or italicizes it. Follow **.ul** with an argument that requests the number of input lines to be italicized, for example **.ul 3**; otherwise, only the line that follows the request is italicized (much the same way that **.ti** indents only the line that follows it).

Fonts also can be changed within a line or word with the escape sequence **\f**. Consider the contents of this file:

```
\fBbold\fIface\fR text
```

A printed version looks like this:

bold*face* text

An escape sequence is a special in-line command that begins with the escape character \ (backslash). This character tells **nroff** that what comes next is special; thus **f** is interpreted as "font" instead of as the letter "f."

To avoid losing the last font requested after each in-line change, restore it with the escape sequence **\fP**, since **nroff** remembers only the last font called. For example, in the next line, the last **\fP** restores the font to whatever the font was before **\fB**:

```
        \fBbold\fP\fIface\fP\fR text\fP
```

In this next example, the **\fP** restores the font to italic:

```
        \fBbold\fIface\f1 text\fP
```

Margins (.po), Line Length (.ll)

If you are not content with the page dimensions that **nroff** gives you, you can change them. **file.in** looks like this:

```
You can change the left margin with .po, which stands for "page offset."
Here, nroff adds one inch to the existing left
margin to determine the new margin.
.po +1i
Here's another line of text.
.po
Once you change the margin, any indentation is relative to the new value.
To restore the previous left margin, type .po without an argument
```

A formatted version looks like this:

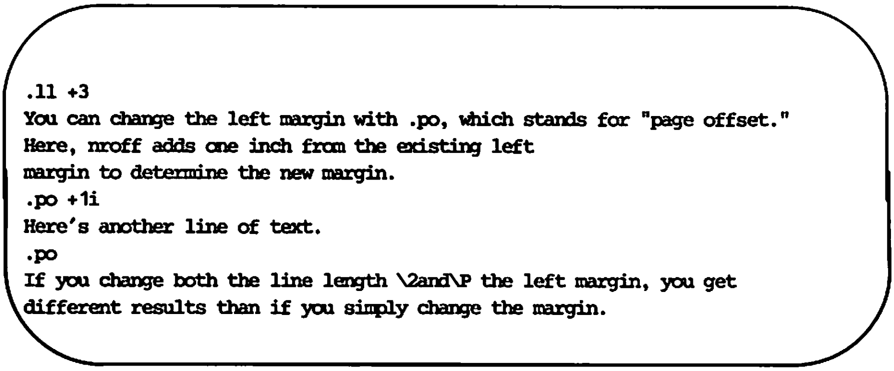
```
You can change the left margin with .po, which stands for "page
offset."  Here, nroff adds one inch to the existing left margin to
determine the new margin.
               Here's another line of text.  Once you change the
               margin, any indentation is relative to the new
               value.  To restore the previous margin, type .po
               without an argument.
```

Even though **.po** may appear to do the same thing that **.in** does, it doesn't. The formatting request **.in** indents from the current left margin, while **.po** changes the current left margin.

Look carefully at this last example. Notice that part of **file.in** before the second **.po** is not offset three spaces on output. Remember, **nroff** works with line buffers, not lines as you have typed them. After the second **.po**, the next buffer, not necessarily the next text line, that **nroff** flushes is the first to obey this request.

Also notice that **nroff** translates the escape sequence **\&** into a character that does not print. **\&** is useful when you want to treat a control line as text rather than as something to be executed. Putting this non-printable sequence in columns one and two, before the dot in a control line, nullifies the line's control (as the last example shows). Use **\&** consistently before any control sequence that you want to nullify, regardless of its current position on a line since when you edit text, the position of words or characters may change in unexpected ways.

The **.ll** request changes "line length." Here's **file.in**:



```
.ll +3
You can change the left margin with .po, which stands for "page offset."
Here, nroff adds one inch from the existing left
margin to determine the new margin.
.po +1i
Here's another line of text.
.po
If you change both the line length \2and\P the left margin, you get
different results than if you simply change the margin.
```

The formatted version looks like this:

```
Change the left margin with .po, which stands for "page offset."  Here,
nroff adds one inch from the existing left margin to determine the new
margin.
                 Here's another line of text.  If you change both the line
                 length and the left margin, you get different results than
                 if you simply change the margin.
```

As you can see, the line length has in fact extended the right margin. Thus, to decrease the right margin, you increase the line length with **.ll**.

Notice that some of the requests covered so far take alphabetic arguments and some take numeric arguments, preceded or not by a plus or a minus sign. To know what's appropriate for any given request, check the "**nroff** Technical Discussion" in the *Technical Discussion and Reference Manual.*

For now, consider the use of + and − before a number. These symbols change the previous setting by the amount you specify, rather than by just overriding it. The distinction is important: **.ll +3** makes lines three characters longer; **.ll 3** makes them three characters long.

Page Length (.pl), Page Breaks (.bp), and Page Numbers (.pn)

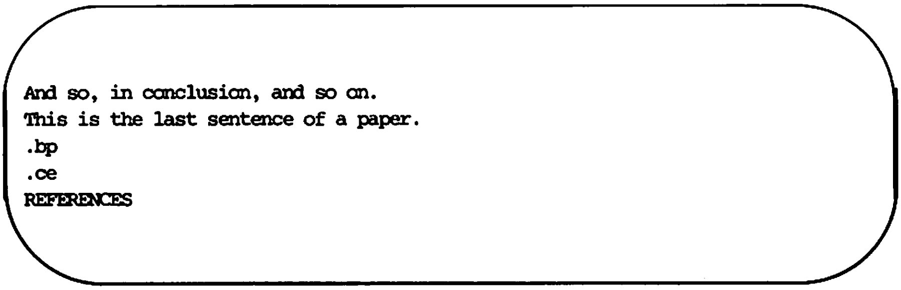
Now you have control of the width of your printed page, but what about the length? By default, **nroff** gives you a page 11 inches long. If you want to change the page length, that is, change the amount of space that **nroff** leaves between the text and the physical top and bottom of the page, use **.pl**.

.pl +1i

Here, **.pl** has an argument that consists of a number and a letter. This letter corresponds to a scale. If you do not specify **i** (which stands for inches) and simply type **.pl +1**, **nroff** assumes that you want to increase the present page length by one line space.

You also may specify units for **.ll**, **.po**, **.in**, and **.ti**. The default unit for **.in** and **.ti**, as for most horizontally oriented commands, is ems. (Remember, one em is roughly the size of the character "m".)

If you want to start a new page, use **.bp**, which stands for begin page. The input file looks like this:



```
And so, in conclusion, and so on.
This is the last sentence of a paper.
.bp
.ce
REFERENCES
```

.pn stands for "page number." The next page, when it occurs, will have the page number that you specify as the argument to this request. Pages that follow will also increment from this new page number.

Keeping Lines Together (.ne)

At times you will want to prevent certain lines from being split across pages. Use **.ne** to tell **nroff** the number of lines of text that you want kept together. Here's an input file using **.ne**:

```
.nf
.ne 4
My address is:
John Smith
1956 Malcolm Road
Hometown, USA
.fi
```

would print the four lines of text on the next page if there was not enough room for all of them on the current page.

Defining a String (.ds)

A string is a named collection of characters not including a newline character. Once you have defined a string with **.ds**, you can use the string name as shorthand for its contents. The following is **file.in**:

```
.ds sG string
Defining your own \*(sG is convenient when
you use a particular word
or sequence of characters many times.
To define a \*(sG, type .ds, then the string name, and then its
definition.
Note that \*(sG is replaced by its definition, the word "string,"
throughout this paragraph when you format it.
How you interpolate a \*(sG depends on
whether the \*(sG name is one or two characters long.
If the \*(sG is one letter long, type "\*"
and then the \(*sG name.
If the \*(sG is two letters long, type "\*("
and then the \*(sG name.
```

The processed file looks like this:

```
    Defining your own string is convenient when you use a par-
    ticular word or sequence of characters many times.  To
    define a string, type .ds, then the string name, and then
    its definition.  Note that string is replaced by its defini-
    tion, the word "string," throughout this paragraph when you
    format it.  How you interpolate a string depends on whether
    the string name is one or two characters long.  If the
    string is one letter long, type "\*" and then the string
    name.  If the string is two letters long, type "\*(" and
    then the string name.
```

Remember that a backslash tells **nroff** that what follows is special in some way. Escape sequences allow in-line control of formatting, such as the interpolation of strings. The backslash begins all escape sequences like *. The "**nroff** Technical Discussion" in the *Technical Discussion and Reference Manual* lists and describes all available escape sequences. Typing \e tells **nroff** to interpret \ as the character, backslash, not as the beginning of an escape sequence.

If you must begin a string with blanks, define it as follows:

```
.ds xx "    text
```

The double quote signals the beginning of a definition. There is no need for a trailing quote; the end of the line ends the string.

A string may be several lines long; if **nroff** encounters a \ at the end of any line of the string definition, the backslash is thrown away and the next line added to the current one. So you can create a long string simply by using the backslash like this:

```
.ds xx this \
is a very \
long string
```

Using Number Registers (.nr)

Number registers, like strings, can be useful in setting up a document that you can change easily later. **nroff** can do arithmetic with these number registers, which hold numeric values that control aspects of output style.

Like strings, number registers have one or two character names. They are set by the **.nr** command and can be used anywhere in your input by typing **\n** and then the name (for a one-character register name) or **\n(** and the name (for a two-character register name).

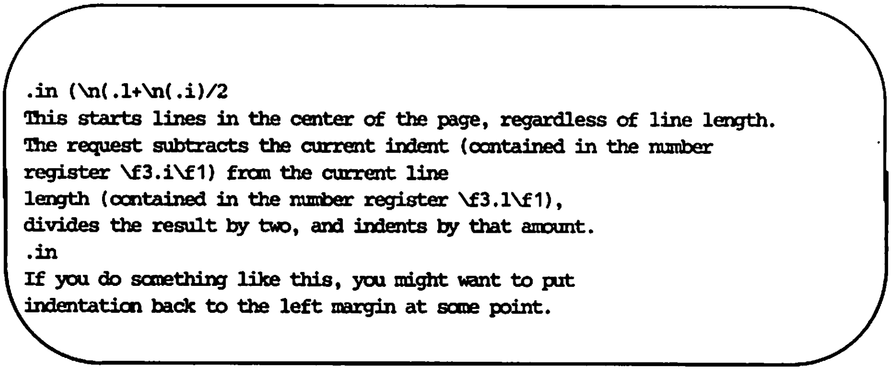
There are many predefined number registers maintained by **nroff**, among thèm **%** for the current page number; **dy**, **mo**, and **yr** for the current day, month, or year; and **.f** for the current font (which is a number from 1 to 3: 1 for roman, 2 for italic, and 3 for bold). Any of the predefined registers listed and described in the "**nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual* may be used in computations, but some, like **.f**, cannot be changed by **.nr**. The following example puts the page number and the current date in a page title (**.tl**).

```
     .tl 'John Smith' \n%' \n(mo-\n(dy-\n(yr'
```

Titles are easy; the whole argument to **.tl** appears as the next line of output. The first part of the argument (**John Smith**) appears in the left-hand corner of the page, the second part appears centered, and the last part appears right justified, like this:

```
John Smith                     21                     12-18-85
```

Here's another example using **nroff** number registers. **file.in** looks like this:



```
.in (\n(.l+\n(.i)/2
This starts lines in the center of the page, regardless of line length.
The request subtracts the current indent (contained in the number
register \f3.i\f1) from the current line
length (contained in the number register \f3.l\f1),
divides the result by two, and indents by that amount.
.in
If you do something like this, you might want to put
indentation back to the left margin at some point.
```

The formatted version looks like this:

```
                              This starts lines in the
                              center of the page, regardless
                              of line length.  The request
                              subtracts the current indent
                              (contained in the number
                              register .i) from the current
                              line length (contained in the
                              number register .l), divides
                              the result by two, and indents
                              by that amount.
       If you do something like this, you might want to put
       indentation back to the left margin at some point.
```

Creating a Simple Macro (.de)

A macro is a shorthand notation similar to a string; it names a collection of requests. When would you want to use a macro rather than a request?

Suppose you want to format every paragraph in a document differently, some with two spaces between them, some indented, some not. Here, it would be reasonable to use requests, since they provide that degree of flexibility. On the other hand, if you wanted to format paragraphs uniformly, you should use a macro. **mm** provides a collection of pre-defined macros that you can use to format several types of documents. However, if you do not want to use **mm** for some reason, you may write your own macros.

For example, in the Sampler file **nroff.letter**, two requests format every paragraph: one request puts a space between the paragraphs (**.sp**) and the other indents the first line five spaces (**.ti +5**). To create your own macro to do the job of these two requests, use the **.de** (for define) request.

You can call your paragraph formatting macro **.pD** (for paragraph definition). Here is how you use **.de** to create **.pD**:

```
.de pD
.sp
.ti +5
..
```

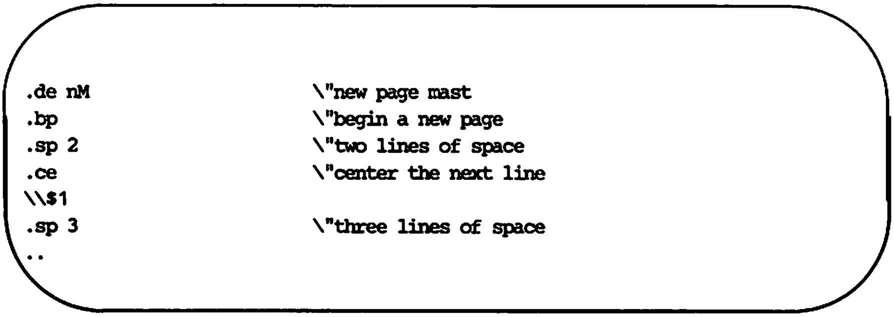
The control line .. closes the macro definition. You can define macros anywhere in your file that you wish, but it is better to keep all macro definitions at the beginning of your file, for easy maintenance.

After you define your macro, you can call it by name

.pD

and it does the tasks specified by the two requests that it incorporates.

Here is a macro that starts a new page and centers the macro's argument at the top of that particular page:

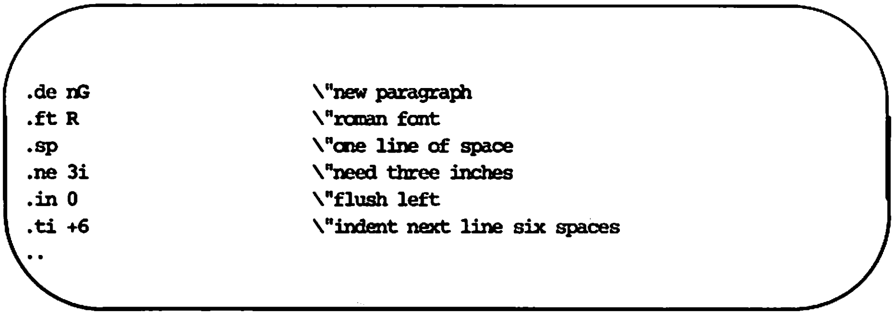


```
.de nM                      \"new page mast
.bp                         \"begin a new page
.sp 2                       \"two lines of space
.ce                         \"center the next line
\\$1
.sp 3                       \"three lines of space
..
```

Inside this macro definition, the string **\\$1** refers to the first argument that you give **.nM** (for example, **.nM REFERENCES**), placing it after **.ce**. Thus, the argument that you give **.nM** gets centered. This centered mast is placed two blank lines down from the top of the page, and then three blank lines are put out.

You may wonder what happens to the words **new page mast**, and so on, inside this macro. **nroff** throws anything after \" away, and then goes to the next line. **nroff** recognizes \" as the beginning of a comment. (Use spaces instead of tabs to set off comments.)

Here's a more elaborate paragraph macro.



```
.de nG                      \"new paragraph
.ft R                       \"roman font
.sp                         \"one line of space
.ne 3i                      \"need three inches
.in 0                       \"flush left
.ti +6                      \"indent next line six spaces
..
```

In this macro, **nroff** loads roman font, puts out a space, sees if there are three inches of space left on the page (if not, it skips to the next page), sets the indent to the left margin, and then indents the first line six spaces.

Why go to the trouble of setting the indent flush left and then indenting six spaces? Why not simply indent six spaces? You never know where your text has been. Earlier in your file, you may have made a request such as **.in +10**. The line **.in 0** resets indentation and puts the following text at the current left margin. Similarly, loading roman font is a way of ensuring that if you have forgotten to restore roman earlier, you have set things right with this new paragraph.

Notice that these macro names consist of a lower-case letter followed by an upper-case letter. It is good practice to stick to this pattern so that you do not accidentally redefine an **mm** macro.

Moving On

If you want to learn more about **nroff**, scan the material in the "**nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual* and find a request that you think would be useful. Read the material, and then experiment with the request. For example, take what you have learned from this tutorial and explore more complicated uses of number registers and strings.

The **troff** tutorial in this guide explains more complicated requests and macros you can define yourself. As mentioned before, **troff** is a more powerful set of requests that provide precise phototypesetting capabilities.

Index

Table of Contents

Introduction

This tutorial teaches the basic principles of **tbl**, a program that produces simple and complex tables. With **tbl**, you can align columns of numerical data, equations, or text. You can draw horizontal or vertical lines in the table and enclose any table or table element in a box.

The prerequisites to benefit from this tutorial are as follows:

- You should know what a file and directory are and how to create them. See the *UNIX System V User Guide.*
- You should know how to use a UNIX system text editor (such as **ed**, **vi**, or **ex**). See the *UNIX System V User Guide.*
- You should know how to execute commands and how to use options and pipes. See the *UNIX System V User Guide.*
- Your understanding of **tbl** would be assisted by a knowledge of the **mm** macro package and of **nroff**. See the tutorials in this guide, "The Macro Package **mm**" and "The Formatter **nroff**."

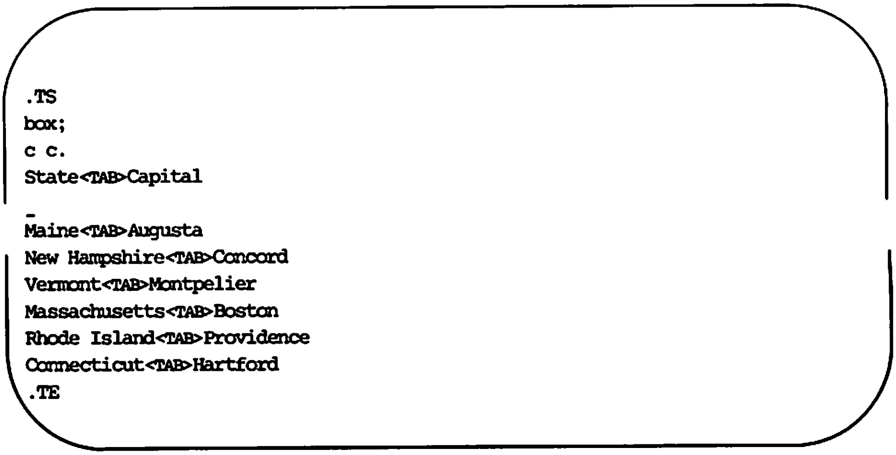
For a detailed description of **tbl**, see the "**tbl** Technical Discussion" in the *DOCUMENTER'S WORKBENCH Technical Discussion and Reference Manual.*

tbl is called a preprocessor because you use it to process a file before you use a formatter such as **nroff**. Like all preprocessors, **tbl** translates special words or characters into control lines that **nroff** or **troff** can use to produce the final formatted document. You may use **tbl** with other preprocessors, such as the equation formatting program **eqn** or the graphics formatting program **pic**, or with macro packages such as **mm**, without duplication of function, since **tbl** only processes lines between the delimiters that it recognizes.

After reading this tutorial, you will be able to prepare tables of varying degrees of complexity, and you will be able to read the "**tbl** Technical Discussion" in the *Technical Discussion and Reference Manual* to learn more about preparing tables.

Formatting a Simple Table

In the following discussion, <TAB> represents hitting the tab key once. If you have a file named **new_england** that contains these lines:



```
.TS
box;
c c.
State<TAB>Capital
_
Maine<TAB>Augusta
New Hampshire<TAB>Concord
Vermont<TAB>Montpelier
Massachusetts<TAB>Boston
Rhode Island<TAB>Providence
Connecticut<TAB>Hartford
.TE
```

typing either of these command lines

tbl −TX new_england | nroff −mm −Tlp | col > new_england.tbl
or
mm −t −Tlp new_england > new_england.tbl

will produce a file called **new_england.tbl**, which should look something like this:

State	Capital
Maine	Augusta
New Hampshire	Concord
Vermont	Montpelier
Massachusetts	Boston
Rhode Island	Providence
Connecticut	Hartford

The page number appears at the top of the page if you use the **mm** command line, but it does not show up if you use **nroff** without **mm**. Otherwise the results should be the same.

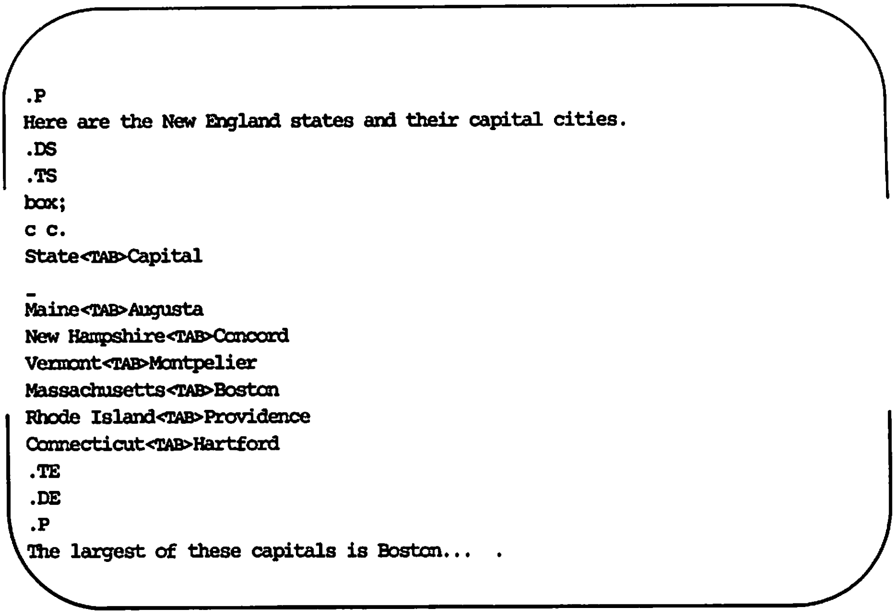
Do not worry about the **tbl** option **–TX**, which is useful when you produce a table on particular kinds of printers. The "**tbl** Technical Discussion" in the *Technical Discussion and Reference Manual* describes **–TX** in detail.

If you look at the output for this and later examples on a terminal screen or a typewriter-like printer, it might appear slightly different from this phototypeset page. You might see more space between the lines of text and the dark horizontal line that extends the width of the table (more about this line later), and you might notice a line of space after the last text line. These minor differences occur because terminals and printers produce **nroff** output, and this page is a product of **troff**, which handles spacing differently.

If **new_england.tbl** does not look right for reasons other than spacing, you might need to select another argument for the terminal (–T) option to **nroff –mm** or to **mm**. Ask your system administrator what to use, consulting the **mm**(1) or **nroff**(1) manual page in the *Technical Discussion and Reference Manual* for a list of the valid terminal arguments.

Table Delimiters

The macro pair .TS and .TE delimits the section of a file that **tbl** interprets. With **new_england** this means the entire file, but they can also delimit a part of a larger file, such as a letter or a report that you are formatting with **mm** macros. For example,



```
.P
Here are the New England states and their capital cities.
.DS
.TS
box;
c c.
State<TAB>Capital
_
Maine<TAB>Augusta
New Hampshire<TAB>Concord
Vermont<TAB>Montpelier
Massachusetts<TAB>Boston
Rhode Island<TAB>Providence
Connecticut<TAB>Hartford
.TE
.DE
.P
The largest of these capitals is Boston... .
```

If you include tables that are less than a page long in a document that you plan to format with **mm**, it is a good idea to put each table in a display, that is, to surround the table delimiters (.TS and .TE) with .DS (or .DF) and .DE so that a page break does not divide the table when it is printed. The "**mm** Technical Discussion" in the *Technical Discussion and Reference Manual* elaborates on producing tables in **mm** documents.

Global Options and the Format Section

After .TS you can specify global options that will affect the entire table. In **new_england**, the only global option is **box**, which tells **tbl** to enclose the whole table in a box. Suppose that you want the table to be printed in the center of the page as well as enclosed in a box. Change the option line to look like this:

```
box center;
```

The order of global options does not matter, so `center box;` will do the same thing.

A semicolon (;) ends the list of global options, and tells **tbl** that what follows is the format section, which specifies how each column in the table will look.

In the file **new_england**, the format section contains one line:

```
c c.
```

The format section consists of key letters that tell **tbl** how many columns there will be and how each column is to be formatted. Here, the two columns of **new_england** are centered, but if you want to left-justify the first column, you would change the format line to look like this:

```
l c.
```

Notice that in **new_england** there is only one format line for all seven lines of data or text. **tbl** applies the last (in **new_england**, the only) format line in the format section to all the remaining rows of a table. Notice that a period (.) ends the format section; more about this later. Below is an example that shows how you can control the format of each line in the table.

```
.TS
center box;
c c
c l
l c
l l.
State<TAB>Capital
_
Maine<TAB>Augusta
New Hampshire<TAB>Concord
Vermont<TAB>Montpelier
Massachusetts<TAB>Boston
Rhode Island<TAB>Providence
Connecticut<TAB>Hartford
.TE
```

The formatted output looks like this.

State	Capital
Maine	Augusta
New Hampshire	Concord
Vermont	Montpelier
Massachusetts	Boston
Rhode Island	Providence
Connecticut	Hartford

As you can see, **tbl** centers both columns of the first data line, `State<TAB>Capital`, centers the first column and left-justifies the second column of the second line, left-justifies the first column and centers the second column of the third line, and left-justifies both columns of the fourth and following lines.

Suppose that you wanted to change the typeface of column entries:

```
.TS
box;
cB cB
c lI
l cI
l lI.
State<TAB>Capital
_
Maine<TAB>Augusta
New Hampshire<TAB>Concord
Vermont<TAB>Montpelier
Massachusetts<TAB>Boston
Rhode Island<TAB>Providence
Connecticut<TAB>Hartford
.TE
```

Follow a key letter by **B** or **b** for bold, or **I** or **i** for italic to change the typeface of that column. Here is how the typeface changes above look:

State	**Capital**
Maine	*Augusta*
New Hampshire	*Concord*
Vermont	*Montpelier*
Massachusetts	*Boston*
Rhode Island	*Providence*
Connecticut	*Hartford*

A period after the last key letter signifies the end of the format section, and that text is next.

More About Options and Format

As the tables above illustrate, **tbl** looks for tabs to distinguish one column from the next. Keeping track of manually inserted tabs in small tables like **new_england** is not troublesome. You might have trouble, though, aligning columns in larger tables that contain tabs. The table of U.S. Presidents below shows that **tbl** allows you to substitute other characters for the tab.

```
.TS
box tab(;);
c s s s
c | c | c | c
l | l | l | l.
American Presidents
_
Name;Party;Term;Election-Opponents
_
Franklin D. Roosevelt;Democratic;1933-1945;1932-Hoover
;;;      Thomas
;;;_
;;;1936-Landon
;;;_
;;;1940-Willkie
;;;_
;;;1944-Dewey
_
Harry S. Truman;Democratic;1945-1953;1948-Dewey
;;;      Thurmond
;;;      Wallace
_
Dwight D. Eisenhower;Republican;1953-1961;1952-Stevenson
;;;_
;;;1956-Stevenson
.TE
```

Notice the options line. The option **tab(;)** tells **tbl** to translate any semicolon that it finds between the table delimiters into a tab character. Use any non-alphanumeric character as the tab character, for example,

```
tab(#);
```

Also notice a heretofore unexplained key letter in the format section, the letter s. This letter tells **tbl** to span the entry from the previous column across this column. Thus in the output shown below, **tbl** has centered the title **American Presidents** across the full table.

Notice that the format section contains bars separating key letters:

```
c ¦ c ¦ c ¦ c
l ¦ l ¦ l ¦ l.
```

When you put bars in a format line, you tell **tbl** to separate columns with vertical lines. If you put two bars between a pair of key letters, **tbl** separates those columns with a vertical double line, but only if your output device has the resolution to create it. Many printers that **nroff** supports do not have this resolution.

Notice too that some lines of text are separated by the underscore character:

```
;;;     Wallace

_
Dwight D. Eisenhower;Republican;1953-1961;1952-Stevenson
;;;_
```

Here, as in the **new_england** example, **tbl** translates an input line that contains only the underscore character (_) into a single line extending the full width of the table. When the underscore character is the only text in a column, as in the fourth column in the line below **Dwight D. Eisenhower**, **tbl** extends the line through only that column. Since there is no text between the semi-colons (tab characters) marking the first, second and third columns, those columns are empty. This shows how using a tab character instead of a tab makes table source files easier to read and change.

tbl does a similar chore when it sees the = character, creating requests for a horizontal double line the width of the table or the width of the column, as appropriate. As with the vertical double line, most printers that **nroff** supports do not have the resolution to create a horizontal double line. Unless you have access to a phototypesetter and to **troff**, you probably should not use this character or the vertical double line.

In this example, there are three manually inserted spaces between the tab character and some column entries (for example, *;;;* **Wallace**). This insertion is cosmetic, and entirely up to you.

Here is the table of U.S. Presidents after it has been formatted:

American Presidents			
Name	Party	Term	Election-Opponents
Franklin D. Roosevelt	Democratic	1933-1945	1932-Hoover Thomas
			1936-Landon
			1940-Willkie
			1944-Dewey
Harry S. Truman	Democratic	1945-1953	1948-Dewey Thurmond Wallace
Dwight D. Eisenhower	Republican	1953-1961	1952-Stevenson
			1956-Stevenson

In a **tbl** table, you must always include a format section and text; options, of course, are optional. To learn about the full complement of options and key letters available, read the "**tbl** Technical Discussion" in the *Technical Discussion and Reference Manual.*

Text Blocks

tbl makes each column slightly wider than the longest line of text that appears in the column. In the table of Presidents, the column containing Presidents' names was expanded to accommodate the name **Dwight D. Eisenhower**, the longest name in that column.

However, there may be times when you want to compress more than one input line into a single column of output. You could break up a text block into separate lines, as was done in the **Election-Opponents** column in the previous table. But **tbl** gives you the power to fill columns more densely with the text block delimiters T{ and T}. Suppose you wanted to add a biographical blurb to the table of Presidents.

```
.TS
box tab(;);
c s s s s
c | c | c | c | c
l | l | l | l | l.
American Presidents
_
Name;Party;Term;Election-Opponents;Notes
_
T{
Franklin
D. Roosevelt
T};Democratic;1933-1945;T{
1932-Hoover
    Thomas
.br
1936-Landon
.br
1940-Willkie
.br
1944-Dewey
T};T{
Inaugurated
the "New Deal"
and led the
nation during
World War II
T}
_
. . .
```

Do not put the text within the text block on the same line as the block-begin delimiter T{, and do not put any spaces or characters after this delimiter. The block-end delimiter T} must always appear first on its line. Additional columns of text may follow T} after you type a tab or tab character on the same line. If you do not specify the line length or the column width, **tbl** calculates the length of the text block as a function of the current line length, the number of columns in the table and the length of the longest line in the text block. Because this calculated length may be greater than some short lines in the text block, use the **.br** request (see "The Formatter **nroff**" in this guide) as specified here, to ensure that input lines do not overlap on output (for example, to ensure you do not get **1932-Hoover** and **Thomas** on the same line).

See the last section of this tutorial for a complete example of text blocks in a table.

Line Length and Column Width

You can change the column width of tables if you choose. Use the **nroff** request **.ll** to set the line length. Set column width with the **tbl** key letter **w** in the format section:

```
c s s s s
c | c | c | c | c
lw(1i) | l | l | l | lw(1i).
```

w sets a minimum column width; if the longest line in a column is wider than the value you specify after **w**, that longest line's width overrides the value. That is, **w** allows you to gain space around the text in a given column. In the format section specified above, the first column is at least one inch wide; the second, third, and fourth are at least as wide as the widest line of text; and the fifth is the same as the first. If you do not specify units (for example, inches), width defaults to ens. One en is about the width of the lower-case character n.

Troubleshooting

If you want to check a table before printing it (using the UNIX Operating System only), the command line

```
tbl filename >/dev/null
```

throws away the output, but prints any error messages that occur.

The program **checkmm** checks whether every **.TS** has a matching **.TE**.

If you forget .TS, or accidentally delete it, the formatter treats **tbl** control lines as if they were ordinary text, printing options, the format section, and text in one jumble. If you forget to put .TE at the end of a table, the formatter includes all input lines until the end of the file (or all lines it encounters until another instance of .TE is read) in the table. That the formatter is this forgiving may lead to strange output.

Remember, global options are optional, but the format section and table text are mandatory. If you want to format tables that are longer than one page on output, beware of long text blocks. Text blocks that span two pages behave unpredictably. If you require long text blocks, break them up into contiguous sub-blocks. For more details about multi-page tables, see the "**tbl** Technical Discussion" in the *Technical Discussion and Reference Manual.*

An Example: Including Text Blocks in a Table

To format the table that follows, type:

```
tbl -TX filename | nroff -mm -Tlp | col
 or
mm -t -Tlp filename
```

where *filename* contains the following input. You may direct output to a file, or to a terminal or printer.

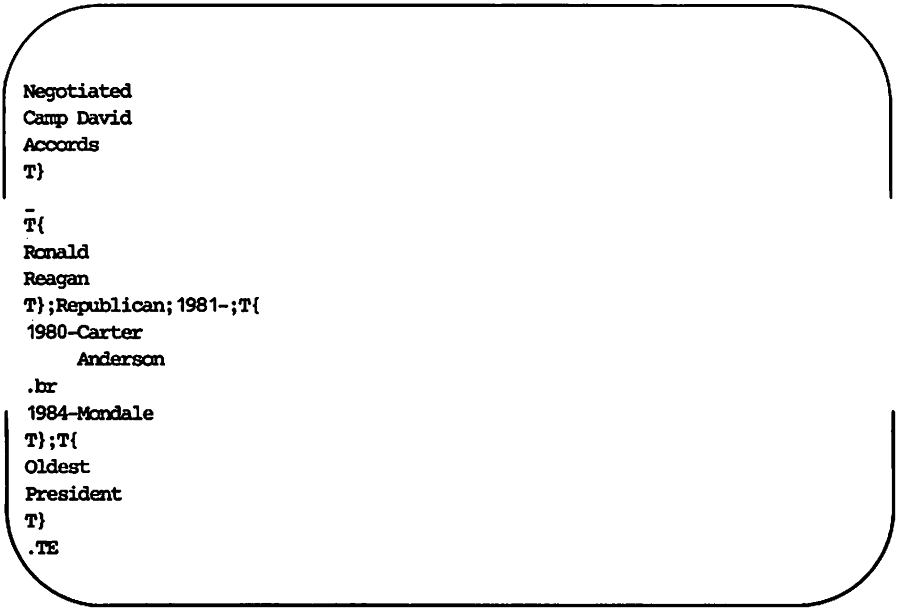
input:

```
.TS
box tab(;);
c s s s s
c | c | c | c | c
l | l | l | l | l.
Postwar American Presidents
_
Name;Party;Term;Election-Oppnts;Notes
_
T{
Harry
S. Truman
T};Democratic;1945-1953;T{
1948-Dewey
     Thurmond
     H. Wallace
T};T{
Led nation
during
Korean War
T}
_
T{
Dwight
D. Eisenhower
T};Republican;1953-1961;T{
1952-Stevenson
1956-Stevenson
T};T{
Presided over
economic boom
of the 1950s
T}
_
T{
John
F. Kennedy
T};Democratic;1961-1963;1960-Nixon;T{
Youngest man
```

input continued:

```
elected to the
Presidency
T}
_
T{
Lyndon
B. Johnson
T};Democratic;1963-1969;1964-Goldwater;T{
Architect of the
"Great Society"
T}
_
T{
Richard
M. Nixon
T};Republican;1969-1974;T{
 1968-Humphrey
     G. Wallace
.br
 1972-McGovern
T};T{
First President
to resign
T}
_
T{
Gerald
R. Ford
T};Republican;1974-1977;Never elected;T{
Became V.P. when
Agnew resigned
T}
_
T{
Jimmy
Carter
T};Democratic;1977-1981;1976-Ford;T{
```

input continued:



```
Negotiated
Camp David
Accords
T}
_
T{
Ronald
Reagan
T};Republican;1981-;T{
1980-Carter
     Anderson
.br
1984-Mondale
T};T{
Oldest
President
T}
.TE
```

output:

Postwar American Presidents				
Name	Party	Term	Election-Oppnts	Notes
Harry S. Truman	Democratic	1945-1953	1948-Dewey Thurmond H. Wallace	Led nation during Korean War
Dwight D. Eisenhower	Republican	1953-1961	1952-Stevenson 1956-Stevenson	Presided over economic boom of the 1950s
John F. Kennedy	Democratic	1961-1963	1960-Nixon	Youngest man elected to the Presidency
Lyndon B. Johnson	Democratic	1963-1969	1964-Goldwater	Architect of the "Great Society"
Richard M. Nixon	Republican	1969-1974	1968-Humphrey G. Wallace 1972-McGovern	First President to resign
Gerald R. Ford	Republican	1974-1977	Never elected	Became V.P. when Agnew resigned
Jimmy Carter	Democratic	1977-1981	1976-Ford	Negotiated Camp David Accords
Ronald Reagan	Republican	1981-	1980-Carter Anderson 1984-Mondale	Oldest President

Index

Table of Contents

Introduction

This tutorial is intended to give you a working knowledge of **troff**, the DOCUMENTER'S WORKBENCH Software formatter for typesetting text. It will, in addition, introduce you to **troff**'s programmable features.

You should be familiar with the following concepts and tools to benefit fully from the pages ahead:

- You should know how to use a UNIX system text editor (**ed**, **vi**, and **ex** are examples). See the *UNIX System V User Guide.*
- You should know what a file and directory are and know how to manipulate them. See the *UNIX System V User Guide.*
- You should be familiar with **nroff**, **troff**'s close relative. See the tutorial in this guide, "The Formatter **nroff**."
- Your understanding of **troff** would be assisted by a knowledge of the **mm** macro package though it is not essential. See the tutorial in this guide, "The Macro Package **mm**."

Basics

The control of text you have seen with **nroff** takes two things for granted: first, that the size and typeface of individual characters will not change and second, that the final printed output will appear in neat, parallel lines of evenly spaced text, reading from left to right.

nroff expects its output to follow the beaten paths of the typewriter carriage. While **nroff** output can be printed on more advanced devices such as phototypesetters or laser printers, it is primarily intended for simpler devices such as dot-matrix and daisy-wheel printers. **nroff** retains the sort of detailed control allowed by a typewriter: Tabs, margins, line spacing, and the rest can be set and reset. Yet, **nroff** offers programmable features as well.

nroff's near relative, **troff** (TEE-roff), also offers a detailed control of the text. While you still have almost all **nroff**'s capabilities, you also have in **troff** the power to shape the characters themselves. **troff** frees your text from the limits of mechanical printing and introduces it to digital typography. Characters can be enlarged or shrunk to a variety of sizes. Text can be changed into diverse shapes by choosing from a library of typefaces (such as Helvetica and Greek) and special characters (such as left and right hand signs and mathematical symbols).

With its requests and escape sequences, **troff** closely resembles **nroff**. The **troff** command line shown here is similar to command lines in **nroff**:

```
troff my.file | typesetter
```

Like **nroff**, **troff** is easy to use. But in its simple requests and escape sequences lie considerable power and capacity for fine-tuning. **troff**'s range includes all of the page, the ability to combine existing characters to make new ones, and the power to place those characters anywhere.

Fonts and Special Characters (.ft and \f)

Unlike **nroff**, which is primarily intended for producing typewriter-quality output, **troff** can present its output in a variety of typefaces, or fonts. The font request, **.ft**, accepts both alphabetical and numerical arguments with which to specify particular fonts. The actual range of fonts **troff** will produce depends on what printer you are using and the device programs (or drivers) supporting that printer.

You can change fonts with the request **.ft**, followed by an argument that names the font. Consider these examples of input and output. First the input:

```
.ft I
This line is italic.
.ft R
This line is roman.
```

And now the output:

> *This line is italic.*
> This line is roman.

The alphabetic argument—here *I* for italic and *R* for roman—mnemonically recalls the font it specifies. The argument **P** means previous font and will instruct **troff** to print all characters following **.ft P** in the font that preceded your last specification. The **.ft** request with no argument is identical to **.ft P**, telling **troff** to revert to the previous font.

The number of **troff** fonts is effectively increased by the request **.bd** (embolden). This request will further embolden any non-bold character and make any bold character even bolder. As with many **troff** requests, the fine-tuning is in your hands. **.bd** requires specifications of both font and width of boldness:

```
.bd F W
```

So, if your printer does not already provide emboldened italics but does provide italics, you can devise your own with **.bd**. Here is the input you would use:

```
.bd 2 3
.ft 2
These characters are printed in
italics that has been emboldened.
.bd 2
```

And here is the result:

These characters are printed in italics that have been emboldened.

You may wonder what the arguments following **.bd** mean. The **2** specifies the font position at which the italic font is loaded, position two. The second argument, **3**, specifies the width of the emboldening. The **.bd** request works by forcing the printer to back up a specified number of units from the place where a character was initially printed to plot it again at a slight offset. You determine that offset with the second argument, in this case, **3**. Any font, consequently, can be emboldened. Light fonts, such as the italic fonts, can be given greater weight and prominence, and bold fonts can be made even bolder. As the example shows, you turn off the emboldening request by typing **.bd** without the second, numerical argument.

To this point you have learned how to control fonts using requests. In fact, every possible font change can be made using requests. You may find, however, that requests are sometimes awkward. Making several font changes in a single line or phrase, for example, is inconvenient with a request. Therefore, **troff** permits you to choose fonts in a variety of ways. Making a font change with an in-line request, or escape sequence (so called because it contains the escape character) would be easier. An input line such as

```
\fIItalic runs. \fRRoman ambles. \fBBold plods.
```

produces

Italic runs. Roman ambles. **Bold plods.**

Notice that the font escape sequence, like other escape sequences, produces no output space. So if you want space, be sure to add it.

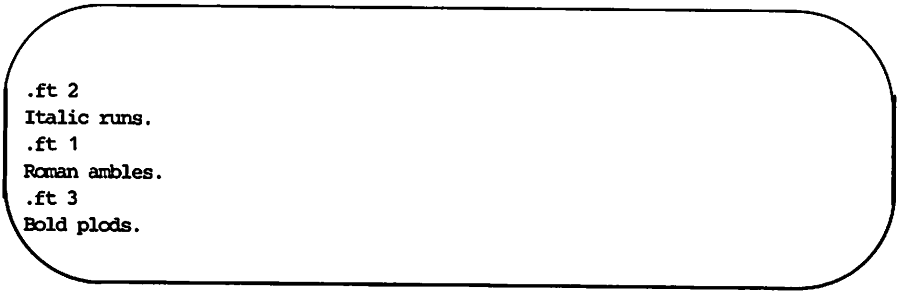
While not all printers offer the full catalog of fonts, all devices supporting **troff** provide at least three fonts: roman, italic, and roman bold. These are frequently mounted at the first three font positions:

1	roman
2	italic
3	roman bold

Using these numbers, you can pose a numerical argument to a font request or escape sequence instead of an alphabetical one. The example given earlier might have been typed as follows:

```
\f2Italic runs. \f1Roman ambles. \f3Bold plods.
```

Alternatively, it might have been specified with the **.ft** request:



```
.ft 2
Italic runs.
.ft 1
Roman ambles.
.ft 3
Bold plods.
```

What is the point of using this numerical method? It makes revision, especially on a large scale, easier. Suppose you decided to change all your italic words to bold and all your bold to italic after having completed a long report. You could change your input using a text editor. On the other hand, you could leave the input alone and simply load bold at position two and italic at position three using the **.fp** (font position) request:

```
.fp 2 B
.fp 3 I
```

Saving time and ensuring comprehensive precision, you've made the switch from bold to italic and from italic to bold.

With this ability to load fonts you can determine the default of a document's typefaces. If you had, for example, loaded italic at the first position, the document's predominant typeface would have been italic. The number of fonts at your disposal depends on your printing device.

Mounting Fonts (.fp)

troff can mount as many fonts as your printer will allow. That is, if your printer only permits four fonts to be present at any one time, **troff** will load four. But **troff** also makes the printer-imposed limit somewhat painless since it gives you the capacity for mounting more as needed. Of course, if the printer has only four fonts, **troff**'s flexibility in this respect is irrelevant. You can benefit from this feature using the request you learned in the last paragraph, the font position (**.fp**) request. Your selection of fonts may be quite large. The Autologic APS-5 phototypesetter at AT&T Bell Laboratories, for example, provides over thirty different fonts including the following:

R	Roman
I	*Italic*
B	**Bold**
S	Special (☛ ∫ ○)
BI	***Bold Italics***
C	News Gothic Condensed
CE	Century Expanded
CI	***Century Bold Italic***
`CW`	`Constant Width`
`CT`	`Courier Typewriter`
GR	Greek (α β γ)
𝔊𝔖	𝔊𝔢𝔯𝔪𝔞𝔫 𝔖𝔠𝔯𝔦𝔭𝔱
H	Helvetica
HB	**Helvetica Black**
HI	*Helvetica Italics*
PA	Palatino
PB	**Palatino Bold**
PI	*Palatino Italics*
PX	***Palatino Bold Italics***
S1	**Special 1**
𝒮𝒞	𝒮𝒸𝓇𝒾𝓅𝓉
SM	Stymie Medium
TB	**Techno Bold**

As you can see, many fonts can be used in even a single paragraph. You cannot, however, use them at random. **troff** is able instantly to apply whatever fonts your typesetter has, but many typesetters have a limited number of font positions. Your typesetter might only have four positions at which to mount fonts. In this case each must be mounted using **.fp** and its appropriate argument. If the list above had been printed on a typesetter limited to four font positions, you would have used the following method for stating your input:

```
...
.fp 1 BI
.fp 2 C
.fp 3 CE
.ft BI
        BI      Bold Italics
.ft C
        C       News Gothic Condensed
.ft CE
        CE      Century Expanded
.fp 1 CI
.fp 2 CW
.fp 3 CT
.ft CI
        CI      Century Bold Italic
.ft CW
        CW      Constant Width
.ft CT
        CT      Courier Typewriter
.fp 1 GP
.fp 2 GS
.ft GP
        GP      Greek (\(*a \(*b \(*g)
.ft GS
        GS      German Script
...
```

And so forth. As each new set of four fonts is about to be used, they are first mounted.

It is best not to remount another font where the Special (S) font had been in any single processing run of text. Ask your system administrator where the Special font is mounted.

Naturally, when you are finished using a relatively rare font, it is good practice to set things back to default:

```
.fp 1 R
.fp 2 I
.fp 3 B
```

As you can guess from what has been said, once each font is mounted you can invoke it by number as well as by letter(s). Courier Typewriter above, for example, could have been specified with **.ft 2** or **\f2** as well as with the request, **.ft CT**, or the escape sequence, **\f(CT**.

What happens when you ask for a font that has not been mounted? **troff** will automatically attempt to mount such a font at position 0. Because the zero position is reserved for this dynamic arrangement, you are not permitted manual access to it as you are to the other positions. Font position 0 cannot be used for more than one font in a given line or diversion.

The Font Macro

In addition to using requests and escape sequences, the **mm** macro package also provides macros to make font changes. These are limited to three: **.R** (roman), **.I** (italic), and **.B** (bold). (The remainder of available font macros are made up of combinations of these.) Each, in fact, is an indirect way to use the **.ft** request. **.R** activates **.ft 1**; **.I** activates **.ft 2**; etc. Each macro thus produces whatever font is loaded at its corresponding numerical position at the time it is used. If Helvetica had been mounted at position three, then **.B** would produce Helvetica, not emboldened roman.

Other font macros are **.RI**, **.RB**, **.BI**, **.BR**, **.IR**, and **.IB**. Each, as its name suggests, is a combination of the primary fonts: roman, italics, and bold. An intervening space between words or characters tells the macro when to switch fonts. Here is a typical input line:

```
.BI bold italic.
```

And now the output:

bold*italic.*

Notice that the space between the words has been closed by the macro. To leave a space or spaces, enclose the word(s) and space(s) in double quotes, for example:

```
.BI "bold " italic
```

These macros can be handy. You can use them conventionally, placing them on the line that precedes the text you want to change. The following is input:

```
.I
Like .ft I, the .I macro produces italic print.
.R
```

And now, the output:

Like .ft I, the .I macro produces italic print.

Like the **.ft I** request, you need to specify the change back to roman.

These macros can also be used on the same line as the text. This usage does not require a return to the previous font. By placing them on the same line as the text you want to modify, you set up an automatic return by default. The input looks like this:

```
.I "Used this way, these macros change font for one line only."
.br
The next line automatically reverts to the default font, roman.
```

The outputshows that the shift to italic occurs only in the line on which you place the macro:

Used this way, these macros change font for one line only.
The next line automatically reverts to the default font, roman.

Note that the double quotes given on the input line do not appear on the output line. This is a convention of **nroff** and **troff** macros when they appear on the text line. If the macro is followed by a single word (a sequence of characters not interrupted by white space), no double quotes are necessary. If the macro is followed by more than one word, it is necessary to enclose the phrase with double quotes. Thus, these **mm** macros are limited to one input line of contents, though the line could be extended by wrapping it instead of pressing RETURN.

Point Size (.ps and \s)

The many fonts **troff** is capable of loading usually come in a variety of sizes, which can be selected with the requests and escape sequences that control point size. You control point size with the request **.ps**, or with the escape sequence **\s**, both of which adjust the height and width of your characters. (The point size refers to the size of the block on which the character is plotted, so individual characters are usually smaller than the measure specified.)

Like the number of fonts available from typesetter to typesetter, the range of **troff**'s point sizes is device-dependent. The span of available sizes found on most devices ranges from six point (one-twelfth of an inch) to thirty-six point (one half an inch). But you can usually expect more. There are fifteen sizes in the following list:

6 point
7 point
8 point
9 point
10 point
11 point
12 point
14 point
16 point
18 point
20 point
22 point
24 point
28 point
36 point

As you can see, in the instances where a point size is skipped, you will usually get the nearest available point size. The following, for example, is a list of illegal sizes and their assigned replacements from a machine whose characters range from three point to forty-eight point:

1-2	→	3
21	→	20
23	→	22
25	→	24
27	→	26
29	→	28
31	→	30
33	→	32
35	→	34
37-38	→	36
39	→	40
41-42	→	40
43	→	44
45-46	→	44
47	→	48
49-	→	48

This reassignment of point size is typical of the UNIX system's behavior. Rather than quitting when it sees a command that is impossible to fulfill, **troff** gives you a reasonably close substitute. In most cases, as the present example illustrates, you get one that is one seventy-second of an inch smaller.

The **.ps** request accepts only numerical arguments. You would set text in twelve point characters as follows:

```
.ps 12
Twelve point is surprisingly
larger than default point size.
.ps 10
```

And here is the output:

Twelve point is surprisingly larger than default point size.

Notice the symmetry. Without an accompanying request specifying a return to the default point size (usually ten point), all succeeding text would have been set to twelve point.

You can also specify point size with the escape sequence **\s**. Like the font escape sequence, the size escape sequence is immediately followed by an argument. The arguments themselves are identical to the ones you would use with requests. That is, the numerical argument is identical to the point size you want:

```
\s10TEN\s15FIFTEEN\s20TWENTY\s10
```

gives you

TENFIFTEENTWENTY

Again, notice the return to default: ten point. A succession of twenty point characters would catch your attention but would require a good deal of page turning. You will also notice that the escape sequence itself occupies no space on the output line.

There are circumstances in which you would not want to specify point size literally. A given document, for example, might be destined to be printed using two different ranges of point sizes. Let's say one was to be printed using twelve point for readers with poor eyesight while a second printing was to be in nine point. To preserve proportion with each document, you might have to face a complicated editing effort. But if the point size changes you had made within each document had been relative changes, printing one document in two different point sizes would be easy. You accomplish relative changes using expressions instead of integers for arguments:

```
DEFAULT \s+4DEFAULT + 4 \s+4DEFAULT + 8\s-8
```

gives you

DEFAULT DEFAULT + 4 DEFAULT + 8

The point size escape sequence, **\s0**, means "return to the previous point size" and allows you to maintain a relative set of point size requests. Thus,

```
TEN \s+9NINETEEN \s0TEN
```

would produce

TEN NINETEEN TEN

Vertical Spacing (.vs and .ls)

troff characteristically offers a great capacity for fine-tuning. It was expressly designed to give you precise control over your text, and point size is no exception. As individual characters are subject to your control, so the spacing between lines of characters is available for your adjustment.

This companion parameter to point size is **.vs**, the vertical spacing request. Its numerical argument specifies the vertical spacing, or leading, between pairs of successive lines. Usually, you set vertical spacing to be about twenty percent larger than the character size. "Nine on eleven" (nine point characters and eleven point spacing) or "ten on twelve" (**troff**'s default value) is conventional.

That is,

```
.ps 9
.vs 11p
```

would give us text
that looked like this.
But if we were to change to

```
.ps 12
.vs 14
```

the result would be markedly different.
Point size and vertical spacing
change substantially the amount
of text per square inch.

```
.ps 6
.vs 7
```

would really drive the point home, though.
For example, ten on twelve uses about twice
as much space as seven on eight. This is six
on seven, which is even smaller.

It certainly
gives you lots of information per page,
but decreasing point size

```
.ps 5
.vs 6
```

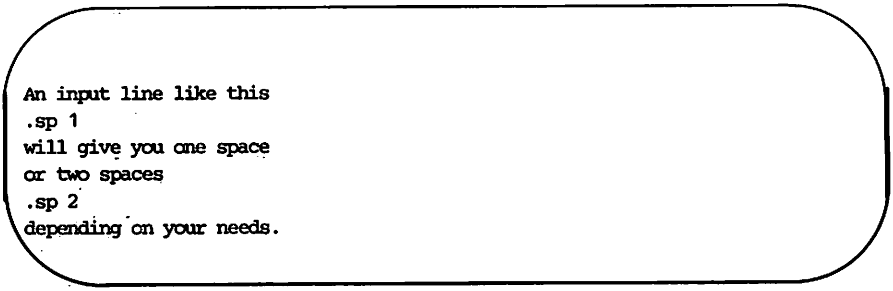
could make your reader
believe that he or she

```
.ps 4
.vs 5
```

were going blind.

When used without arguments **.ps** and **.vs** revert to the previous value.

The command **.sp** is also used to get extra vertical space. It does not set the amount of space between all lines of characters as **.vs** does. Rather it specifies the number of spaces you want at a given place in the text. Consider this usage:



```
An input line like this
.sp 1
will give you one space
or two spaces
.sp 2
depending on your needs.
```

The output appears as

```
An output line like this

will give you one space
or two spaces


depending on your needs.
```

Unadorned, **.sp** gives you one extra blank line (one vertical space, whatever **.vs** has been set to). If that's not what you want, **.sp** can be followed by information about how much space to leave:

```
.sp 2i
```

means "two inches of vertical space."

```
.sp 2p
```

means "two points of vertical space." Finally,

```
.sp 2
```

means "two vertical spaces"—two of whatever **.vs** is set to. (This can be made explicit with **.sp 2v**.) **troff** also understands decimal fractions in most request arguments, so

```
.sp 1.5i
```

is a space of 1.5 inches. These same types of measures (inches, points, and spaces) can be used after **.vs** to define line spacing and, in fact, after most commands that deal with physical dimensions.

In the same company with **.vs** and **.sp** is **.ls** (line spacing). **.ls** is typically used at the beginning of a document or block of text to specify the number of vertical spaces between successive pairs of lines. By default this number is one. The numerical arguments it accepts determine the ratio of single blank lines per lines of text. The argument itself is a total of space lines and text lines. Thus, **.ls 2** represents the sum of one line of space plus one line of text. **.ls 3** will give you two spaces for every line of text, and so forth. **.ls** is not a replacement for **.sp**. It sets the default value for line spacing. **.sp** may still be used at any time to specify a particular line spacing.

It should be noted that all size numbers are converted internally to "machine units" (whose ratio to inches varies from machine to machine).

Local Motions

Moving further from the carriage rows of a typewriter, **troff** will let you depart, at a moment's notice, from the place where your next character was to be struck, to a remote part of the page. This local activity of movement is called local motions. Although you might expect such a radical departure from the norm to require elaborate arrangements, local motions are done as font and point size changes are: with a escape sequence.

Vertical Motion (\v)

Consider using vertical motion, for example, to lift and lower individual characters and words:

Up the Down STAIRCASE

The baseline (the line unchanged by vertical motion) is the line where "Down" is sitting.

You can elevate or lower characters with one escape sequence, the \v. To move the word "Up" downward, you simply precede it with the escape sequence together with directions for the distance you want "Up" to travel:

```
\v'+1'Up
```

This says, in effect, move down (a positive direction in the world of forward-moving text). As you might imagine, all words following the vertical motion escape sequence will also be elevated or lowered at precisely the level dictated by the escape sequence. To return to the initial vertical position, simply give the opposite escape sequence argument: in this case **\v'–1'**. The next word from the example, "the," is preceded by **\v'–0.5'**, thus moving back half the distance from the place where "Up" was printed. "Down"'s escape sequence, likewise, is **\v'–0.5'**, bringing that word back to the baseline. The input line for these words looks like this:

```
\v'+1'Up \v'-0.5'the \v'-0.5'Down
```

We could have substituted '1' for the argument '+1'. As is often the case in **troff**, arguments that increment or decrement assume "+" to be default. Only the explicit "–" will, in the case of \v, give you upward movement.

Notice that the local motions escape sequence, **\v**, is similar to the font and point size escape sequences except for one thing. Unlike those near relatives, **\v** encloses its argument in single quotes (apostrophes). The numerical argument you give the **\v** escape sequence specifies units of **.vs** (whatever you set vertical spacing to). If vertical spacing is set to be one inch, then **\v′−1′** will move the characters following it upward a distance of one inch.

An alternative to **\v** are the escape sequences **\u**, the up escape sequence, and **\d**, the down escape sequence. These do not have the range of **\v**, but their simplicity comes in handy for some commonly encountered words or expressions:

```
Footnotes\u7\d
14\u\s610\s0\d
Trademarks\u\s6TM\s0\d
```

for example, will give you

Footnotes^7
14^{10}
Trademarks^{TM}

Notice the symmetry of **troff** usage. Like **\v**, which requires a companion escape sequence to return to the baseline, **\u** and **\d** accompany each other: for every *down* there must be an *up* and vice versa (unless, of course, you don′t want to return). The exponent and trademark examples are especially interesting ones. At the center of the activity are the characters **10** and **TM** waiting both to be elevated and shrunk by the escape sequences **\u** and **\s6**, respectively. All following characters, however, will remain elevated and reduced if **10** and **TM** are not followed by escape sequences bearing values opposing the initial settings.

\s0 is especially useful for local motions since you often do not care what the former point size is; you simply want to return to it.

Horizontal Motion (\h)

At present, you have learned to travel the upward and downward corridors of a page. To move freely in any direction, you need the further capability to move to the left and right. This you can do with the escape sequence **\h**. Like **\v**, **\h** requires arguments that are set off by single quotes (apostrophes) and are given with minus (−) or plus (+) signs. The + will give you rightward movement (a positive direction in the context of **troff** output); the − will give you leftward movement. As with **\v**, + is default, and the omission of the + or − will be understood to be a positive, or rightward, movement. Consider the following examples: Typing

```
<\h'-0.3m'>
[\h'-0.2m']
```

will give you a simple diamond and rectangle, respectively:

```
<>
[]
```

But

```
<\h'0.3m'>
[\h'0.2m']
```

will spread these companion marks apart according to your numerical argument:

```
< >
[ ]
```

Notice that these arguments specify measures in ems (about the width of an *m* in whatever point size that applies at the time). The em is the default for the horizontally-oriented requests and escape sequences, and these escape sequences could have been expressed as **\h'−0.3'** and **\h'−0.2'**, respectively. The **\h** escape sequence also accepts other measures including inches (in decimal fractions), picas, and machine units.

NOTE

See the "**nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual* for an explanation of these measures.

The additional use of point size (**.ps**) and emboldening (**.bd**) will, of course, give you variations on these themes. While the preprocessor **pic** enables you to draw pictures, **troff**'s ability to combine existing characters to make new ones can also be useful.

Creating New Characters With Local Motion

Consider the following example, which uses characters made by both horizontal and vertical motion:

> WHen **Turnus** in this finall fight downethrowne, his flittring ghost
> Had yælded vp into $\overset{e}{y}$ aire, in middest of all the host
> **Aeneas** valient victour stands, god **Mauors** chapion bold.
> The **Latines** stoynisht standing, from their hartes great groanes vnfold,
> And dœpely from their inward thoughts reuoluing cause of care,
> Their daunted minds they do let fall

The input for the first word of this passage looks like this:

```
\v'1.7'\s36V\h'-0.5m'V\s0\v'-1.7'Hen
```

The thirty-six point size of the **W** was incremented and decremented by the **\s** escape sequence. The more interesting aspect of this construction, though, is the drawing together of the two **V**s with the **\h** escape sequence. The second **V** moved back toward the first a distance of two ems (at thirty-six point, remember) until they actually crossed and formed another character. The downward movement (**\v'1.7'**) and opposing upward movement (**\v'-1.7'**) was done to allow for space and to position the character gracefully. The surrounding text was offset with the temporary indent request **.ti 0.5i** moving it horizontally a half an inch to the right.

One character (it's actually a word) that probably caught your eye is the early English contraction for *the*: $\overset{e}{y}$. It is created using two unconventional features: horizontal and vertical motion and size change. To make $\overset{e}{y}$, we must reduce the size of the **e** and move it both slightly backwards and above the **y**. We determine size change by inserting **\s** before the character we want to alter. Thus, ye as a line of processed text would be accomplished by typing **y\s6e** as a line of input. The **y** appears in the default point size, ten, and the *e* appears in point size six, dictated by **\s6**.

We determine vertical motion, similarly, with the escape sequence **\v**. Thus,

```
y\s6\v'-0.5'e
```

will produce y^{e}. We specify that *e* be moved up one-half of a unit.

Finally, we want to move the e to the left.

```
y\s6\v'-0.5'\h'-0.35m'e\h'0.35m'\v'0.5'\s0
```

produces $\overset{e}{y}$. Again, notice that the amount of motion following **\v** and **\h** is conventionally set off by single quotes.

As you have seen throughout these tutorials, the language of **nroff** and **troff** is one of graceful symmetry. In the example above, for every **\s6**, we must counterpose a **\s0** to cancel its predecessor. Otherwise, everything following the initial point size change would be reduced to that specified change. Likewise, every **\v'-0.5'** must be followed with a countervailing **\v'+0.5'**, and every **\h'−0.35m'** must be followed with **\h'+0.35m'**. Thus, our $\overset{e}{y}$ is formed by

```
y\s6\v' -0.5' \h' -0.35m' e\h' 0.35m' \v' 0.5' \s0
```

Such a word is obviously too cumbersome to type every time we want to produce so brief a word as $\overset{e}{y}$. The solution, as the **nroff** tutorial shows, is to store it in a macro or string and call it with.yE or *(yE.

Finally, there are also several special-purpose **troff** formatting commands for local motions. **\0** is an unpaddable white space of the same width as a digit. Unpaddable means that it will never be widened or split across a line by line justification or filling. There is also \ (backslash blank), which is an unpaddable character the width of a space; \|, which is half that width; \^, which is one quarter the width of a space; and **\&**, which has zero width and is often used as the first character on a text line that begins with a dot (.), which otherwise would be read as a formatting command.

The escape sequence **\o** used like

```
\o'set of characters'
```

causes (up to nine) characters to be overstruck, centered on the widest. This is nice for accents, as in

```
syst\o"e\`"me t\o"e\´"l\o"e\´"phonique
```

which makes

    système téléphonique

The input for the accents are \' and \', or **\(ga** (grave accent) and **\(aa** (acute accent).

You can make your own overstrikes with another special convention, **\z**, the zero-motion command. **\z***x* suppresses the normal horizontal motion after printing the single character *x*, so another character can be laid on top of it. Although sizes can be changed with **\o**, it centers the characters on the widest, and there can be no horizontal or vertical motions, so **\z** may be the only way to get what you want: in this case, a tunnel of squares.

is produced using the predefined **troff** escape sequence for making squares, **\(sq**.

```
.sp 2
\s8\z\(sq\s14\z\(sq\s22\z\(sq\s36\(sq
```

The **.sp 2** is needed to leave room for the result.

As another example, an extra-heavy semi-colon that looks like

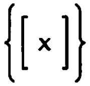
; instead of ; or ;

can be constructed with a big comma and a big period above it:

```
\s+6\z,\v'-0.25m'.\v'0.25m'\s0
```

"0.25m" is an empirical constant.

A more ornate overstrike is given by the bracketing function, **\b**, which piles up characters vertically, centered on the current baseline. Thus, we can get big brackets, constructing them with piled-up smaller pieces:



by typing in this:

```
\b' \(lt\(lk\(lb' \b' \(lc\(lf' x \b' \(rc\(rf' \b' \(rt\(rk\(rb'
```

In this case, the piled-up pieces are given in two groups on either side of the **x**, each delimited with a leading **\b**. The first forms a large, left-hand curly bracket:

\(lt, or left top of curly bracket (⎧)
\(lk, or left center of curly bracket (⎨)
\(lb, or left bottom of curly bracket (⎩)

Following the \b as they do, they all must back-up to be overstruck, forming the curly bracket. The next group forms a left-hand square bracket:

> **\(lc**, or left ceiling of square bracket (⌈)
> **\(lf**, or left floor of square bracket (⌈)

The groups to the right of the **x** simply form the right-hand counterparts to the left-hand curly and square brackets. The important thing these pieces have in common is that they are delimited by the **\b** with its associated single quotes and are therefore all subject to overstriking the other members of their groups.

troff also provides a convenient facility for drawing horizontal and vertical lines of arbitrary length using arbitrary characters. **\l'1i'** draws a line one inch long, like this: ______________. The length can be followed by the character to use if the "_" doesn't suit you. **\l'1i.'** will draw, for example, a one-inch line of dots: .... The construction **\L** is analogous except that it draws a vertical line instead of horizontal.

Programming in troff

In the preceding tutorials you have seen the requests, escape sequences, and macros you need to produce a wide range of text formatting features. These formatting commands, concerned with page control, might be viewed as the surface of **nroff** and **troff**. They stretch and diminish indents, line lengths, page lengths, tabs, and the rest depending on the arguments you give each request or macro.

The capabilities of **nroff** and **troff**, however, go beyond interpreting and carrying out your formatting instructions. They are able to store them, remember them, and use them again. What is more, **nroff** and **troff** can use arithmetic to evaluate and compute such stored information. This below-the-surface activity gives you something truly extraordinary in text processing: formatters that you can program.

You might choose to ignore what goes on below-the-surface. This would be fine: you might simply build your own library of macros and escape sequences and not be troubled with the rest. If, on the other hand, you want to explore this new territory, it is there for the taking.

Number Registers

You have already read about number registers in the **nroff** tutorial in this guide. They are the storage places for various page values. The amount of indent, for example, is stored in a place called the **.i** register. As you change the indent in the customary way (**.in +1i**), the register **.i** also changes. You cannot change the **.i** register directly; it is a read-only register (unlike various others) and changes automatically according to the arguments you make to the indent request. Let's see how this works.

You ask for the value of the register, whatever it is at the time, by preceding the register name with the characters **\n** for one-character register names and **\n(** for two-character register names. Thus, **\n%** would give you the value of the page number register, %, and **\n(.l** would find the value of the line length register (.l). Because the indent register is **.i**, we derive its value with **\n(.i**. If you set the indent to zero, the register value will be zero. A file that contains

```
.in 0
\n(.i
```

will yield an indent as far to the left as possible and the number 0. Let's see what happens to the .i register when we increment .in one inch. Typing

```
.in +1i
\n(.i
```

in your input file would give you the following in your output:

```
723
```

You would get the number 723, the number of basic units for the APS-5 phototypesetter, and all succeeding text, as you can see, would be moved over one inch from the previous left-hand margin. These basic units reflect the one-inch indent just made plus whatever indent is controlling this page at the time of printing. (The number of basic units for **nroff** is 240; units are typesetter-dependent for **troff**.) Typing a simple **.in** will give you the previous indent.

The fact that the value of registers is calculated in units need not interest you. What is interesting is that you can use arithmetic to find and use valuable information about any given page.

Imagine, for instance, that you wanted to indent a block of text a distance that would be proportionately offset no matter what the size of the page might be. A single, numerical incrementation naturally would pose problems. What would be indented one-fourth of a page to the right on a small page might be indented one-sixth the width of a larger page. Using the value of registers, however, we avoid this obstacle. If we added the value of the current line length (**\n(.l**) to the current indent (**\n(.i**) and divided the sum by four, we could find one-fourth the page width whatever the page's size might be.

So that this device will be easier to use, let's define it as a macro and give it a name that identifies its function of indenting one-fourth of a page:

```
.de i4
.in( \\n(.i+\\n(.l)/4
..
```

This macro definition (which you would place at the beginning lines of your document) is simply made of a request and one very long argument. The arithmetic expression used to define the macro would have been easier to read had we used blanks (white space) to set off the expressions elements. **nroff/troff** arithmetic expressions, unfortunately, do not permit the use of white space. The parentheses are used to ensure that the addition of both the line length and the indent will be divided by four.

Notice the double backslashes following the macro definition. Some backslashes between the **.de** request and its accompanying **..** request must be protected by an extra backslash. The extra backslash in the example above delays interpretation of the register since the value of that register could change from the time the **troff** is copied to the time the document is printed.

The input file would look like this. (Since this example demonstrates indentation, it has not itself been indented):

```
.P
This is normal text that should be subject to current line length
and indent values.
That is, this text should be identical to the text that has
preceded it on this tutorial page.
.i4
This block, however, should be indented by a distance equal to
twenty-five percent of the text preceding it.  And all text
following the \f3.i4\f1 macro should be similarly indented.
.in
Text following the request for the previous indent should
bring things back to normal.
```

The output file is next:

This is normal text that should be subject to current line length and indent values. That is, this text should be identical to the text that has preceded it on this tutorial page.

> This block, however, should be indented by a distance equal to twenty-five percent of the text preceding it. And all text following the .i4 macro should be similarly indented.

Text following the request for the previous indent should bring things back to normal.

These basic concepts—deriving the value of number registers and using arithmetic to compute those values—are important and constitute two basic techniques essential to advanced text programming. Be sure you understand them.

NOTE You will find a complete list of number registers in the "**nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual.*

Traps

This tutorial will present two additional concepts that are frequently used in text programming: testing and conditional statements. The page trap is a rudimentary way of understanding and using both of these concepts.

The need request (**.ne**) is, in effect, a simple page trap. **.ne** followed by a number (which represents a number of output lines) both tests data and conditionally responds to that data. Supposing it were important to print a block of text uninterrupted by page breaks, you would want to find how far from the page's bottom your text block might be. The **.ne** request would conduct that test and print the block if you had enough room or save the block for the next page if you did not. Because this material is covered earlier in the **nroff** tutorial, a brief example will suffice:

```
.ne 3
.nf
This block of text, which takes up three output lines,
cannot be separated.  The ability to hold text together is
especially important when comparing examples or illustrating a point.
```

If two lines of space remain on the page, this entire block will be printed, intact, at the start of the following page.

The page trap is a more sophisticated version of the **.ne** request. Like the need request, the page trap tests data and responds conditionally. Unlike the need request, the page trap does more than print or fail to print.

You set the page trap with the **.wh** (when) request. The first argument to **.wh** is either zero, indicating the top of the page, or a negative number, indicating distance from the bottom. The arguments are then followed by a request or macro you want to begin operation *when* you cross the trap. Consider the examples:

```
.de hD   \" define header
'sp 1i          \" space one inch
..       \" end definition
.de fO   \" define footer
'bp      \" break page
..       \" end definition
.wh 0 hD \" set top-of-page trap
.wh -1i fO      \" set bottom-of-page trap
```

You probably wondered why the macros **.hD** and **.fO** lacked dots in the page trap examples. These are macros whose names are given to **.wh**; they are not actual macro calls in this instance.

Because the examples in this section will become increasingly more complex, key lines will be followed by comments. Remember, comments preceded by .\" or '\" are neither printed nor interpreted by the system.

The preceding example emphasizes the latitude of the trap: where you can include one macro, by virtue of defining new macros, you can include any and everything. In this case, **.hD** and **.fO** are defined simply. When you get to the header (determined by the **0**), output a one-inch space; when you get to within one inch of the page's bottom, break page. Notice that you are limited to one numerical argument per **.wh** request since these are to set page locations for springing their respective traps.

Notice that the **.sp** and **.bp** requests, which normally cause a line break, begin with "'" instead of "." to suppress the line break function. This is an important, if complex, part of setting page traps. Often in fill mode the output line that springs the trap is not neatly processed. Some word or part of a word might be squeezed out. Should that temporarily stray fragment meet up with a line break while going through a trap, it would be lost. The "'" character suppressing that break function thus plays an important role in protecting such text.

NOTE

For a list of requests that force a line break, see the "**nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual.*

Defining macros with additional escape sequences, requests, registers, and traps can be used to accomplish a great deal of text processing:

```
.de hD                        \" define header
'sp ¦ 0.5i
.tl ''' \\n(mo/\\n(dy/\\n(yr \\n%'   \" give date and page number
'sp 0.5i
..
.de fO                        \" define footer
'sp 0.2i
.ps 12                        \" set title in twelve point
.ft I                         \" set title in italic
.ce 1                         \" center title at the bottom of each page
Tutorial
.ps                           \" restore point size
.ft                           \" restore font
..
.wh 0 hD                      \" set hD to activate at header trap
.wh -1i fO                    \" set fO to activate at footer trap
```

This header prints the date (current each day) and current page number at the upper right hand of the page. The footer prints an essay title in italics at twelve point, reverts to the previous font and point size and calls for the next page to be printed. Because you can define macros to collect and print text and register values as well as to perform requests, escape sequences, macros, you begin to see how powerful the page trap can be. But more on traps later. Instead let's explore testing and conditional statements further.

Conditional Acceptance of Input (.if, .ie, .el)

Testing and conditional activity, at this point, are nothing novel in **nroff/troff**. You have seen them in the need request, which specified when text would be processed, and in the page trap, which specified both when and what text would be processed. The if requests (**.if**, **.ie**, **.el**) add one more capability to these: they decide whether text will be processed.

The two tables below will give a rough overview of the if requests and will set out terms from which to work. In the following, *c* is a one-character, built-in condition name, **!** signifies not, *N* is a numerical expression, **u** stands for basic units (a device-dependent measure), *string1* and *string2* are strings delimited by any non-blank, non-numeric character *not* in the strings, and *anything* represents what is conditionally accepted:

Request Form	Explanation
.if *c anything*	If condition *c* true, accept *anything* as input; in multi-line case use \\{*anything*\\}.
.if !*c anything*	If condition *c* false, accept *anything*.
.if *N anything*	If expression $N > 0$, accept *anything*.
.if !*N anything*	If expression $N \leqslant 0$, accept *anything*.
.if *'string1'string2' anything*	If *string1* identical to *string2*, accept *anything*.
.if !*'string1'string2' anything*	If *string1* not identical to *string2*, accept *anything*.
.ie *c anything*	If portion of if-else; all above forms (like **.if**).
.el *anything*	Else portion of if-else.

The built-in condition names are:

Condition Name	True If
o	Current page number is odd
e	Current page number is even
t	Formatter is **troff**
n	Formatter is **nroff**

If the condition *c* is true, then *anything* (requests, escape sequences, or text) is accepted for input. Likewise, if the number *N* is greater than zero, anything following the conditional expression is input. If the strings compare identically (including motions and character size and font), anything is accepted as input. Finally, if a ! precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multi-line case, the first line must begin with a left delimiter \\{ and the last line must end with a right delimiter \\}.

The request **.ie** (if-else) is identical to **.if** except that the acceptance state is remembered. A subsequent and matching **.el** (else) request then uses the reverse sense of that state. **.ie-.el** pairs may be nested.

All this may seem daunting at first, but a few examples will show the **.if** requests to be in fact quite tame. Perhaps the most common use of these requests is to distinguish between **nroff** and **troff** parameters. Often a document is destined to be formatted by each of these programs depending on the occasion. Preparing two different versions would be wasteful; the **.if** requests allow you to prepare two sets of parameters for one document. Each set automatically activates when its corresponding formatter is used.

The following abbreviated examples show how this is done:

```
.if n \{ \                  \" if nroff
.ft R                       \" use roman font
.ps 10                      \" use point size ten
etc. \ }
.if t \{ \                  \" if troff
.fp 1 PA                    \" load Palatino font
.ft PA                      \" use Palatino font
.ps 9                       \" use point size nine
etc. \ }
```

You undoubtedly noticed that **n** and **t** correspond to **nroff** and **troff**, respectively. These are permanent, predefined condition names and will never change. When using **nroff** to format your document, the **.if** request asks that parameters suitable for a mechanical printer be used; if **troff** were used, a format appropriate to digital typography would be used.

We could have done this in a slightly more straightforward fashion:

```
.ie t \{ \                  \" if troff
.fp 1 PA                    \" load Palatino at position one
.ft PA                      \" use Palatino
.ps 9                       \" use point size 9
etc.
.el \{ \                    \" or else
.ft R                       \" make roman your predominant font
.ps 10                      \" use point size ten
etc. \ }
```

If **troff** is used, execute all formatting commands inside the corresponding delimiters. If not, then do all formatting commands within the second set of delimiters. Since the only alternative to **troff** is **nroff**, else will always mean **nroff** in such a case.

Each set of **.if** requests has its own corresponding pair of delimiters. The delimiters are to unify several lines of formatting commands into a single package belonging to its controlling **.if** request. Had the request **.el** been followed by a single request, the delimiters would not have been necessary:

```
.el .ft R
```

(Notice that in previous examples, subsequent requests or macros on control lines do include dots, unlike the **.wh** request.)

The preceding example would have been duplicated, in effect, with this even shorter solution:

```
.if t \{ \
.fp PA
.ft 1 PA
.ps 9
... \ }
```

If you were to use **nroff**, the **.if** request would be ignored, and you would get default values (which we were assigning to **nroff** anyway). If you used **troff**, the parameters following **.if** would apply.

Notice that in the above examples the opening if-statement delimiter (\{) is trailed by a final backslash, escaping the newline character. The starting delimiter expects to be followed by a formatting command or text. By escaping the newline, this requirement is met.

If we had used .if requests in our previous example of page traps for headers and footers, it might have looked something like this:

```
.de hD
.ie \\n%>1 \{\          \" For all pages following page one do the following:
.ie e .tl '%'''         \" place page number at upper left of even pages
.el .tl '''%'           \" place page number at upper right of odd pages
'sp ¦ 1i \}             \" then space down one inch
.el 'sp ¦ 1.25i\        \" Don't number first page and make larger header
..
.wh 0 hD
```

Notice the nesting in this example. Inside the main if-else statement (which determined activity based on first-page and non-first-page status) was another if-else (which determined activity based on even and odd pages). This nest of conditional statements was delimited by \{\ and \}. Like **n** and **t**, **e** and **o** are predefined condition names for even and odd pages, respectively. These too are permanent definitions.

A more realistic example of header and footer control would be the following:

```
.de hD                      \" header
.if t .tl ' −'' −'          \" troff cut mark
.if \\n%>1 \{ \
'sp ¦ 0.5i−1                \" tl base at 0.5i
.tl ''− % −''               \" centered page number
.ps                         \" restore size
.ft                         \" restore font
.vs  \ }                    \" restore vs
'sp ¦ 1.0i                  \" space to 1.0i
.ns                         \" turn on no-space mode
..
.de fO                      \" footer
.ps 10                      \" set footer/header size
.ft R                       \" set font
.vs 12p                     \" set base line spacing
.if \\n%=1 \{ \
'sp ¦ \\n(.pu−0.5i−1        \" tl base 0.5i up
.tl ''− % −'' \ }           \" first page number
'bp
..
.wh 0 hD                    \" set top-of-page trap
.wh −1i fO                  \" set bottom-of-page trap
```

This arrangement sets the size, font, and base line spacing for the header/footer material and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages.

If you are typesetting text on a device that produces a continuous role of paper, you must specify a cut mark at the bottom of each page. This was done above by simply requesting that hyphens be drawn using the **.tl** request.

The **.sps** refer to absolute positions to avoid dependence on the base line spacing. Another reason for absolute spacing in the footer is that the footer is invoked by printing a line whose vertical spacing swept past the trap position by possibly as much as the base line spacing. The no-spacing mode is turned on at the end of **.hd** to render ineffective accidental occurrences of **.sp** at the top of the running text.

The above method of restoring size, font, and leading presupposes that such requests (that set previous value) are not used in the running text. A better scheme is to save and restore both the current and previous values for size as the following shows:

```
.de fO
.nr s1 \\n(.s               \" current size
.ps
.nr s2 \\n(.s               \" previous size
etc.                        \" rest of footer
..
.de hD
...                         \" header stuff
.ps \\n(s2                  \" restore previous size
.ps \\n(s1                  \" restore current size
..
```

Page numbers can be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bN                      \" bottom number
.tl ''– % –''               \" centered page number
..
.wh –0.5i–1v bn              \"tl base 0.5i up
```

While an exhaustive presentation of **troff**'s features and applications is not practical in a tutorial, its great flexibility and precision can certainly be shown. Some of the unusual—in some cases, unique—capabilities we have not discussed are well worth learning through the "**nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual.*

With **troff** you are able to divert text into special save-areas and measure its dimensions, placing the text according to your findings while the job is running. Once you have placed traps, you can calculate the distance to the next trap, defining macros to behave according to that distance. You can create your own automatically incrementing registers and set them according to textual dimensions or contextual page behavior. In short, **troff** provides both a nimble text processing facility as well as an elaborate instrument for determining the weights and measures of words. Finally, these complementary features combine in a programming language that can be effectively used with other UNIX system tools and languages enabling you to undertake applications of considerable variety and scope.

Index

Table of Contents

Introduction

This tutorial describes how to use **eqn** and **neqn**, tools for presenting mathematical notation. These programs are run from the command line as preprocessors, preceding their corresponding formatters, **troff** and **nroff**. With **eqn** or **neqn** you specify mathematical expressions with control lines that you can set up like a display or embed in the running text of a manuscript. This tutorial gives examples of **eqn** output only.

The prerequisites to benefit from this tutorial are as follows:

- You should know what a file and directory are and how to create them. See the *UNIX System V User Guide.*
- You should know how to use a UNIX system text editor (such as **ed** or **vi**) to create and change files. See the *UNIX System V User Guide.*
- You should know how to run programs with options and how to use pipes. See the *UNIX System V User Guide.*
- You must have a working knowledge of **nroff** or **troff**, preferably **troff**. See the tutorials in this guide "The Formatter **nroff**" and "The Formatter **troff**."
- Your understanding of **eqn/neqn** would benefit from knowledge of the **mm** macro package though it is not essential. "The Macro Package **mm**" gets you started, and "The **mm** Macro Package: Technical Discussion" in the *Technical Discussion and Reference Manual* explains it in detail.

NOTE

Other preprocessors that you may use with **eqn/neqn** are discussed in tutorials in this guide. You may also refer to the "**nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual.*

eqn and **neqn** are identical in their input. Their output, though, is different. **neqn** is a preprocessor to **nroff**, and its output is printed on typewriter-like devices such as the DASI 300, 300S, and 450; the C-Itoh 8510; the DTC 382; and the Teletype Model 37. **eqn** is a preprocessor to **troff**, and its output is typeset on digital typesetters such as the Autologic APS 5. The results of **neqn** are a rough draft of **eqn**'s output, because **neqn**'s output devices do not provide the variety of characters, sizes, and fonts that a typesetter does. **neqn**'s output devices, themselves, vary in sophistication. Some, for instance, do not allow for vertical motions necessary for subscripts and superscripts and print such notation on the equation's baseline.

Because the input of **eqn** and **neqn** are so similar, this text uses "**eqn/neqn**" to refer to both programs, unless a functional distinction requires otherwise.

Typesetting Equations with eqn

The following sections introduce you to the macros, keywords, metacharacters, and font and point size controls needed to use **eqn**.

Displayed Equations (.EQ and .EN)

To tell **eqn/neqn** where a mathematical expression begins and ends, use the macro pair .EQ and .EN. Thus if you have a file containing the lines

```
.EQ
x=y+z
.EN
```

your output will look like this

$$x=y+z$$

The following command line uses **troff** to process the output:

eqn [*options*] [*file*] | **troff** [*options*] | *phototypesetter*

The following command line preprocesses the equation for the **nroff** formatter:

neqn [*options*] [*file*] | **nroff** [*options*] | *printer*

Your system administrator should know what printer name is appropriate for your output.

The delimiters .EQ and .EN are copied untouched; they are not otherwise processed by **eqn/neqn**. You must take care of things such as positioning equations on the page yourself. The most common way to handle this is to use the **mm** display macros, which allow you to center or indent text blocks. This tutorial gives tips for using .EQ and .EN with the display delimiters in the section entitled "Troubleshooting."

Any equation can be labeled with an arbitrary equation number, which appears at the right margin when your file is printed. For example, the input

```
.EQ I (2a)
x = f(y/2) + y/2
.EN
```

produces the output

$$x = f(y/2) + y/2 \qquad (2a)$$

Shorthand for In-line Equations (delim)

In a document containing mathematical notation, you should follow mathematical conventions not just in display equations, but also in the body of the text. For example, variable names such as x should be in italic. Although you could do this by surrounding the appropriate input with .EQ and .EN, it would be a nuisance to continually repeat these delimiters.

eqn/neqn provides a shorthand method for presenting short in-line expressions. You can define two characters to mark the beginning and end of in-line expressions, and then use them to type expressions right in the middle of text lines. To set both the left and right characters to pound signs, for example, add the following three lines to the beginning of your document:

```
.EQ
delim ##
.EN
```

Having set these delimiters, you may use them in your text to indicate mathematical expressions that need to be processed by **eqn/neqn**. For example:

```
Let #alpha sub i# be the primary variable, and let #beta# be zero.
Then we can show that #x sub 1# is #>=0#.
```

Spaces, new-lines, and so on, are significant in the text, but not inside the **eqn/neqn** expression itself. More than one expression can occur in a single input line.

> Let α_i be the primary variable, and let β be zero. Then we can show that x_1 is $\geqslant 0$.

Adequate room should be made before and after a line containing an in-line expression so that something such as $\sum_{i=1}^{n} x_i$ does not interfere with the lines surrounding it. Once pound signs, for example, have been made **eqn** delimiters, all instances of them will have a special meaning.

To turn off the delimiters type the following lines:

```
.EQ
delim off
.EN
```

Don't use braces, tildes, circumflexes, or double quotes as delimiters, or you will get unwanted results. Also, you must close in-line font changes (for example, \f3) before you begin in-line equations.

Spaces and Newlines within .EQ and .EN

Spaces and new-lines within an expression are thrown away by **eqn/neqn**. Thus between .EQ and .EN, or between whatever characters you have defined as expression delimiters,

```
x=y+z
```

and

```
x = y + z
```

and

```
x = y
         + z
```

all produce the same output:

$$x=y+z$$

Output Spaces

To force spaces to appear in the output, use a tilde "~" for each space you want:

```
x~=~y~+~z
```

gives

$$x ~=~ y ~+~ z$$

You also can use a circumflex (^), which gives a space that is half the width of a tilde. It is useful for fine-tuning. Tabs also may be used to position pieces of an expression, but the tab stops must be set by **nroff** or **troff**.

Symbols, Special Names, and Greek

eqn/neqn knows some mathematical symbols, some mathematical names, and the upper- and lower-case ancient Greek alphabet. For example,

```
.EQ
x=2 pi int sin ( omega t)dt
.EN
```

produces

$$x=2\pi\int\sin(\omega t)dt$$

Here the spaces in the input are necessary so that **eqn/neqn** can recognize **int, pi, sin** and **omega** as keywords that get special treatment. On output, the **sin**, the digit 2, and the parentheses are each set in roman type instead of italic; **pi** and **omega** become their Greek counterparts; and **int** becomes the integral sign.

When in doubt, leave spaces around separate parts of the input. A common error is to type **f(pi)**, for example, without leaving spaces on both sides of the **pi**. As a result, **eqn/neqn** does not recognize **pi** as a keyword, and it appears as $f(pi)$ instead of $f(\pi)$.

You must tell **eqn/neqn** to look in the file **/usr/pub/eqnchar** for particular names and symbols that it would not understand otherwise. To tell it to look there, use the following command line.

eqn/neqn /usr/pub/eqnchar [*file* **] | troff**

Here is a complete list of the keywords and their corresponding symbols that are in **/usr/pub/eqnchar**:

ciplus	⊕	||	‖	*square*	□
citimes	⊗	*langle*	⟨	*circle*	○
wig	~	*rangle*	⟩	*blot*	■
−wig	*iwig*	*hbar*	ħ	*bullet*	•
>wig	≳	*ppd*	⊥	*prop*	∝
<wig	≲	*<−>*	↔	*empty*	∅
=wig	≅	*<=>*	⇔	*member*	∈
star	•	|<	≮	*nomem*	∉
bigstar	✱	|>	≯	*cup*	∪
−dot	≐	*ang*	∠	*cap*	∩
orsign	∨	*rang*	∟	*incl*	⊑
andsign	∧	*3dot*	⋮	*subset*	⊂
−del	≜	*thf*	∴	*supset*	⊃
oppA	∀	*quarter*	¼	*!subset*	⊆
oppE	∃	*3quarter*	¾	*!supset*	⊇
angstrom	Å	*degree*	°	*scrL*	ℓ
==<	≦	*==>*	≧		

Spaces, Again

The only way that **eqn/neqn** can deduce that some sequence of letters might be special is if that sequence is separated from the letters on either side of it. You can do this by surrounding a keyword by ordinary spaces (or tabs or new-lines), as explained in the previous section.

You can also make special words stand out by surrounding them with tildes or circumflexes:

```
x~=~2~pi~int~sin~(~omega~t~)~dt
```

This is much the same as the last example except that the tildes not only separate the special words like **sin**, **omega**, and so on, but they also result in real spaces in the output, one space per tilde:

$$x\ =\ 2\ \pi\ \int\ \sin\ (\ \omega\ t\ )\ dt$$

You also can separate special words with braces { } and double quotes "...", which have special meanings, as you will see.

Subscripts and Superscripts (sub and sup)

Subscripts and superscripts are obtained with the words **sub** and **sup**.

```
x sup 2 + y sub k
```

gives

$$x^2+y_k$$

eqn/neqn takes care of all the size changes and vertical motions needed to make the output look right. Like all special words, **sub** and **sup** must be surrounded by spaces; **x sub2** will give you *xsub*2 instead of x_2. Furthermore, a space (or a tilde) must mark the end of text to be subscripted or superscripted. A common error is to say something like

```
y = (x sup 2)+1
```

which causes

$$y=(x^{2)+1}$$

instead of the intended

$$y=(x^2)+1$$

Subscripted subscripts and superscripted superscripts also work:

```
x sub i sub 1
```

produces

$$x_{i_1}$$

The same element of an expression can have both a subscript and a superscript if the subscript comes first:

```
x sub i sup 2
```

becomes

$$x_i^2$$

Other than in this special case, **sub** and **sup** group to the right, so

```
x sup y sub z
```

means

$x sup {y sub z }$

not

${x sup y } sub z$

Braces for Grouping

Normally, the end of a subscript or superscript is marked simply by a blank (or tab or tilde). But what if the subscript or superscript is composed of elements that have to be typed in with blanks? In that case, you can use braces ({ and }) to mark the beginning and end of the subscript or superscript:

```
e sup {i omega t}
```

becomes

$$e^{i\omega t}$$

Braces can always be used to force **eqn/neqn** to treat something as a unit or just to clarify your intention. Thus:

```
x sub {i sub 1} sup 2
```

becomes

$$x_{i_1}^2$$

with braces, but

```
x sub i sub 1 sup 2
```

becomes

$$x_{i_1^2}$$

which is clearly different.

Braces can occur within braces if necessary:

```
e sup {i pi sup {rho +1}}
```

is

$$e^{i\pi^{\rho+1}}$$

The general rule is that anywhere you can use a single expression such as *x*, you can use multiple expressions if they are enclosed in braces. **eqn/neqn** looks after all the details of positioning and spacing.

Always make sure you have the right number of braces. Leaving one out or adding an extra causes **eqn/neqn** to complain.

Occasionally you have to print braces. To do this, enclose them in double quotes as follows: `"{"` will give you a literal brace.

Fractions (over)

To produce a fraction, use the word **over**:

```
a+b over 2c = 1
```

gives

$$\frac{a+b}{2c} = 1$$

The line is automatically made the right length and positioned. Braces can be used to clarify what goes over what:

```
{alpha + beta} over {sin (x)}
```

becomes

$$\frac{\alpha+\beta}{\sin(x)}$$

What happens when there is both an **over** and a **sup** in the same expression? In such an ambiguous case, **eqn/neqn** interprets the **sup** before the **over**, so

```
-b sup 2 over pi
```

becomes

$$\frac{-b^2}{\pi}$$

instead of

$$-b^{\frac{2}{\pi}}.$$

The rules that decide what operation is done first in cases like this are summarized below in the section, "Keywords and Precedences." When in doubt, however, braces will make clear what goes with what.

Square Roots (sqrt)

To draw a square root, use **sqrt**:

```
sqrt a+b + 1 over sqrt {ax sup 2 +bx+c}
```

is

$$\sqrt{a+b}+\frac{1}{\sqrt{ax^2+bx+c}}$$

Square roots of tall quantities do not look attractive, because a root-sign big enough to cover the quantity is too dark and heavy:

```
sqrt { a sup 2 over b sub 2 }
```

is

$$\sqrt{\frac{a^2}{b_2}}$$

Big square roots are generally better written in an alternative style of mathematical notation:

```
(a sup 2 /b sub 2 ) sup half
```

which becomes

$$(a^2/b_2)^{½}$$

Summation and Integral (sum, from, and to)

Summations, integrals, and similar constructions are easy:

```
sum from i=0 to {i= inf} x sup i
```

produces

$$\sum_{i=0}^{i=\infty} x^i$$

Notice that the braces specify where the upper part, $i=\infty$, begins and ends. No braces were necessary for the lower part, $i=0$, because it contained no blanks. If the **from** and **to** parts contain any blanks, you must use braces around them.

The **from** and **to** parts are both optional, but if both are used, **from** must always precede **to**.

Other useful characters can replace the **sum** in our example:

```
int   prod   union   inter
```

become

$$\int \quad \Pi \quad \cup \quad \cap$$

Since the expression before the **from** can be anything, even something in braces, **from** and **to** can often be used in unexpected ways:

```
lim from {n -> inf} x sub n =0
```

becomes

$$\lim_{n \to \infty} x_n = 0$$

Diacritical Marks (dot, dotdot, hat, tilde, vec, dyad, bar, and under)

There are several keywords that will produce diacritical marks:

x dot	$\dot{x}$
x dotdot	$\ddot{x}$
x hat	$\hat{x}$
x tilde	$\tilde{x}$
x vec	$\vec{x}$
x dyad	$\overleftrightarrow{x}$
x bar	$\bar{x}$
x under	$\underline{x}$

The diacritical mark is placed at the correct height and centered over the letter. The **bar** and **under** marks are made the right length for the entire construct, as in $\underline{x+y+z}$.

Quoted Text

Any input enclosed in quotes ("...") is not subject to any of the font changes and spacing adjustments normally done by **eqn/neqn**; therefore, you can adjust your own spacing if needed:

```
italic "sin(x)" + sin (x)
```

becomes

$$\mathit{sin(x)}+\sin(x)$$

Quotes are also used to get braces and other **eqn/neqn** keywords printed:

```
"{ size alpha }"
```

becomes

{ *size alpha* }

and

```
roman "{ size alpha }"
```

becomes

{ size alpha }

The quote construction, "", is often used as a place-holder when **eqn/neqn** needs something to fulfill a syntactic requirement, but you don't actually want anything in your output. For example, to make $"" sup 2 roman Be$, you can't just type **sup 2 roman Be** because a **sup** has to be preceded by something it can be a superscript of. Thus you must say

```
" " sup 2 roman Be
```

To print an actual quotation mark use """. **troff** characters like **\(bs** can appear unquoted, but something more complicated, like horizontal and vertical motions with **\h** and **\v**, should always be quoted.

NOTE To learn more about **\h** and **\v**, see the "**nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual.*

Lining up Equations (mark and lineup)

Sometimes it's necessary to line up a series of equations at some horizontal position, often at an equal sign. You can do this with **mark** and **lineup**.

The word **mark** may appear only once, at any place, in an equation. It remembers the horizontal position where it appeared. Successive equations can contain one occurrence of the word **lineup**. The place where **lineup** appears is made to line up with the place marked by the previous **mark** if possible. Thus, for example, you can say

```
.EQ
x+y mark = z
.EN
.EQ
x lineup = 1
.EN
```

to produce

$$\begin{aligned} x+y &= z \\ x &= 1 \end{aligned}$$

Brackets, Braces, Parentheses, Bars, and Floor/Ceiling

To get big brackets [], braces {}, parentheses (), and bars || around expressions, use the **left** and **right** keywords:

```
left { a over b + 1 right }
 ~=~ left ( c over d right )
 + left [ e right ]
```

becomes

$$\left\{ \frac{a}{b}+1 \right\} = \left(\frac{c}{d} \right) + \left[e \right]$$

The resulting brackets are made big enough to cover whatever they enclose. Other characters can be used besides these, but they are not likely to look good. Two exceptions are the **floor** and **ceiling** characters:

```
left floor x over y right floor
<= left ceiling a over b right ceiling
```

produces

$$\left\lfloor \frac{x}{y} \right\rfloor \leqslant \left\lceil \frac{a}{b} \right\rceil$$

The **right** construction may be omitted: a "left something" need not have a corresponding "right something." If the **right** part is omitted, put braces around the thing you want the left bracket to encompass. Otherwise, the resulting brackets may be too large.

If you want to omit the **left** part, things are more complicated because technically you can't have a **right** without a corresponding **left**. Instead you have to say

left "" ..... right)

for example. The *left* "" means a "left nothing." This satisfies the rules without hurting your output.

Several warnings about brackets are in order. First, braces are typically bigger than brackets and parentheses, because they are made up of three, five, seven, or more pieces, while brackets can be made up of two, three, or more. Second, big left and right parentheses often are not attractive in print.

Pile (above, pile, lpile, rpile, and cpile)

pile is a general facility for making vertical piles of things. For example:

```
A ~=~ left [
  pile { a above b above c }
  ~~ pile { x above y above z }
right ]
```

makes

$$A = \left[\begin{matrix} a & x \\ b & y \\ c & z \end{matrix} \right]$$

The elements of the pile (there can be as many as you want) are centered one above another, at the correct height for most purposes. The keyword **above** is used to separate the elements of the **pile**; braces are used to delimit the list. The elements of a pile can be as complicated as needed, even containing more piles.

There are three other forms of pile: **lpile** makes a pile in which the elements are left-justified; **rpile** makes a right-justified pile; and **cpile** makes a centered pile, just like **pile**. The vertical spacing between the pieces is somewhat larger for **l-**, **r-** and **cpiles** than it is for ordinary piles.

```
roman sign (x)~=~
left {
  lpile {1 above 0 above -1}
  ~~ lpile
   {if~x>0 above if~x=0 above if~x<0}
```

makes

$$\mathrm{sign}(x) = \left\{ \begin{array}{ll} 1 & \mathrm{if}\ x>0 \\ 0 & \mathrm{if}\ x=0 \\ -1 & \mathrm{if}\ x<0 \end{array} \right.$$

Notice that the brace is not part of **pile**'s output, but appears because the keyword **left** construction was also used in this expression. This example also shows the use of a left brace without a matching right one.

Matrices (lcol, rcol, ccol, and matrix)

It is also possible to make matrices. For example, you can use the keywords **matrix** and **ccol** like this

```
matrix {
  ccol { x sub i above y sub i }
  ccol { x sup 2 above y sup 2 }
}
```

to array these elements neatly like this

$$\begin{matrix} x_i & x^2 \\ y_i & y^2 \end{matrix}$$

This produces a matrix with two centered columns. The elements of the columns are listed, inside braces, just as for a pile, each element separated by the word **above**. You can also use **lcol** or **rcol** to produce left or right adjusted columns. Each column can be separately adjusted, and there can be as many columns as you like.

The advantage of using a matrix instead of two adjacent piles is that piles may not line up properly if all the elements don't have the same height. A matrix forces elements to line up because it looks at the entire structure before deciding on spacing.

A word of warning about matrices: each column must have the same number of elements in it.

Definitions (define, ndefine, and tdefine)

eqn/neqn provides a facility for naming a frequently-used string of characters, so thereafter you can just type the name instead of the whole string. For example, if the sequence

```
x sub i sub 1 + y sub i sub 1
```

appears repeatedly throughout a paper, you can avoid re-typing it each time by defining it like this:

```
define xy 'x sub i sub 1 + y sub i sub 1'
```

xy can now be used to stand for whatever characters occur between the single quotes in the definition. You can use any character instead of quotes to mark the beginning and end of the definition, so long as it doesn't appear inside the definition.

Now you can use **xy** like this:

```
.EQ
f(x) = xy . . .
.EN
```

and so on. Each occurrence of **xy** expands into its definition. Be careful to leave spaces or their equivalent around the name when you use it, so **eqn/neqn** will be able to identify the defined name as a special word.

There are several things to watch out for. First, although definitions can use previous definitions, as in

```
.EQ
define xi  ' x sub i '
define xi1  ' xi sub 1 '
.EN
```

don't define something in terms of itself. A common error is to type

```
define X  ' roman X '
```

This is a guaranteed disaster since **X** is now defined in terms of **X**. If you type

```
define X  ' roman "X" '
```

however, the quotes prevent the second **X** from being read as a keyword, and everything works fine.

eqn/neqn keywords can be redefined. You can make / mean **over** by saying

```
define /  ' over '
```

or redefine **over** as / with

```
define over  ' / '
```

If you need to print an element differently on a terminal screen than it is printed on a typesetter you can define the same symbol specifically for **neqn** and specifically for **eqn**. This is done with **ndefine** and **tdefine**. A definition made with **ndefine** only takes effect if you are running **neqn**; one made with **tdefine** only applies for **eqn** Names defined with plain **define** apply to both **eqn** and **neqn**

Size and Font Changes (size, font, roman, italic, bold, and fat)

By default, equations typeset with **eqn/neqn** are put in 10-point type, and use standard mathematical conventions to determine which characters are in roman and which in italic. **neqn** constrains equations to your printer's capabilities. So if you are only going to use **neqn**, you can skim the information about size changes.

Although **eqn/neqn** makes a valiant attempt to use aesthetically pleasing sizes and fonts, it is not perfect. To change sizes and fonts, use **size** *n* and **roman, italic, bold** and **fat**. Like **sub** and **sup**, size and font changes affect only the element that follows them, and the expression reverts to the normal case when it reads a blank space. Thus

```
bold x y
```

becomes

$$\mathbf{x}y$$

and

```
size 14 bold x = y +
   size 14 {alpha + beta}
```

gives

$$\mathbf{x}=y+\alpha+\beta$$

As always, you can use braces if you want to affect something more complicated than a single letter. For example, you can change the size of an entire equation to 12 point by

size 12 { ... }

Legal sizes that may follow **size** are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, 36. You can also change the size by a given amount; for example, you can say **size +2** to make the size two points bigger, or **size −3** to make it three points smaller. The advantage of this method is that you don't have to know what the current size is.

If you are using fonts other than roman, italic, and bold, you can say **font** X where X is a one character **troff** name or number for the font. Since **eqn/neqn** expects roman, italic, and bold, other fonts may not give as good an appearance.

The **fat** operation widens a font by overstriking: **fat grad** is $\boldsymbol{\nabla}$ and **fat** {**x sub i**} is $\boldsymbol{x_i}$.

If an entire document is to be in a non-standard size or font, it would be a severe nuisance to have to write out a size and font change for each equation. Accordingly, you can set a global size or font to thereafter affect all equations. At the beginning of any equation, you might say, for instance,

```
.EQ
gsize 16
gfont R
  . . .
.EN
```

to set the size to 16 and the font to roman thereafter. In place of R, you can use any of the **troff** font names. The size after **gsize** can be a relative change with + or −.

Generally, **gsize** and **gfont** appear at the beginning of a document, but they can also appear throughout a document: the global font and size can be changed as often as needed. For example, in a footnote in which the relative point size and leading is smaller than in general text, the size of equations should match the size of the footnote text. Don't forget to reset the global size at the end of the footnote.

Local Motion (back, fwd, up, and down)

Although **eqn/neqn** tries to get most things at the right place on the paper, it isn't perfect, and occasionally you need to fine-tune the output to make it just right. Small, extra horizontal spaces can be obtained with tilde and circumflex. You can say **back** *n* and **fwd** *n* to move small amounts horizontally. *n* is how far to move in 1/100's of an em (an em is about the width of the letter m). Thus **back 50** moves back about half the width of an m. Similarly you can move things up or down with **up n** and **down n**. As with **sub** or **sup**, a local motion affects only the next element in the input, which can be several elements enclosed in braces.

Keywords and Precedences

If you don't use braces, **eqn/neqn** does operations in the order shown in each line in this list.

dyad vec under bar tilde hat dot dotdot
fwd back down up
fat roman italic bold size
sub sup sqrt over
from to

These operations group to the left:

over sqrt left right

All others group to the right.

Digits, parentheses, brackets, punctuation marks, and these mathematical words are converted to roman font when encountered:

sin cos tan sinh cosh tanh arc
max min lim log ln exp
Re Im and if for det

The following list shows special character sequences in their input form (left column) and output form (right column).

>=	⩾
<=	⩽
==	≡
!=	≠
+-	±
->	→
<-	←
<<	≪
>>	≫
inf	∞
partial	∂
half	½
prime	′
approx	≈
nothing	
cdot	

times	×
del	∇
grad	∇
...	...
,...,	,...,
sum	Σ
int	∫
prod	Π
union	∪
inter	∩

To obtain Greek letters, simply spell them out in whatever case you want:

DELTA	Δ	iota	ι
GAMMA	Γ	kappa	κ
LAMBDA	Λ	lambda	λ
OMEGA	Ω	mu	μ
PHI	Φ	nu	ν
PI	Π	omega	ω
PSI	Ψ	omicron	ο
SIGMA	Σ	phi	φ
THETA	Θ	pi	π
UPSILON	Υ	psi	ψ
XI	Ξ	rho	ρ
alpha	α	sigma	σ
beta	β	tau	τ
chi	χ	theta	θ
delta	δ	upsilon	υ
epsilon	ε	xi	ξ
eta	η	zeta	ζ
gamma	γ		

These are all the words known to **eqn/neqn** except for the following characters with names:

above
back
bar
bold
ceiling
ccol
col
cpile
define
delim
dot
dotdot
down
dyad
fat
floor
font
from
fwd
gfont
gsize
hat
italic
lcol
left
lineup
lpile
lpile
mark
matrix
ndefine
over
pile
rcol
right
roman
rpile
size
sqrt
sub
sup
tdefine
tilde
to
under
up
vec
^
´
{ }
"..."

Troubleshooting

If you make a mistake in an equation, such as leaving out a brace (a common mistake) or typing one too many, or typing **sup** with nothing before it, **eqn/neqn** gives you an error message of the following form:

```
eqn: syntax error
file file, between lines n and n+x
```

where *n* and *n+x* are approximately the lines between which the trouble occurred; and *file* is the name of the file in question. There are also self-explanatory messages that arise in other circumstances.

If you want to check the document *file* before printing it (on the UNIX system only) type the following:

```
eqn file >/dev/null
```

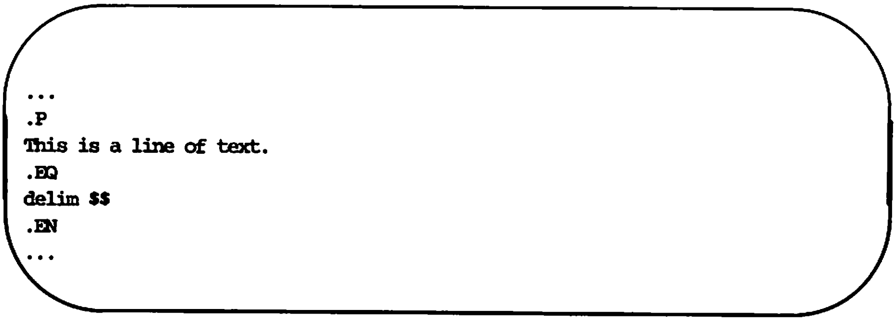
This command line throws away the output but prints any error messages.

If you use something like dollar signs as delimiters, it is easy to forget one, which causes trouble. The program **checkmm** checks for misplaced or missing dollar signs and similar troubles.

The size of in-line expressions is limited because of an internal buffer in the formatter **troff**. If you get a message "word overflow," it means you have exceeded this limit. If you print the equation as a displayed equation, this message usually goes away. The message "line overflow" signifies that you have exceeded an even bigger buffer. The only cure for this is to break the equation into two separate ones.

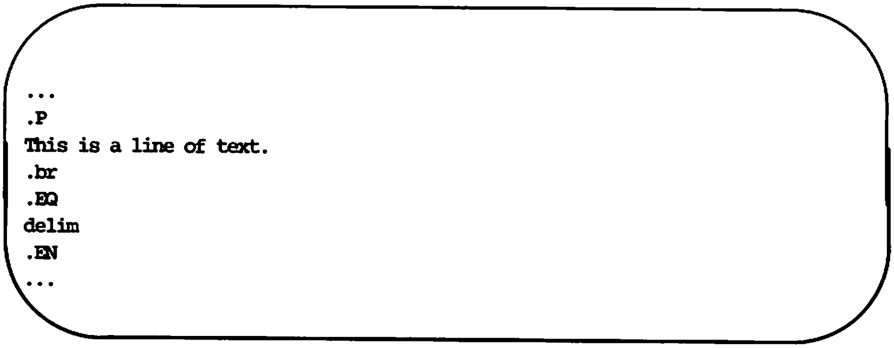
On a related topic, **eqn/neqn** does not break equations by itself; you must split long equations across multiple lines by yourself, marking each by a separate .EQ and .EN sequence. **eqn/neqn** does warn about equations that are too long to fit on one line.

When you format equations in an **mm** document, you must surround the delimiters .EQ and .EN with the **mm** display macros .DS and .DE. There is an exception to this rule; if you use .EQ and .EN only to specify the delimiters for in-line expressions or to specify **eqn/neqn** "defines" (which are explained above), do not use .DS and .DE. To do so causes extra blank lines to appear in the output. However, when you use .EQ and .EN to specify delimiters for in-line equations, one line of text before .EQ gets lost unless you take precautions. For example, suppose the document looks like this:



```
...
.P
This is a line of text.
.EQ
delim $$
.EN
...
```

"This is a line of text" gets lost because it is not flushed from the line buffer before the .EQ is encountered. One remedy is to insert the **nroff/troff** break instruction before .EQ:



```
...
.P
This is a line of text.
.br
.EQ
delim
.EN
...
```

Index

Table of Contents

Introduction

This tutorial is intended to give you a working knowledge of **pic**, the DOCUMENTER'S WORKBENCH Software tool for drawing pictures. It will also introduce you to **pic**'s programmable features. With this knowledge you will be able to draw figures and characters to be used in text. You will also learn how to implement advanced features, such as loops and conditional statements, for handling drawing tasks in sophisticated ways. Whatever uses you have for **pic**, your pictures will be fully integrated in the text formatting world of **troff** and will be treated as part of the text in large or small printing jobs.

For a brief technical discussion of the **pic** language, see the "Technical Discussion" near the back of this tutorial. For a variety of **pic** examples, see the "**pic** Sampler" immediately following the "Technical Discussion."

You should be familiar with the following concepts and tools to fully benefit from this tutorial.

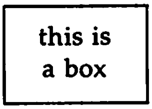
- You should know how to use a UNIX System V text editor (**ed**, **vi**, and **ex** are examples). See the *UNIX System V User Guide.*
- You should know what a file and directory are and know how to manipulate them. See the *UNIX System V User Guide.*
- You should know how to redirect input and output using pipes. See the *UNIX System V User Guide.*
- You should be familiar with a UNIX System V formatter (**nroff** or **troff**). See the tutorials in this book that discuss **nroff** or **troff**.
- Your understanding of **pic** would be assisted by a knowledge of **grap** though this is not essential. To learn about **grap**, see "The Preprocessor **grap**" in this book.

Basics

pic is a simple language for drawing pictures. Since it produces typesetter-quality text, it is used in conjunction with **troff**. You introduce pictures into your document by using macros (**.PS** and **.PE**) and by inserting, between those macros, instructions that **pic** understands. The format looks like this:

```
.PS
box "this is" "a box"
.PE
```

And the result looks like this:



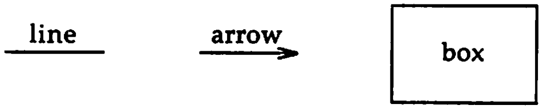
You would process the file that contains this bit of **pic** with the following command line:

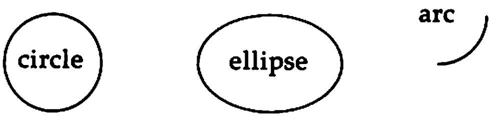
pic *file* | **troff** | *typesetter*

You need to use **troff** here rather than **nroff** because **pic** requires the free motions of digital typography to make its pictures: you, naturally, could not use a daisy-wheel printer to produce **pic**'s variegated forms.

The **.PS** and **.PE** macros (**pic** start and **pic** end) mark off and define **pic**'s workspace. In the area set off by these macros you use **pic**'s special instructions rather than **nroff** or **troff** requests and macros. This does not mean that **pic** is not an integral part of the document. On the contrary it is fully part of the **troff** environment in which you are developing your text. (In fact, **pic** makes its pictures, or forms, by automatically writing requests and escape sequences that are given to **troff** to interpret as it interprets the rest of your requests, escape sequences, and macros.) What it does mean is that **pic** has its own particular language, which is more English-oriented than are most formatting commands.

pic offers a basic set of forms that you make by simply naming them. The following illustration shows them in their default sizes:





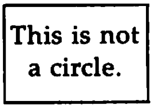
These pictures were made by naming them inside the area marked off by **.PS** and .PE. (Naturally, saying "box" or "ellipse" anywhere else in **troff** would produce merely the letters you typed and not the forms these words are able to materialize in **pic**.)

The following examples of input produced the forms given above:

```
.PS
line "line" above; move
arrow "arrow" above; move
box "box"; move
.PE
.sp 1
.PS
circle "circle"; move
ellipse "ellipse"; move
arc "arc"; move
.PE
```

This input example is divided into two **pic** groups; in between each group of two is the **troff** request, **.sp**. The point here is that, while **pic** generates requests and escape sequences for **troff** to read, it does not itself understand **troff** code. You must speak to **pic** in its native tongue; it will do the translating for **troff**. Thus, if you want to intersperse **troff** requests or escape sequences, you must first get out of the **pic** workspace for a moment with .PE.

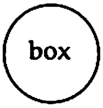
Notice also in order to place text inside forms, you must follow the form's name with the words you want, setting them off with double quotes. Each line of words must have its own set of quotes. If, for example, you wanted to place two lines of words in a box like this:



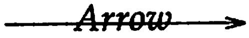
you would use the following input file:

```
.PS
box "This is not" "a circle."
.PE
```

Each set of quoted words occupies a separate line because each has its own set of quotes. Don't be fooled by these quoted words; they bear no responsibility for *making* a given form:



Many of the examples above introduce ordinary English words or text into their pictures. **pic** not only permits you to integrate these, but also allows you to adjust where they appear. While there may be times you would like the option of putting an arrow through something:



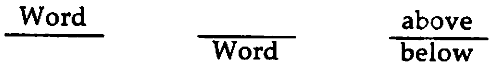
it will not often be your choice. Normally, you'll want to place words where you can read them clearly. To place a word above a line you would follow the word with **above**; a word to be placed below would be followed with **below**. The shot-through "Arrow" shown above was made with this input:

```
.PS
arrow right 1.25i "\s14\f2Arrow\f1\s0"
.PE
```

The arrow itself points to the right and is one and a quarter inches long. The word "Arrow" is specified to be in italic and given in fourteen points.

```
.PS
line "Word" above; move
line "Word" below; move
line "above" "below"
.PE
```

would give you



The third case, as you probably noticed, was not followed by either **above** or **below**. Rather it was the order in which the text was given that determined its placement.

These methods are easy and useful. But they are not the most precise methods **pic** has to offer. In succeeding pages you will learn to navigate your way around **pic**'s forms using even compass points to place text.

Note how the input examples you have seen throughout are separated from one another. Each shape has its own set of instructions. Each stands alone on its own line or is followed by semi-colons, a substitute for a new line. Even **move** must be set off by semi-colons.

move is not a shape but might be understood as one of **pic**'s invisible forms. Consider these two closely related examples. The first uses arrows, which separate boxes:

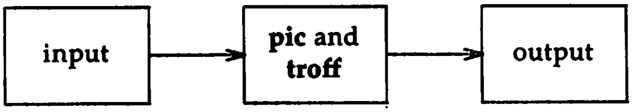
```
.PS
box "input"; arrow
box "\f3pic\f1 and" "\f3troff\f1"; arrow
box "output"
.PE
```

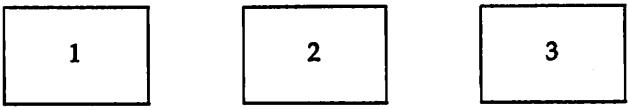
The second example uses space to separate boxes with the **move** instruction:

```
.PS
box "1"; move
box "2"; move
box "3"; move
.PE
```

Notice that both the **arrow** and the **move** instructions are set off by semicolons: they each have their separate **pic** function and need to be segregated from the **box** function.

The output of these two **pic** diagrams demonstrates that **move**, though invisible, has the same status as **arrow** or any other form:



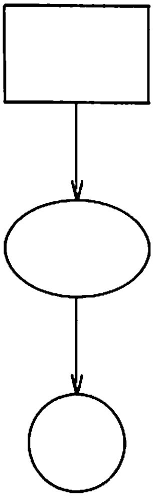


Direction of Motion

Most **pics** appear in a left-to-right fashion, but you can choose other directions. The **pic** instructions to move downward look like this:

```
.PS
down; box; arrow; ellipse; arrow; circle
.PE
```

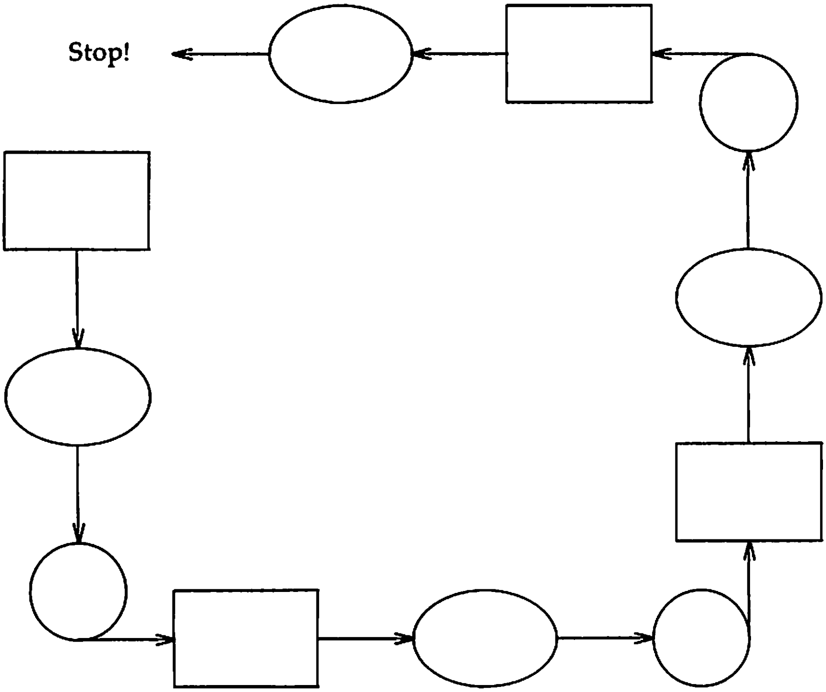
The result follows:



Once you specify a direction, the whole sequence of forms moves in that direction until you tell **pic** otherwise. The following input file includes a succession of different movements:

```
.PS
down; box; arrow; ellipse; arrow; circle
right; arrow; box; arrow; ellipse; arrow; circle
up; arrow; box; arrow; ellipse; arrow; circle
left; arrow; box; arrow; ellipse; arrow; box invis "Stop!"
.PE
```

which would produce this:



You can also specify "mini-environments" for direction of movement. If instructions are enclosed in braces, {...}, the established direction of motion inside the braces is isolated from the **pic** instructions outside. When the enclosed instructions are finished, all motion will revert to values established before the braces. Nothing else is restored.

Control of Size and Distance

The forms you have seen to this point each have been given in reasonable dimensions. **pic** tries to choose sensible measures, so that simple figures can be drawn without much bother. That is, if you don't mention what size you would like a particular shape to be, you will get default sizes. The following dimensions are understood to be in inches, **pic**'s exclusive measure:

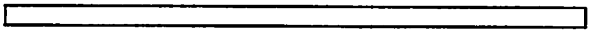
arcrad = 0.25	circlerad=0.25
arrowwid = 0.05	arrowht = 0.1 (arrowhead dimensions)
boxwid = 0.75	boxht = 0.5
ellipsewid = 0.75	ellipseht = 0.5
linewid = 0.5	lineht = 0.5
movewid = 0.5	moveht = 0.5
textwid = 0	textht = 0
dashwid = 0.05	scale = 1

These are known as "variables," and provide information to their corresponding forms. The abbreviations given here are standard **pic** terms: **wid** or **width**; **rad** or **radius**; **ht** or **height**. The abbreviation and the term it stands for may be used interchangeably in the **pic** language.

But you may want to view these dimensions as just a starting place. Each form will accept arguments to tailor its precise shape to suit your needs. For example, the input

```
.PS
box width 3 height 0.1
.PE
```

draws a long, flat box



3 inches wide and 1/10 inch high. If you forget to include the **i** (indicating inches), **pic** will infer what you mean since it uses no other measure. (If you prefer to state the **i**, however, no space may intervene between it and its corresponding number.)

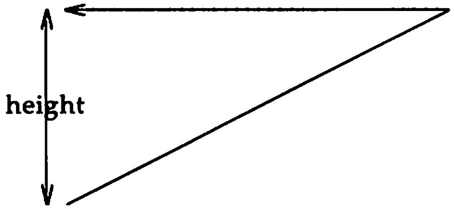
Changing one box does not change them all. The next box you ask for after this flat one will look like the ones you saw earlier. You can change the default sizes for **pic**'s forms, but that will be discussed later.

The attributes of **height** (which you can abbreviate to **ht**) and **width** (or **wid**) apply to boxes, circles, ellipses, and to the head on an arrow. The attributes of **radius** (or **rad**) and **diameter** (or **diam**) can be used for circles and arcs if they seem more natural.

The easiest way to draw lines and arrows is to state their direction and distance relative to wherever you are. The words **up, down, left** and **right** are all the terms you need to get started. You simply attach them to the form you want (**line, arrow,** or **move**) and **pic** will produce the result its language suggests. For example,

```
.PS
line up 1i right 2i; arrow left 2i
move left 0.1i; line <-> down 1i "height"
.PE
```

draws

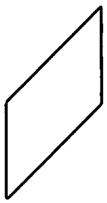


The notation <−> indicates a two-headed arrow; use −> for a head on the right end and <− for one on the left. Lines and arrows are, technically speaking, the same thing; in fact, **arrow** is a synonym for the expression, **line −>**.

If you don't put any distance after **up, down,** and so on, **pic** uses the standard distance. So

```
.PS
line up right; line down; line down left; line up
.PE
```

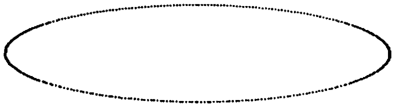
draws the parallelogram



Should you want to change whole classes of forms, you can do this by resetting the default measures assigned to the variables given above. For example, you could set **ellipsewid** to two inches if you find the standard half inch ellipse too small:

```
.PS
ellipsewid = 2
ellipse
.PE
```

would produce



You should realize, however, that by setting **ellipsewid** to two inches, you have also set all succeeding ellipses in your document to two inches. Unlike the notation,

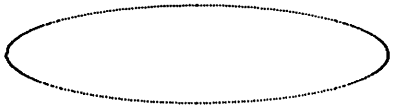
```
ellipse width 3
```

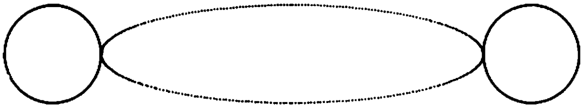
which would affect only the ellipse following it, the variable **ellipsewid** provides the width information for every ellipse in the document.

Let's make more pictures without mentioning the variable, **ellipsewid**. First, the input:

```
.PS
ellipse
.PE

.PS
circle; ellipse; circle
.PE
```

Now, the output:



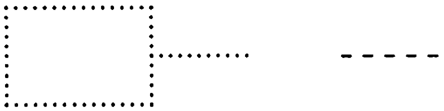


All ellipses are two inches wide while circles are unaffected. It is prudent, therefore, to reset variables as you get ready to leave the **pic** workspace:

```
.PS
ellipsewid = 3
ellipse
ellipsewid = 0.75
.PE
```

Texture

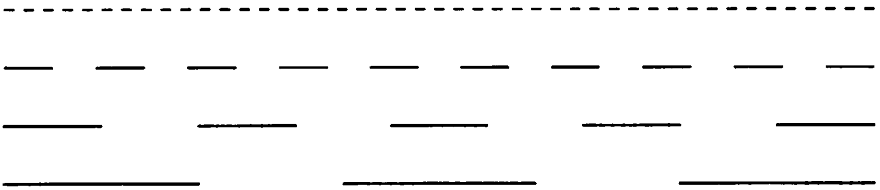
As you probably gathered from the *User's Guide*'s Sampler, boxes and lines may be dotted or dashed:



were made from

```
.PS
box dotted; line dotted; move; line dashed
.PE
```

What you may not know is that the length of these dashes and distance between dots is also subject to your control. If there is a number after **dot**, the dots will be that far apart. You can also control the size of the dashes (at least somewhat): if there is a length after the word **dashed**, the dashes will be that long, and the intervening spaces will be as close as possible to that size. So, for instance,



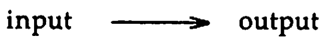
comes from the following inputs:

```
.PS
line right 4.5i dashed
.PE
.PS
line right 4.5i dashed 0.25i
.PE
.PS
line right 4.5i dashed 0.5i
.PE
.PS
line right 4.5i dashed 1i
.PE
```

Circles and arcs cannot be dotted or dashed.

Positioning Text

An interesting texture—it might be called a non-texture—is **invisible** or simply **invis**. You produce this by adding the word **invis** after the **pic** form. This is also a particularly easy and natural way to position things in a general format:



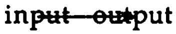
This neatly balanced figure was done with

```
.PS
box invis "input"; arrow; box invis "output"
.PE
```

pic also permits you to specify text alone without an accompanying invisible form. You must, however, retain the double quotes to distinguish text from other **pic** instructions. For example, you could have given

```
.PS
"input"; arrow; "output"
.PE
```

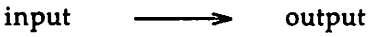
and your result would have approximated the example that used **invis**:



As you can see, this picture suffers from a problem of placement. You need to **move** the text to make it readable:

```
.PS
"input"; move; arrow; move; "output"
.PE
```

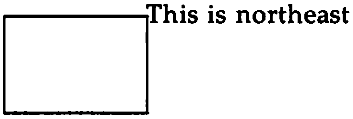
and you would get this improved result:



The issue of placement gets at the heart of **invis**'s usefulness for positioning text. Because you can name a form and not print it, you can take advantage of the form's virtues discussed in the section, "Mapping and Naming **pic**'s Forms." That is, you can specify a corner or other relative position with respect to the invisible form. The following example will not use the **invis** instruction, so you can see the demonstration more clearly:

```
.PS
A: box
"This is northeast" at A.ne ljust
.PE
```

The second line of these instructions uses a colon to associate the letter **A** with a box. The box is then referred to as **A** on the next line with the suffix **ne** (separated by a dot). This compass point tells **pic** to place the quoted string to the northeast of the named box: **at A.ne**:

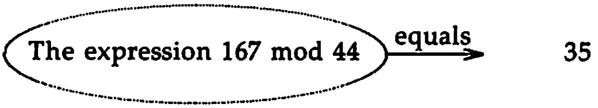


The instruction **ljust** is added to ensure that the word will be left adjusted with respect to **box** A. But this is a taste of things that are yet to come.

pic also knows arithmetic, and you can give it expressions that you would like to appear in a form. The instruction you would use is **plot**:

```
.PS
ellipsewid = 2
ellipse "The expression 167 mod 44"; arrow "equals" above; move; plot 167 % 44
ellipsewid = 0.75
.PE
```

Here is the result:



Changing Default Sizes

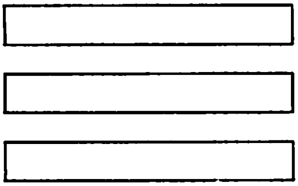
The "width" of an arrowhead is the distance between the widest part of the vee. The "height" is the distance along the shaft from back of the arrowhead to the tip.

By default, arcs move in a ninety-degree motion counterclockwise from where you are right now, and **arc cw** changes this to clockwise. The default radius is the same as for circles, but you can change it with the **rad** attribute. It is also easy to draw arcs between specific places; this will be described in the next section.

To put an arrowhead on an arc, use <–, –> or <–>

In each picture, unless an explicit dimension for some object is specified, you will get the default size. You can store non-default sizes in the **pic** reserved word, **same**. Thus, if you chose a box to be wide and shallow and wanted succeeding boxes to do the same, you would follow the word **box** with the extra information **same**. In the set of boxes given by

```
.PS
down; box ht 0.2i wid 1.5i; move down 0.15i
box same; move same; box same
.PE
```

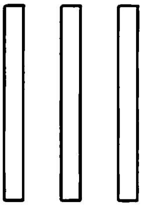


the dimensions set by the first **box** are used several times; similarly, the amount of motion for the second **move** is the same as for the first one.

It is possible to change the default sizes of objects by assigning values to certain **pic** variables. So if you want all your boxes to be long and skinny and relatively close together, for example, use the following input.

```
.PS
boxwid = 0.1i; boxht = 1i
movewid = 0.2i
box; move; box; move; box
.PE
```

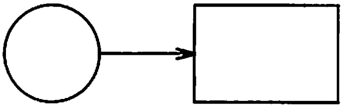
gives



You may, if you like, enter dimensions right on the **.PS** line. If you want a picture four inches wide, for example, you would follow your **.PS** with a **4**. **pic** will understand you mean inches and make the whole picture four inches wide, preserving the scale of the default picture. Let's look at the default picture first. Here is the input:

```
.PS
circle; arrow; box
.PE
```

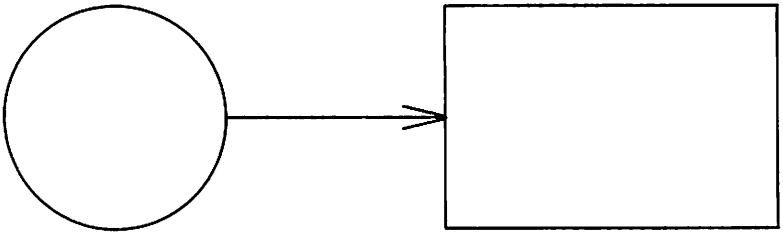
And now the output:



Next, let's see the same picture blown up to a width of four inches:

```
.PS 4
circle; arrow; box
.PE
```

will give you



pic works internally in what it thinks are inches. Setting the variable **scale** to some value causes all dimensions to be scaled down according to that value. Thus, for example, **scale=2.54** causes dimensions to be interpreted as centimeters.

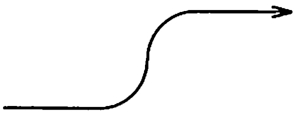
The number given as a width in the **.PS** line overrides the dimensions given in the picture; this can be used to force a picture to a particular size even when coordinates have been given in inches. Experience indicates that the easiest way to get a picture of the right size is to enter its dimensions (in inches), then if necessary add a width to the **.PS** line.

Lines and Splines

You have now seen seven **pic** forms: **line, arrow, arc, circle, ellipse, box** and **move**. One remains. You might approximate it with the knowledge you've accumulated. Input comprised of these familiar instructions:

```
.PS
line; arc; arc cw; arrow
.PE
```

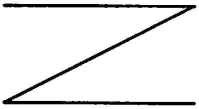
would give you an acceptable **spline**:



The **spline** tells us something useful about the **pic** language. The **pic line,** as the preceding example demonstrates, is more than a short (or even long), straight segment. It can unfold and turn before you according to your instructions:

```
.PS
line right 1i then down .5i left 1i then right 1i
.PE
```

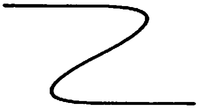
The result suggests the rambling potential of the **pic** line:



As their names suggest, **lines** and **splines** are similar to one another. Had you substituted the word **spline** for **line** in the example above:

```
.PS
spline right 1i then down .5i left 1i then right 1i
.PE
```

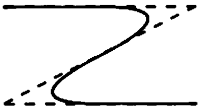
you would have gotten this:



The following overlay provides a graphic comparison between the two:

```
.PS
line dashed right 1i then down .5i left 1i then right 1i
spline from start of last line \
  right 1i then down .5i left 1i then right 1i
.PE
```

The result follows:



Long input lines can be split by ending each partial line with a backslash, escaping the newline.

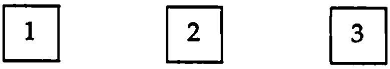
Arrowheads may only be put on the ends of a line or spline.

Controlling Positions

You can place things anywhere you want; **pic** provides a variety of ways to talk about places. As you have seen, **pic** accepts fairly straightforward instructions like **up**, **down**, **right**, and **left** with **line** and **move**. Thus

```
.PS 2                              # make picture two inches wide.
box ht 0.2 wid 0.2 at 0,0 "1"      # draw a .2" X .2" square
move to 0.5,0                      # move right 0.5
box "2" same                       # use same dimensions as last box
move same                          # use same motion as before
box "3" same
.PE
```

draws three boxes, like this:



Notice the use of **same** to repeat the previous dimensions instead of reverting to the default values. And notice the lines beginning with "#." These comment lines are used to record your intentions in case you forget what you were drawing. They are discarded during processing and will not print. They begin with a pound sign (#) and end at the end of the line.

For a more precise method of charting positions, **pic** also uses a standard Cartesian coordinate system, so any point or object has an *x* and *y* position. The first object is placed with its start at position 0,0 by default. The *x,y* position of a box, circle or ellipse is its geometrical center; the position of a line or motion is its beginning; the position of an arc is the center of the corresponding circle.

Position modifiers like **from**, **to**, **by**, and **at** are followed by an *x,y* pair and can be attached to boxes, circles, lines, motions, and so on to specify or modify a position.

Attributes like **ht** and **wid** and positions like **at** can be written out in any order. So

```
box ht 0.2 wid 0.2 at 0,0
box at 0,0 wid 0.2 ht 0.2
box ht 0.2 at 0,0 wid 0.2
```

are all equivalent, though the last is harder to read and thus less desirable.

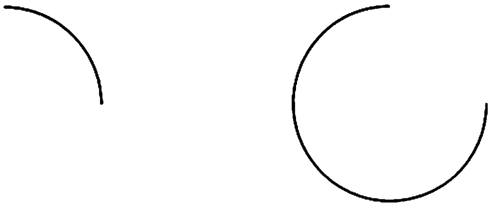
The **from** and **to** attributes are particularly useful with arcs, to specify the beginning and ending points. By default, arcs are drawn counterclockwise,

```
arc from 0.5i,0 to 0,0.5i
```

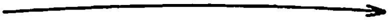
is the short arc, and

```
arc from 0,0.5i to 0.5i,0
```

is the long one:



If the **from** attribute is omitted, the arc starts where you are now and goes to the point given by **to**. The radius can be increased or diminished to make flatter or tighter arcs. To form a flat arc like this one:



you would use a radius of fifteen inches:

```
arc -> cw from 0,0 to 2i,0 rad 15i
```

You may not want this graph-paper level of precision, but it is nonetheless at your disposal. Typifying the DOCUMENTER'S WORKBENCH Software family of tools, **pic** puts as much control into your hands as would like to assume.

Mapping and Naming pic's Forms

Objects can be labelled or named so that you can talk about them later. For example,

```
.PS
Box1:
        box ...
        # ... other stuff ...
        move to Box1
.PE
```

Placenames have to begin with an upper-case letter (to distinguish them from variable names, which begin with lower-case letters). The name refers to the "center" of the object, which is the geometric center for most things. It's the beginning for lines and motions.

Other combinations also work:

```
.PS
line from Box1 to Box2
move to Box1 up 0.1 right 0.2
move to Box1 + 0.2,0.1      # same as previous
line to Box1 - 0.5,0
.PE
```

The reserved name **Here** may be used to record the current position of some object, for example as

```
        Box1:  Here
```

Labels can function as variables—they can be reset several times in a single picture, so a line of the form

```
Box1: Box1 + 1i,1i
```

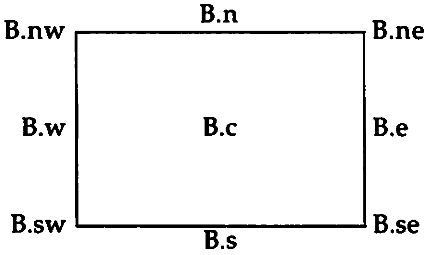
is perfectly legal.

You can also refer to previously drawn objects of each type, using the word **last**. For example, if we began with the following input:

```
box "A"; circle "B"; box "C"
```

then **last box** refers to box C, **last circle** refers to circle B, and **2nd last box** refers to box A. Numbering of objects can also be done from the beginning, so boxes A and C are **1st box** and **2nd box** respectively.

To cut down the need for explicit coordinates, most objects can be described using compass points:



The primary compass points may also be written as **.r**, **.b**, **.l**, and **.t**, for **right, bottom, left**, and **top**. The box above was produced with

```
.PS 1.5
B: box "B.c"
" B.e" at B.e ljust
" B.ne" at B.ne ljust
" B.se" at B.se ljust
"B.s" at B.s below
"B.n" at B.n above
"B.sw " at B.sw rjust
"B.w " at B.w rjust
"B.nw " at B.nw rjust

.PE
```

Note the use of **ljust, rjust, above,** and **below** to adjust the positioning of the inserted text, and note the blank space used within the double quotes to move text away from the box's vertical lines.

Normally, text is centered at the geometric center of the object with which it is associated. The attribute **ljust** causes the left end to be at the specified point (which means that the text lies to the *right* of the specified place), and **rjust** puts the right end at the place. **above** and **below** center the text one half line space in the given direction.

You may not combine text attributes. It is illegal, for instance, to say "*text*" **above ljust.**

Text is most often an attribute of some other object, but you can also have self-standing text:

```
"this is some text" at 1,2 ljust
```

Lines and arrows have a **start,** an **end,** and a **center** in addition to corners. There are many ways to talk about the corners of an object. Besides the compass points, almost any sensible combination of **left, right, top, bottom, upper,** and **lower** will work. Furthermore, if you don't like the shorthand notation of compass points, as in

```
last box.ne
```

you can instead say

```
upper right of last box
```

A longer statement like

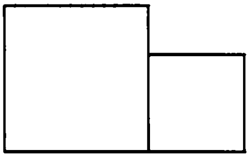
```
.PS
...
line from upper left of 2nd last box to bottom of 3rd last ellipse
...
.PE
```

begins to wear after a while, but it is descriptive.

It is sometimes easiest to position objects by positioning some part of one at some part of another, for example the northwest corner of one at the southeast corner of another. The **with** attribute in **pic** permits this kind of positioning. For example,

```
.PS
box ht 0.75i wid 0.75i
box ht 0.5i wid 0.5i with .sw at last box.se
.PE
```

produces

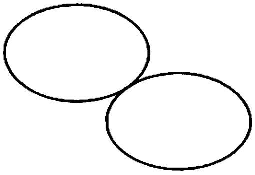


Notice that the corner after **with** is written .sw

As another example, consider

```
.PS
ellipse; ellipse with .nw at last ellipse.se
.PE
```

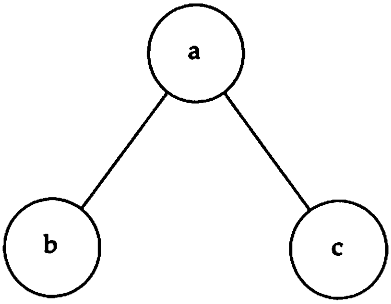
which makes



Sometimes it is desirable to have a line intersect a circle at a point which is not one of the eight compass points that **pic** knows about. In such cases, the proper visual effect can be obtained by using the attribute **chop** to chop off part of the line.

```
.PS
circle "a"
circle "b" at 1st circle - (0.75i, 1i)
circle "c" at 1st circle + (0.75i, -1i)
line from 1st circle to 2nd circle chop
line from 1st circle to 3rd circle chop
.PE
```

produces



By default the line is chopped by **circlerad** at each end. This may be changed:

```
line ... chop r
```

chops both ends by **r**, and

```
line ... chop r1 chop r2
```

chops the beginning by **r1** and the end by **r2**

There is one other form of positioning that is sometimes useful, to refer to a point some fraction of the way between two other points. This can be expressed in **pic** as

fraction of the way between **position1** and **position2**

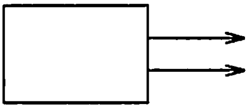
fraction is any expression, and **position1** and **position2** are any positions. You can abbreviate this phrase; "of the way" is optional, and the whole thing can be written instead as

fraction < position1 , position2 >

As an example,

```
.PS
box
arrow right from 1/3 of the way between last box.ne and last box.se
arrow right from 2/3 <last box.ne, last box.se>
.PE
```

produces



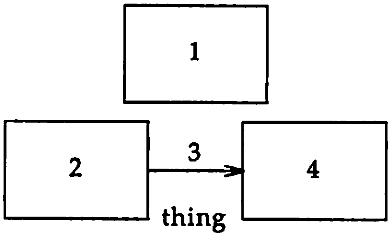
Naturally, the distance given by **fraction** can be greater than 1 or less than 0.

Blocks

Any sequence of **pic** statements may be enclosed in brackets [...] to form a block, which can then be treated as a single object and manipulated rather like an ordinary box. For example, if we say

```
.PS
box "1"
[ box "2"; arrow "3" above; box "4" ] with .n at last box.s - (0,0.1)
"thing" at last [].s
.PE
```

we get



Notice that **last**-type constructs treat blocks as a unit and don't look inside for objects: **last box.s** refers to box 1, not box 2 or 4. You can use **last []**, etc., just like **last box** .

Blocks have the same compass corners as boxes (determined by the bounding box). It is also possible to position a block by placing either an absolute coordinate (like **0,0**) or an internal label (like **A**) at some external point, as in

```
[ ...; A: ...; ... ] with .A at ...
```

Blocks join with other things like boxes do (i.e., at the center of the appropriate side).

Names of variables and places within a block are local to that block, and thus do not affect variables and places of the same name outside. You can get at the internal place names with constructs like

```
last [].A
```

or

```
B.A
```

where **B** is a name attached to a block like so:

```
B : [ ... ;  A: ...;  ]
```

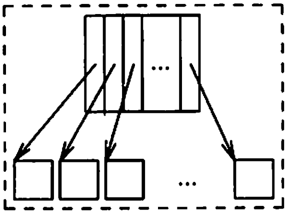
When combined with **define** statements (next section), blocks provide a reasonable simulation of a procedure mechanism.

Although blocks nest, it is possible to look only one level deep with constructs like **B.A** although **A** may be further qualified (i.e., **B.A.sw** or **top of B.A** are legal).

The following example illustrates most of the points made above about how blocks work.

```
.PS
h = .5i
dh = .02i
dw = .1i
[
        Ptr: [
                boxht = h; boxwid = dw
                A: box
                B: box
                C: box
                box wid 2*boxwid "..."
                D: box
        ]
        Block: [
                boxht = 2*dw; boxwid = 2*dw
                movewid = 2*dh
                A: box; move
                B: box; move
                C: box; move
                box invis "..." wid 2*boxwid; move
                D: box
        ] with .t at Ptr.s - (0,h/2)
        arrow from Ptr.A to Block.A.nw
        arrow from Ptr.B to Block.B.nw
        arrow from Ptr.C to Block.C.nw
        arrow from Ptr.D to Block.D.nw
]
box dashed ht last [].ht+dw wid last [].wid+dw at last []
.PE
```

This produces



Variables and Expressions

It's generally a bad idea to write everything in absolute coordinates if you are likely to change things. Different size pages are likely to make you wish for proportional measures. **pic** variables enable you to change the pictures you make without much ado.

```
.PS
a = 0.5;  b = 1
box wid a ht b
ellipse wid a/2 ht 1.5*b
move to Box1 - (a/2, b/2)
.PE
```

Expressions may use the standard arithmetic operators +, −, *, /, and % for adding, subtracting, multiplying, dividing, or determining a modulus. Parentheses may be used to group operations. The logical operators ==, !=, >, <, ≥, ≤, &&, and |are allowed as well as the assignment operator, =.

Probably the most important variables are the predefined ones for controlling the default sizes of objects. These may be set at any time in any picture and retain their values until reset.

You can use the height, width, radius, and *x* and *y* coordinates of any object or corner in an expression:

```
.PS
Box1.x                    # the x coordinate of Box1
Box1.ne.y                 # the y coordinate of the NE corner of Box1
Box1.wid                  # the width of Box1
Box1.ht                   # and its height
2nd last circle.rad       # the radius of the 2nd last circle
.PE
```

Any pair of expressions enclosed in parentheses defines a position; furthermore, such positions can be added or subtracted to yield new positions:

(x , y)

$(x_1,y_1)+(x_2,y_2)$

If p_1 and p_2 are positions, then (p_1,p_2) refers to the point

Macros

pic provides a rudimentary macro facility, the simple form of which is identical to that in **eqn**:

```
    define  name  X replacement text X
```

defines *name* to be the *replacement text;* X or any character that does not appear in the replacement may be used as a delimiting character. Any subsequent occurrence of *name* will be replaced by *replacement text.*

Macros with arguments are also available. The replacement text of a macro definition may contain occurrences of **$1** through **$9** ; these will be replaced by the corresponding actual arguments when the macro is invoked. The invocation for a macro with arguments is

```
    name(arg1, arg2, ...)
```

Non-existent arguments are replaced by null strings.

As an example, one might define a *square* by

```
.PS
define square X box ht $1 wid $1  $2 X
...
.PE
```

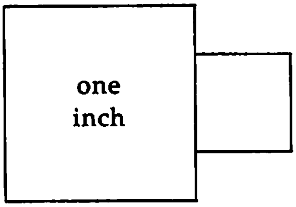
Then

```
.PS
...
square(1i, "one" "inch")
...
.PE
```

calls for a one-inch square with the obvious label, and

```
.PS
...
square(1i, "one" "inch")
...
.PE
```

calls for a square with no label:



Coordinates like x,y may be enclosed in parentheses, as in (x,y), so they can be included in a macro argument.

copy and copy...thru Facilities

pic offers a facility similar to macros in that it allows you to take input from a remote source. You use this facility with the instruction **copy**, which copies data from the file you give it as an argument. Here's how it works:

```
.PS
...
copy "file"
...
.PE
```

When **pic** gets to the line beginning **copy**, it reads the contents of the *file* for its data. It ignores .PS's and .PE's in the remote file, so you can incorporate smaller, already drawn pictures into larger ones.

A slight refinement of this usage is the

```
copy "file" thru macro-name
```

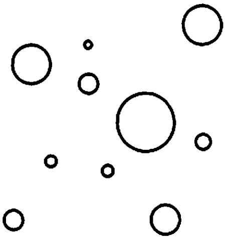
copies *file*, treating each line as an invocation of the named macro (each field being an argument). A literal macro may be used instead of a name:

```
copy "file" thru X macro replacement text X
```

and if no file name is given, the remainder of the input until the next .PE is used. So to plot a set of circles at points whose coordinates and radii come from a file:

```
.PS
copy thru / circle rad $3 at $1,$2 /
0 0 .05
1 1 .1
...
.PE
```

Here are the results:



The **sh** command executes an arbitrary UNIX system command line:

```
.PS
sh X anything X
...
.PE
```

as with the macro facility, **X** is any character not in *anything*..

A last touch to the **thru** instruction is **until**:

```
.PS
copy thru macro until "string"
...
.PE
```

or

```
copy file thru macro until "string"
...
.PE
```

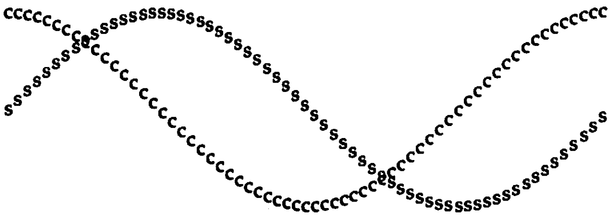
That is, you are able to specify how much of the remote macro or remote file you want to read as input. When **pic** sees *string,* whatever you specify it to be, **pic** ceases to use *macro* or *file* as input.

Loops and Conditional Statements

pic provides an **if** statement and a **for** loop:

```
.PS
.ps -2
pi = atan2(0,-1)                    # atan2 is a pic expression.
for i = 0 to 2 * pi by 0.1 do X
                                    "s" at i, sin(i)
                                    "c" at i, cos(i)
X
.ps +2
.PE
```

The processed output follows:



The body of the loop is delimited by any character not found within it. The **by** clause is optional; values may be preceded by any of the four basic arithmetic operators:

+	(addition)
−	(subtraction)
*	(multiplication)
/	(division)

The **if** statement is

`if` *expression* `then X` *anything* `X else X` *anything* `X`

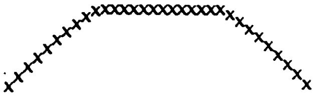
where the **else** clause is optional. The *expression* may use the usual relational operators:

==	(equal to)
!=	(not equal to)
>	(greater than)
>=	(greater than or equal to)
<	(less than)
<=	(less than or equal to)
&&	(and)
|	(or)
!	(not)

For example, the following **pic** file:

```
.PS
for i = 0 to pi by 0.1 do X
                                        s = sin(i)
                                        if s > 0.8 then Y s = 0.8 Y
                                        "x" at i/2, s/2
X
.PE
```

produces the following:



A string comparison using == or != is also permitted:

`if` *string1* `==` *string2* `then ...`

Technical Discussion

pic is a **troff** preprocessor. The command line you use to run it reflects this relationship with **troff**:

> **pic** *file* | **troff −mm** | *typesetter*

If **eqn** is also present, make sure **eqn** follows **pic** in the order of processing:

> **pic** *file* | **eqn** | **troff −mm** | *typesetter*

pic copies the **.PS** and **.PE** lines from input to output intact, except that it adds two things on the same line as the **.PS**. Type:

> **.PS w h**

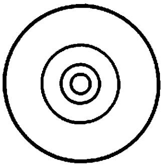
in order to specify **h** and **w** as the picture's height and width in inches.

If **.PF** is used instead of **.PE**, the position after printing is restored to where it was before the picture started, instead of being at the bottom. (**F** is for "flyback.")

Any input line that begins with a period is assumed to be a **troff** request or macro and is interpreted at that point. Adding vertical space (\v) or extra spaces (**.sp**) will probably affect the outcome of your pictures. Point size and font changes are generally harmless, though. Consider the following file:

```
.PS
.ps 24
circle radius .4 at 0,0
.ps 12
circle radius .2 at 0,0
.ps 8
circle radius .1 at 0,0
.ps 6
circle radius .05 at 0,0
.ps 10                              \" don't forget to restore point size
.PE
```

which produces this picture:

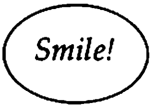


pic does preserve the state of **troff**'s fill mode across pictures.

It is also safe to include sizes, fonts and local motions within quoted strings ("...") in **pic**, so long as whatever changes are made are unmade before exiting the string. For example, to print text in italic two points larger:

```
.PS
ellipse "\s+2\fISmile!\fP\s-2"
.PE
```

results in the following:

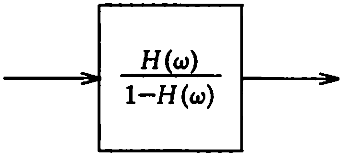


This is essentially the same rule as applies in **eqn**.

There is a subtle problem with complicated equations inside **pic** pictures — they come out wrong if **eqn** has to leave extra vertical space for the equation. If your equation involves more than subscripts and superscripts, you must add to the beginning of each equation the extra information "**space 0**". The following file:

```
.PS
boxht=0.75
arrow
box "$space 0 {H( omega )} over {1 - H( omega )}$"
arrow
boxht=0.5
.PE
```

produces this:



pic output is specified in inches and is, therefore, independent of any particular typesetter. In the rare case that you have to specify your typesetter, use the −T option as in

pic −Taps ...

for the Autologic APS-5 phototypesetter.

Pictures

The top-level object in **pic** is the "picture":

picture:

.PS *optional-width optional-height*
element-list
.PE

If *optional-width* is present, the picture is made that many inches wide, regardless of any dimensions used internally. The height is scaled in the same proportion unless *optional-height* is present.

If .PF is used instead of .PE, the position after printing is restored to what it was upon entry.

Elements

An *element-list* is a list of elements; the elements are

shape attribute-list
placename : *element*
placename : *position*
variable = *expression*
direction
{ *list of elements* }
[*list of elements*]
for statement
if statement
copy statement
print statement
plot statement
sh X *commandline* X
troff-command

Specify a *placename* with a capital letter
followed by zero or more letters or numbers.

Specify a *variable* with a letter
followed by zero or more letters or numbers.

Elements in a list must be separated by newlines or semicolons; a long element may be continued by ending the line with a backslash. Comments are introduced by a # and terminated by a newline.

Variable names begin with a lower case letter; place names begin with upper case. Place and variable names retain their values from one picture to the next.

The current position and direction of motion are saved upon entry to a {...} block and restored upon exit.

Elements within a block enclosed in **[...]** are treated as a unit; the dimensions are determined by the extreme points of the contained objects. Names, variables, and direction of motion within a block arė local to that block.

troff-command is any line that begins with a period. Such lines are assumed to make sense in the context where they appear; accordingly, if it doesn't work, don't call.

Primitives

The primitive objects are

```
primitive:
        box
        circle
        ellipse
        arc
        line
        arrow
        spline
        move
        text-list
```

arrow is a synonym for **line ->**.

Attributes

An *attribute-list* is a sequence of zero or more attributes; each attribute consists of a keyword, perhaps followed by a value. In the following, *e* is an expression and *opt-e* an optional expression.

attribute:

h(eigh)t *e*	wid(th) *e*
rad(ius) *e*	diam(eter) *e*
up *opt-e*	down *opt-e*
right *opt-e*	left *opt-e*
from *position*	to *position*
at *position*	with *corner*
by *e, e*	then
dotted *opt-e*	dashed *opt-e*
chop *opt—e*	-> <- <->
invis	same
text-list	

Missing attributes and values are filled in from defaults. Not all attributes make sense for all primitives; irrelevant ones are silently ignored. These are the currently meaningful attributes:

box:
    height, width, at, same, dotted, dashed, invis, *text*
circle, ellipse:
    radius, diameter, height, width, at, same, invis, *text*
arc:
    up, down, left, right, height, width, from, to, at, radius,
    invis, cw, <-, ->, <->, *text*
line, arrow
    up, down, left, right, height, width, from, to, by, then, at,
    same, dotted, dashed, invis, <-, ->, <->, *text*
spline:
    up, down, left, right, height, width, from, to, by, then, at,
    same, invis, <-, ->, <->, *text*
move:
    up, down, left, right, to, by, same, *text*
text-list:
    at, *text-item*

The attribute **at** implies placing the geometrical center at the specified place. For lines, splines and arcs, **height** and **width** refer to arrowhead size.

Text

Text is normally an attribute of some primitive; by default it is placed at the geometrical center of the object. Stand-alone text is also permitted. A *text-list* is a list of text items; a text item is a quoted string optionally followed by a positioning request:

text-item:
"..."
"..." center
"..." ljust
"..." rjust
"..." above
"..." below

If there are multiple text items for some primitive, they are centered vertically except as qualified. Positioning requests apply to each item independently.

Text items can contain **troff** commands for size and font changes, local motions, etc., but make sure that these are balanced so that the entering state is restored before exiting.

Positions and Places

A position is ultimately an *x,y* coordinate pair, but it may be specified in other ways.

position:
place
(*position*)
expression, expression
(*position*) [± *(expression, expression)*]
(*position*) [± *expression, expression*]
(*place1, place2*), i.e., (*place1*.x, *place2*.y)
expression < *position* , *position* >
*expression [*of the way*]* between *position* and *position*

place:

placename [*corner*]
corner placename
Here
corner of *nth shape*
nth shape [*corner*]

A *corner* is one of the eight compass points or the center or the start or end of a primitive.

corner:

.n .e .w .s .ne .se .nw .sw
.t .b .r .l
.c .start .end

Each object in a picture has an ordinal number; *nth* refers to this.

nth:

*n*th
*n*th last

Since **pic** is flexible enough to accept names like **1th** and **3th**, synonyms like **1st** and **3st** are accepted as well.

Variables

The built-in variables and their default values are:

arcrad = 0.25	circlerad=0.25
arrowwid = 0.05	arrowht = 0.1 (arrowhead dimensions)
boxwid = 0.75	boxht = 0.5
ellipsewid = 0.75	ellipseht = 0.5
linewid = 0.5	lineht = 0.5
movewid = 0.5	moveht = 0.5
textwid = 0	textht = 0
dashwid = 0.05	scale = 1

These may be changed at any time, and the new values remain in force from picture to picture until changed again.

The variable **textht** and **textwid** may be set to any values to control positioning. The width and height of the generated picture may be set independently from the **.PS** line. Variables changed within "[" and "]" revert to their previous value upon exit from the block. Dimensions are divided by **scale** during output.

Expressions

Expressions in **pic** are evaluated in floating point. All numbers representing dimensions are taken to be in inches.

expression:

e + *e*
e − *e*
e * *e*
e / *e*
e % *e* (modulus)
− *e*
(*e*)
variable
number
place .x
place .y
place .ht
place .wid
place .rad
sin(*e*) cos(*e*) atan2(*e*,*e*) log(*e*) sqrt(*e*) int(*e*)
max(*e*,*e*) min(*e*,*e*) rand(*e*)

Logical Operators

pic provides the following operators for logical evaluation:

!	(not)
>	(greater than)
<	(less than)
⩾	(greater than or equal to)
⩽	(less than or equal to)
&&	(and)
|	(or)
==	(equal to)
!=	(not equal to)

Definitions

The **define** statement is not part of the grammar.

define:

define *name* X *replacement text* X

Occurrences of **$1, $2,** etc., in *replacement text* will be replaced by the corresponding arguments if **name** is invoked as

name(*arg1*, *arg2*, ...)

Non-existent arguments are replaced by null strings. *Replacement text* may contain newlines.

copy and copy...thru Statements

The **copy** statement includes data from a remote file or data immediately following **copy** in the **pic** file:

copy "*file*"
copy thru *macro*
copy "*file*" thru *macro*
copy "*file*" thru *macro* until "*string*"

The *macro* may be either the name of a defined macro, or the body of a macro enclosed in some character not part of the body. If no file is given, **copy** copies the input until the next .PE.

for Loops and if Statements

The **for** and **if** statements provide for loops and decision-making:

```
for var=expr to expr by expr do X anything X
if expr then X anything X else X anything X
```

The **by** and **else** clauses are optional. The *expr* in an **if** may use the usual relational operators or the string tests *str1* == (or !=) *str2.*

Miscellany

The **sh** command executes a command line:

```
sh X commandline X
```

It is possible to plot the value of an expression:

```
plot expr opt-format attributes
```

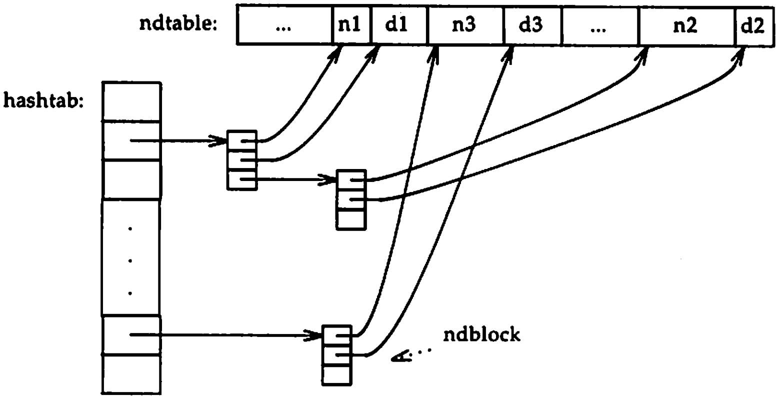
The *expr* is evaluated and converted to a string (using the format specification if provided).

The state of fill or no-fill mode is preserved around a picture.

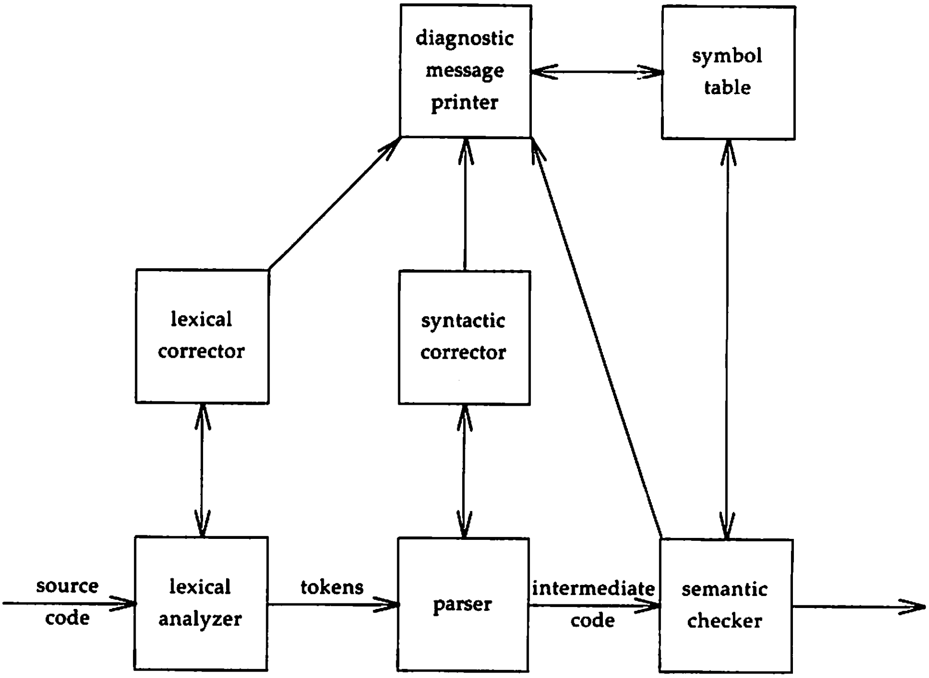
Input numbers may be expressed in E notation.

pic Sampler

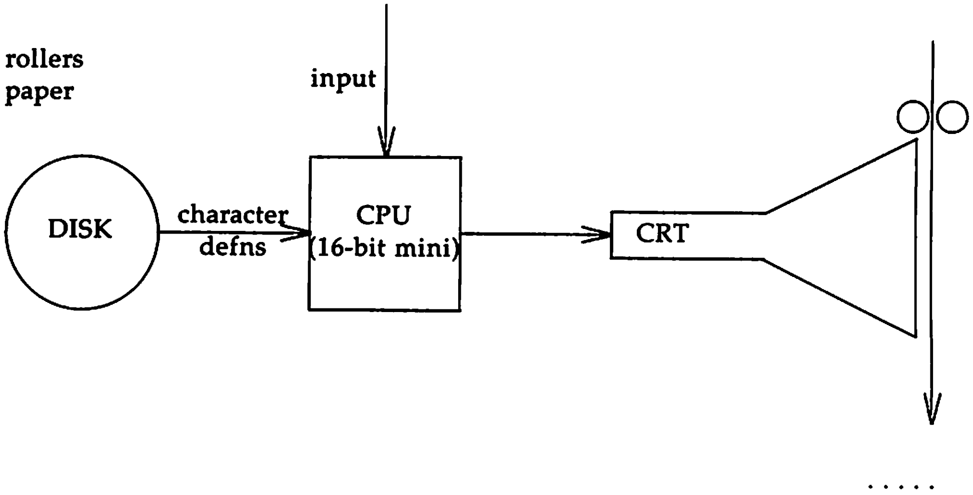
```
.PS
define ndblock X
                  box wid boxwid/2 ht boxht/2
                  down;  box same with .t at bottom of last box;   box same
X
boxht = .2i; boxwid = .3i; circlerad = .3i
down; box; box; box; box ht 3*boxht "." "." "."
L: box; box; box invis wid 2*boxwid "hashtab:" with .e at 1st box .w
right
Start: box wid .5i with .sw at 1st box.ne + (.4i,.2i) "..."
N1: box wid .2i "n1";  D1: box wid .3i "d1"
N3: box wid .4i "n3";  D3: box wid .3i "d3"
box wid .4i "..."
N2: box wid .5i "n2";  D2: box wid .2i "d2"
arrow right from 2nd box
ndblock
spline -> right .2i from 3rd last box then to N1.sw + (0.05i,0)
spline -> right .3i from 2nd last box then to D1.sw + (0.05i,0)
arrow right from last box
ndblock
spline -> right .2i from 3rd last box to N2.sw-(0.05i,.2i) to N2.sw+(0.05i,0)
spline -> right .3i from 2nd last box to D2.sw-(0.05i,.2i) to D2.sw+(0.05i,0)
arrow right 2*linewid from L
ndblock
spline -> right .2i from 3rd last box to N3.sw + (0.05i,0)
spline -> right .3i from 2nd last box to D3.sw + (0.05i,0)
circle invis "ndblock"  at last box.e + (.7i,.2i)
arrow dotted from last circle to last box chop
box invis wid 2*boxwid "ndtable:" with .e at Start.w
.PE
```


ndtable:
...
n1
d1
n3
d3
...
n2
d2
hashtab:
ndblock

```
.PS 4.8
.ps 8
boxht =.5
boxwid =.5
circlerad = .25
                    arrow "source" "code"
LA:                 box "lexical" "analyzer"
                    arrow "tokens" above
P:                  box "parser"
                    arrow "intermediate" "code"
Sem:                box "semantic" "checker"
                    arrow

                    arrow <-> up from top of LA
LC:                 box "lexical" "corrector"
                    arrow <-> up from top of P
Syn:                box "syntactic" "corrector"
                    arrow up
DMP:                box "diagnostic" "message" "printer"
                    arrow <-> right  from right of DMP
ST:                 box "symbol" "table"
                    arrow from LC.ne to DMP.sw
                    arrow from Sem.nw to DMP.se
                    arrow <-> from Sem.top to ST.bot
.ps 10
.PE
```


diagnostic message printer
symbol table
lexical corrector
syntactic corrector
source code
lexical analyzer
tokens
parser
intermediate code
semantic checker

```
.PS 5
circle "DISK"
arrow "character" "defns"
box "CPU" "(16-bit mini)"
{ arrow <- from top of last box up "input " rjust }
arrow
CRT: "   CRT" ljust
line from CRT - 0,0.075 up 0.15 \
then right 0.5 \
then right 0.5 up 0.25 \
then down 0.5+0.15 \
then left 0.5 up 0.25 \
then left 0.5

Paper: CRT + 1.0+0.05,0
arrow from Paper + 0,0.75 to Paper - 0,0.5
{ move to start of last arrow down 0.25
  { move left 0.015; circle rad 0.05 }
  { move right 0.015; circle rad 0.05; "   rollers" ljust }
}
"  paper" ljust at end of last arrow right 0.25 up 0.25
line left 0.2 dotted
.PE
```


rollers
paper
input
DISK
character
defns
CPU
(16-bit mini)
CRT
. . . . .

Index

Table of Contents

Introduction

This tutorial is intended to give you a working knowledge of **grap**, the DOCUMENTER'S WORKBENCH Software tool for making graphs. It will also introduce you to **grap**'s programmable features. With this knowledge you will be able to represent statistics in graphs that can be automatic or manipulated using programming features. You will also learn how to implement features, such as loops and conditional statements, for handling information in sophisticated ways.

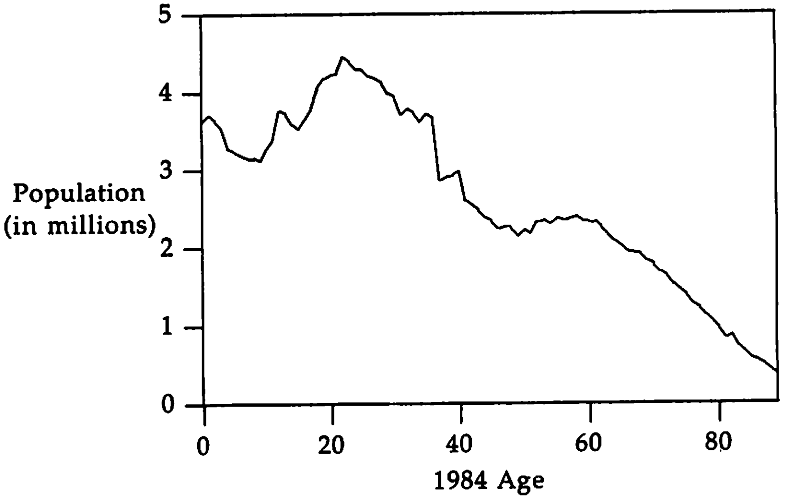
You should be familiar with the following concepts and tools to benefit fully from this tutorial.

- You should know how to use a UNIX System V text editor (**ed**, **vi**, and **ex** are examples). See the *UNIX System V User Guide*.
- You should know what a file and directory are and know how to manipulate them. See the *UNIX System V User Guide*.
- You should know how to redirect input and output using pipes. See the *UNIX System V User Guide*.
- You should be familiar with a UNIX System V formatter (**nroff** or **troff**). See the tutorials in this book that discuss **nroff** or **troff**.
- Your understanding of **grap** would be assisted by a knowledge of **pic** though this is not essential. To learn about **pic**, see "The Preprocessor **pic**: A Tutorial" in this book.

Basics

To chart lists of numbers, **grap** is largely automatic. It will take a list of figures with no further information and draw a box, draw tick marks that represent the range of values suggested by the numbers, and plot the numbers according to their values. For more detailed tasks, its requirements are modest. In the workarea between its **.G1** and **.G2**, it accepts special formatting instructions stating the size of the graph (though **grap** will give you a default graph size of two inches high by three inches wide), for specifying tick mark intervals (though **grap** will infer these from your numbers), for placing labels along axes, and for providing other details concerning the graph's presentation. It then reads a list of numbers and plots them in conformance with the specifications you have stated. While **grap**, like **pic**, offers sophisticated features, such as proportional scaling, macros, a **copy...thru** facility, a looping facility, and the ability to evaluate conditional statements, it is also a concise and easy-to-learn language. The following example suggests its streamlined form.

This graph of the 1984 age distribution in the United States



was produced by these **grap** instructions:

```
.G1
coord x 0,89 y 0,5
label left "Population" "(in millions)"
label bottom "1984 Age"
draw solid
copy "pop.cp"
.G2
```

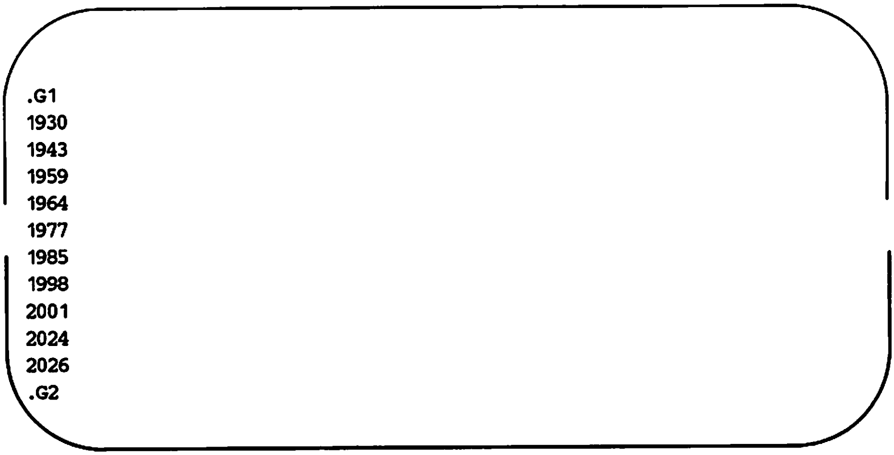
The second line, giving coordinates, dictates the graph's range of tick marks along the *x* and *y* axes. The next two lines, beginning **label**, allow for the introduction of text to explain the graph. Notice the language you use to enter labels: **left** side, **bottom** of picture, etc. The next line calls for a **solid** line to be drawn, charting the peaks and valleys of U. S. age distribution. Omitting **draw solid** would have given you scattered points for each age group. Finally, **copy** asks that **grap** look for its numbers in a file called **pop.cp**. The file contains two simple columns: 1) an ordered numerical sequence of ascending ages from new-born to eighty-nine and 2) the number of living Americans in these age groups.

The **grap** preprocessor works in concert with **pic** and **troff** as the following typical **grap** command line illustrates:

grap *file* | **pic** | **troff −mm** | *typesetter*

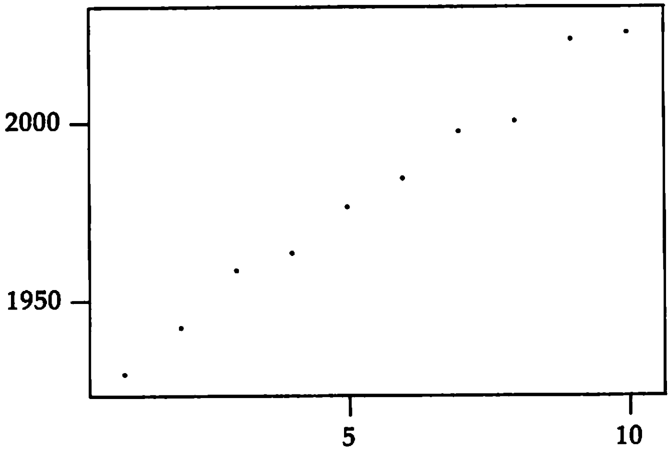
grap converts all instructions between **.G1** and **.G2** into **pic** instructions; **pic** converts them into input for **troff**; and **troff** processes them along with the remainder of the file's text and formatting commands.

Let's see what **grap** does with a simple list of dates. The following is a list of ten different years:



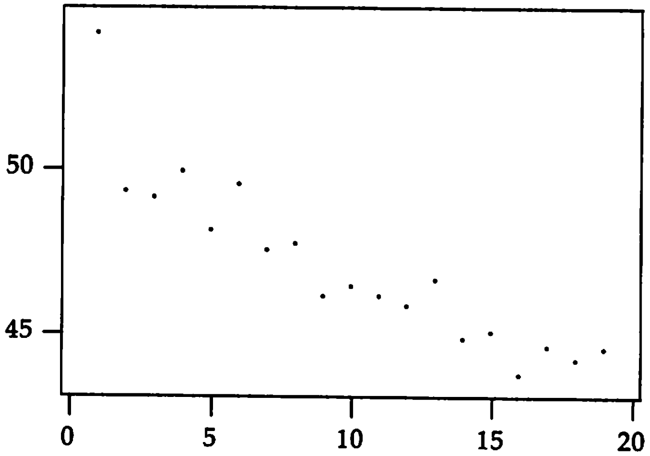
```
.G1
1930
1943
1959
1964
1977
1985
1998
2001
2024
2026
.G2
```

Given to **grap**, the data is interpreted into the following graph:



grap examined the figures and determined two salient features before labeling the *x* and *y* axes. First, it counted the number of items, and second, it found the range of values expressed by the dates. In each case, **grap** decided upon sensible intervals with which to mark the information. The dates themselves are given as a series of scattered plotting marks.

The following **grap** example is more fast-paced, graphically displaying the winning times of Olympic runners. Here's the graph:



It demonstrates the generally decreasing winning times from 54.2 seconds to 44.60 seconds. Here's how it was done:

```
.G1
54.2
49.4
49.2
50.0
48.2
...
44.60
.G2
```

The single number on each line is the winning time in seconds for the men's 400 meter run, from the first modern Olympic Games (1896) to the nineteenth (1980). This file of winning times, **olymp.g**, would be processed as follows:

grap olymp.g | pic | troff | *typesetter*

This tutorial will frequently give only the first five lines and the last line of data. Omitted lines are indicated by "...".

There are a couple of things especially worth noticing in the **olymp.g** file. First, the **coord** line, present in the first example, is missing here. **grap** is able to adjust its presentation of your numbers automatically into a reasonably proportioned graph. What is more, it automatically provides the ticks along its axes, inferring them from your numbers. Second, without the **draw solid** instructions, the graph is presented using scattered points rather than a climbing and falling line.

If the winning times represented above were contained in the file **times.cp**, you could produce the same graph with the program:

```
.G1
copy "times.cp"
.G2
```

That is, **copy "times.cp"** is equivalent to including the data itself between the **.G1** and **.G2** macros. **copy** requires the file's name to be surrounded by double quotes (just as **label** requires its labels to be in double quotes.)

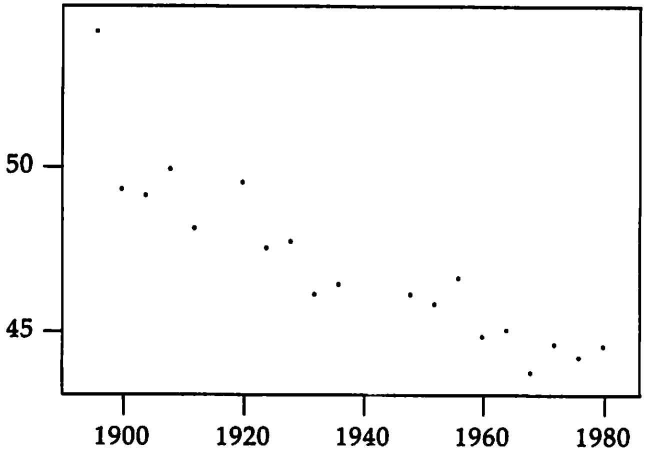
A variant of the file **times.cp** is the following file, **mtimes.cp**. This file contains two columns. In effect the first column (the Olympic year) pushed over the second column (the winning times):

```
1896 54.2
1900 49.4
1904 49.2
1908 50.0
1912 48.2
...
1980 44.6
```

If you plot these data with the following program:

```
.G1
copy "mtimes.cp"
.G2
```

you produce the following graph:



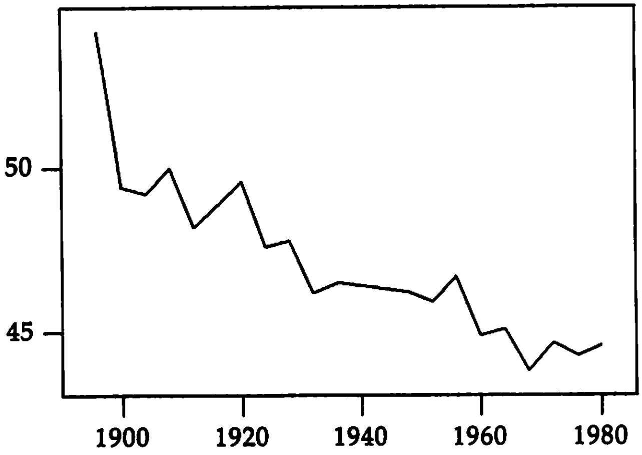
The file's first column, as you can see, gives the information that will be spelled out along the x axis. Because no Olympic games were held during wartime (1916, 1940, 1944), no data for these years are included in **mtimes.cp**. This is reflected in the graph by an absence of points for these x-axis values.

Because the previous data (in **times.cp**) had just one number per line, **grap** viewed it as a time series and supplied x-values of 1, 2, 3, ... before plotting the data as y-values. The input to the second program has two values per line, so they are interpreted as (x, y) pairs.

Rather than a scatter plot of points, you might prefer to see the winning times connected by a solid line. The program

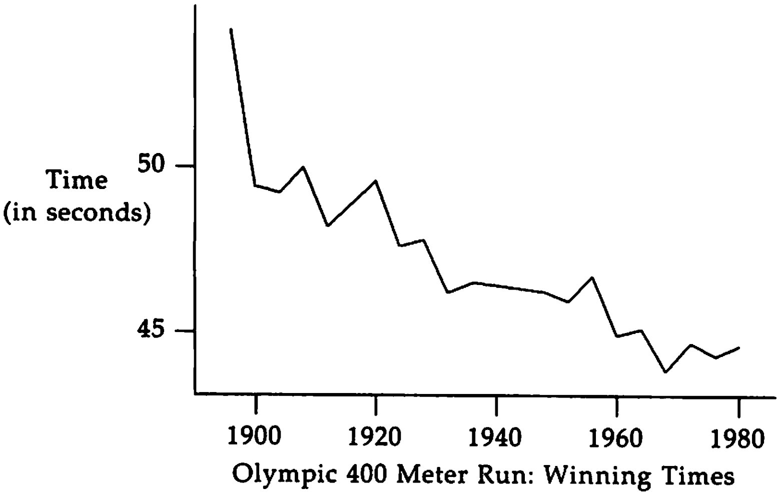
```
.G1
draw solid
copy "mtimes.cp"
.G2
```

produces the graph



Fine-tuning the Details

You can make the graph more attractive by modifying its frame and adding labels:



These details were added with the following lines:

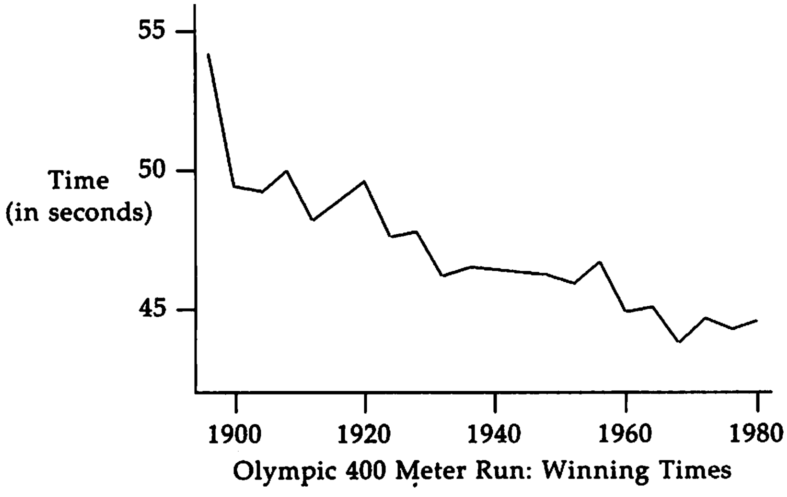
```
.G1
frame invis ht 2 wid 3 left solid bot solid
label left "Time" "(in seconds)"
label bot "Olympic 400 Meter Run: Winning Times"
draw solid
copy "mtimes.cp"
.G2
```

The **frame** instruction describes the graph's bounding box: the overall **frame** (which has four sides) is **invisible**; its **height** is two inches, and its **width** is three inches (no change from default sizes for height and width); and the **left** and **bottom** sides are **solid** (they could have been **dashed** or **dotted** instead). The **labels** appear on the **left** and **bottom**, as requested.

To set the range of each axis, **grap** examines the data and pads both dimensions by seven percent at each end. The **coord** (coordinates) line, as you saw in the opening example, allows you to specify the range of one or both axes explicitly. That is, it turns off automatic padding:

```
.G1
frame invis ht 2 wid 3 left solid bot solid
label left "Time" "(in seconds)"
label bot "Olympic 400 Meter Run: Winning Times"
coord x 1894,1982 y 42, 56
draw solid
copy "mtimes.cp"
.G2
```

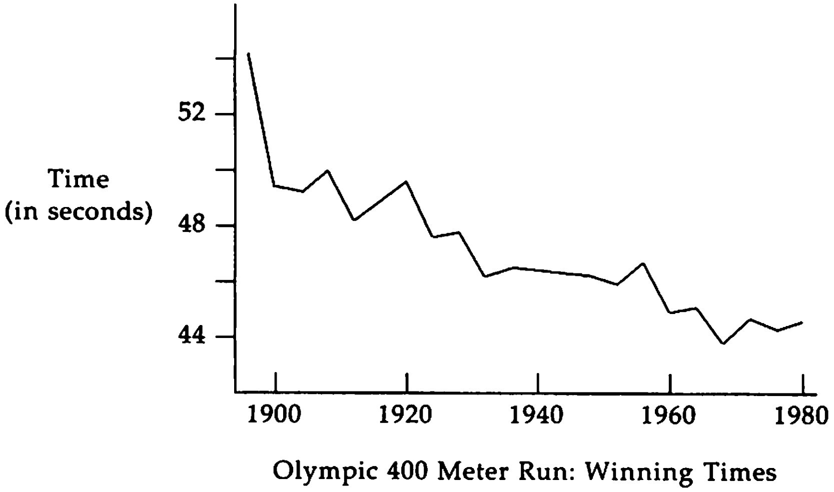
The *y*-axis now ranges from 42 to 56 seconds (a little more than before), and the *x*-axis from 1894 to 1982 (a little less):



The ticks in the preceding graphs were generated by **grap's** guessing at reasonable values. If you would rather provide your own, you may use the **ticks** instructions, which come in the two varieties illustrated below:

```
.G1
frame invis ht 2 wid 3 left solid bot solid
label left "Time" "(in seconds)" left .2
label bot "Olympic 400 Meter Run: Winning Times"
coord x 1894,1982 y 42, 56
ticks left out at 44 "44", 46, 48 "48", 50, 52 "52", 54
ticks bot in from 1900 to 1980 by 20
draw solid
copy "mtimes.cp"
.G2
```

The first **ticks** instruction deals with the left axis: it puts the ticks facing **out** at the numbers in the list. **grap** puts labels only at values with strings, except that when no labels at all are given, each number serves as its own label, as in the second **ticks** instruction. That line is for the bottom axis: it puts the **ticks** facing **in** at steps of 20 from 1900 to 1980. The instruction **ticks off** turns off all ticks. **grap** does its best to place labels appropriately, but it sometimes needs your help: the **left .2** clause moves the left label 0.2 inches further left to avoid the new ticks.



Combining Different Statistics

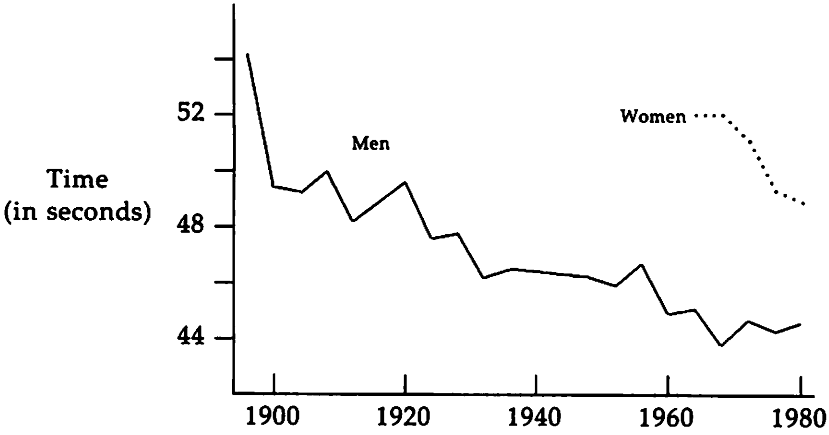
The file **wtimes.cp** contains the times for the women's 400 meter race, which has been run only since 1964:

```
1964 52
1968 52
1972 51.08
1976 49.29
1980 48.88
```

Rather than redrawing an entirely different graph in order to include the women's times, you can use a **grap** instruction. To add these times to the graph, you use the instruction **new**:

```
.G1
frame invis ht 2 wid 3 left solid bot solid
label left "Time" "(in seconds)" left .2
label bot "Olympic 400 Meter Run: Winning Times"
coord x 1894,1982 y 42, 56
ticks left out at 44 "44", 46, 48 "48", 50, 52 "52", 54
ticks bot in from 1900 to 1980 by 20
draw solid
copy "mtimes.cp"
new dotted
copy "wtimes.cp"
"Women" size -3 at 1958,52
"Men" size -3 at 1915,51
.G2
```

The instruction, **new**, tells **grap** to end the old curve and start a new curve (which in this case will be drawn with a dotted line):

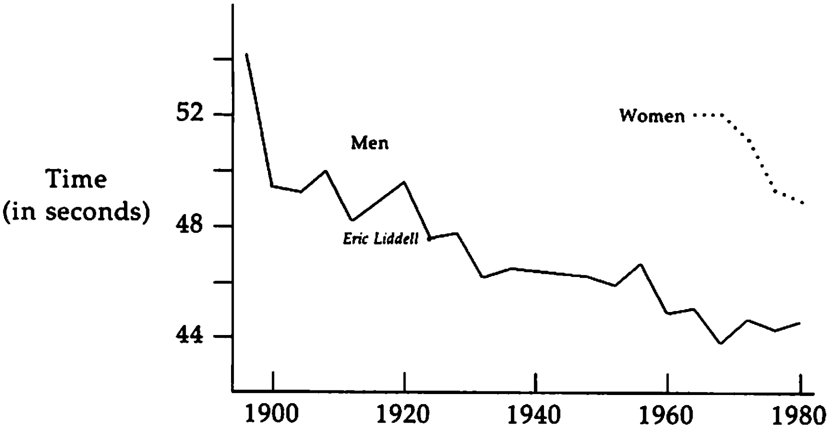


Olympic 400 Meter Run: Winning Times

The text was placed on the graph with instructions of the form

"string" at *xvalue, yvalue*

The **size** clauses following the quoted strings tell **grap** to shrink the characters by three points (absolute point sizes may also be specified). Strings are usually centered at the specified position but can be adjusted by clauses to be illustrated shortly. Consider the next graph as a brief introduction to the text adjustment feature. (Those of you who know **pic** will feel at home):



Olympic 400 Meter Run: Winning Times

This version incorporates text to emphasize Eric Liddell's 1924 gold medal performance of 47.6 seconds. (Remember *Chariots of Fire*?) The graph itself is, of course, only slightly different than the one preceding it. In fact, only two lines have been added to the input file:

```
.G1
frame invis ht 2 wid 3 left solid bot solid
...
"Men" size -3 at 1915,51
bullet at 1924,47.6
"\fIEric Liddell\fR  " size 6 rjust at 1924,47.6
.G2
```

These last two lines place a dot, or bullet, at the point of Liddell's winning time and place his name next to the bullet. There are a few details worth noticing here. First, markers like the bullet are not text and therefore do not require double quotes. Second, Eric Liddell's name is specified to be set in an italic font with the escape sequence \f. Next, the size of the name is expressed as an absolute value, unlike its predecessors, which were set using relative values. Finally, using the instruction **rjust**, Liddell's name is adjusted to be set off slightly from the bullet. In addition, space has been apportioned inside the double quotes.

Expressing Exponential Values

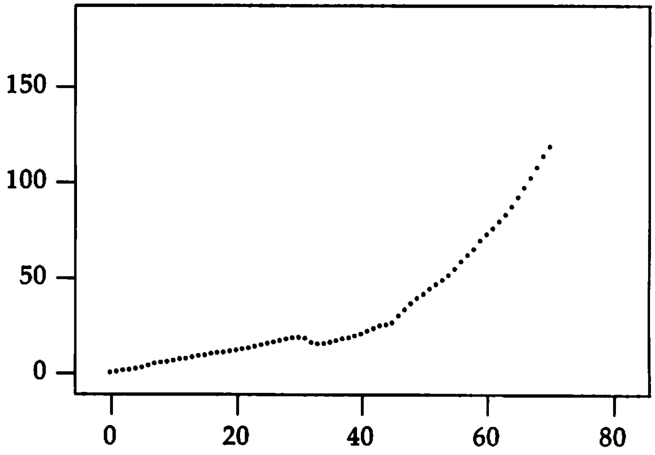
The file **phone.cp** records the number of telephones in the United States from 1900 to 1970:

```
00 1.3
01 1.8
02 2.3
03 2.8
04 3.3
...
70 120.2
```

The file contains two columns of information: the first column is a list of years (given in abbreviated form); the second is a corresponding list of the number of U.S. telephones in service (given in millions, truncated to the nearest hundred thousand). The simple **grap** program

```
.G1
copy "phone.cp"
.G2
```

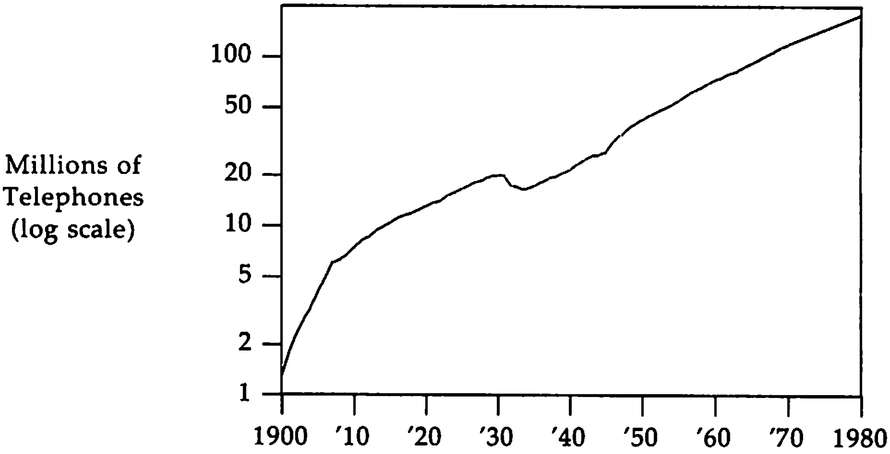
produces the following graph:



The number of telephones appears to grow exponentially. To represent that different magnitude of growth, you can refine the simpler, preceding graph into a slightly more sophisticated one. Focusing on this expansion's exponential quality, you can use a logarithmic *y*-axis with which to plot the data. You do this by adding **log y** to the **coord** instruction. What is more, you could add label changes, more ticks, and a solid line to enhance the graph's appearance:

```
.G1
label left "Millions of" "Telephones" "(log scale)" left .5
coord x 0,70 y 1,130 log y
ticks left out at 1, 2, 5, 10, 20, 50, 100
ticks bot out at 0 "1900", 70 "1970"
ticks bot out from 10 to 60 by 10 "'%g"
draw solid
copy "phone.cp"
.G2
```

The third **ticks** instruction uses a string, %g, which provides a method for presenting text in a particular format. In the example above, the string is used to place an apostrophe before each number **from 10 to 60**. In effect %g represents each number to be printed in the given range. The string, "19%g", for example, would have produced the full, four-digit year for each entry along the *x*-axis. (Those readers who use the C Programming Language will recognize **%g** as a **printf** format string.) To suppress labels, use the empty format string (""). Finally, the graph produced by this **grap** program is the following:

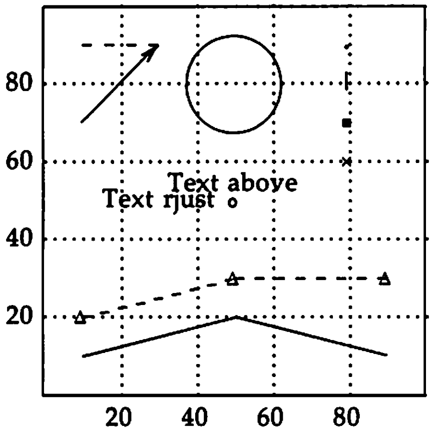


The number of telephones grew rapidly in the first decade of this century and then settled down to an exponential growth rate interrupted only by the Great Depression. A post-war growth spurt prompted a return to the pre-Depression curve.

In the "**grap** Sampler," which appears in the closing pages of this tutorial, you will see a variety of complex **grap** programs that also evolved from simple formats.

Manual Drawing

All the examples so far have placed data on the graph implicitly by copying a file of numbers (either a time series with one number per line or pairs of numbers). It is also possible to draw points and lines explicitly. Here is a graph reflecting a specific placement of forms and text:



The **grap** instructions to draw on a graph are given in the following file:

```
.G1
frame ht 2 wid 2
coord x 0,100 y 0,100
grid dotted bot from 20 to 80 by 20
grid dotted left from 20 to 80 by 20
"Text above"  above at 50,50
"Text rjust  " rjust at 50,50
bullet at 80,90
vtick  at 80,80
box    at (80,70)
times  at 80, 60
circle at 50,50
circle at 50,80 radius .25
line dashed from 10,90 to 30,90
arrow from 10,70 to 30,90
draw A solid
draw B dashed delta
next A at 10,10
next B at 10,20
next A at 50,20
next A at 90,10
next B at 50,30
next B at 90,30
.G2
```

The way of expressing the coordinate system (**at n, n**) and the English-oriented instructions (**above** and **rjust** or **dashed** and **dotted**) make **grap** an intelligible language.

The **grid** instruction is similar to the **ticks** instruction except that grid lines extend across the frame. The next two instructions plot text at specified positions. The markers (**boxes, circles, times** marks, and **vticks**) appear centered at their named coordinates.

The **circle** instruction, for instance, draws a circle whose center is the place specified by **at**. The circle's size may be determined by stating the radius. The circle above is said to be .25 because **grap** understands all dimensions to be in inches though, like **pic**, you can scale your dimensions. If no radius is given, then the circle will be a default size: the small circle shown at the center of the graph.

The **line** and **arrow** instructions draw the obvious objects shown at the upper left. The plotting characters (such as **bullet**) are implemented as predefined macros, but more on that later.

This example also illustrates the combined use of the **draw** and **next** instructions. Saying **draw A solid** defines the style for a connected sequence of line fragments to be càlled **A**. Each subsequent instruction of **next A at** *point* adds *point* to the end of **A**. There are two such sequences active in the above example (**A** and **B**); notice that their **next** instructions are intermixed. Because the predefined string **delta** follows the specification of **B**, that string is plotted at each point in the sequence.

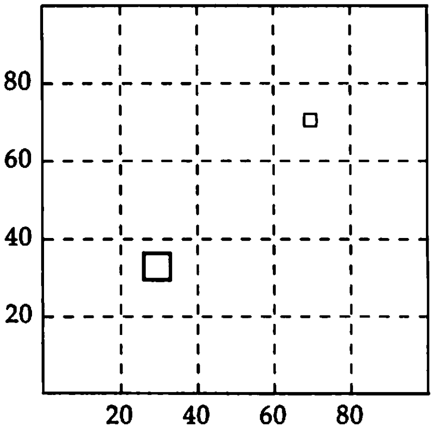
grap has numeric variables (implemented as double-precision floating point numbers) and a familiar assortment of arithmetic operators and mathematical functions.

Macros

grap provides the same rudimentary macro facility that **pic** does. Consider the following **grap** file:

```
.G1
frame ht 2 wid 2
coord x 0,100 y 0,100
grid dashed bot from 20 to 80 by 20
grid dashed left from 20 to 80 by 20
define bsquare X "\s+9\(sq\s-9" X
define lsquare X "\s-1\(sq\s+1" X
bsquare at 30,30
lsquare at 70,70
.G2
```

Two macros, **bsquare** (big square) and **lsquare** (little square), are **defined** using the **troff** escape sequences, **\(sq** and **\s**. The big square is incremented three point sizes. And the little square is decremented one point size. They are then simply stated along with their respective locations. The graph looks like this:



The method for defining macros is simple:

define *macro* X *replacement text* X

This definition stores the data from *replacement text* in the variable *macro.* Any character that does not appear in *replacement text* may be used as delimiters for *replacement text.* Any subsequent occurrence of *macro* will be replaced by *replacement text.* (The **bsquare** and **lsquare** statements of *replacement text* are surrounded by double quotes because they are text, **troff** text in this case. As such they must conform to the **grap** rules for presenting text and take quotes.)

The replacement text of a macro definition may contain occurrences of **$1**, **$2**, etc.; these will be replaced by the corresponding actual arguments when the macro is invoked. The invocation for a macro with arguments is

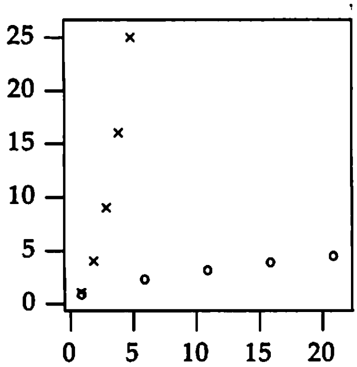
name (arg1, arg2, ...)

Non-existent arguments are replaced by null strings.

The following **grap** program uses macros and arithmetic to plot crude approximations to the square and square root functions:

```
.G1
frame ht 1.5 wid 1.5
define square X ($1)*($1) X
define root ! exp(log($1)/2) !
define P %
    times at i, square(i); i=i+1
    circle at j, root(j); j=j+5
%
i=1; j=1
P; P; P; P; P
.G2
```

Because **grap** has the square root function **sqrt**, the macro **root** is valuable only for the purposes of demonstration. The delimiters used in the preceding example may not have been obvious. **X** and **Y** will be used as delimiters in most of the remaining examples. The preceding grap program produces



copy thru

To plot files that are not stored as a time series or as *x*, *y* pairs, **grap** has a special mechanism. The **copy** instruction has a **thru** parameter that allows each line of a file to be treated as though it were a macro call, with the first field serving as the first argument, the second field serving as the second, and so forth. (Of course, using **$1**, **$2**, etc., the order of arguments can differ from the given sequence of columns: when given first, **$2**—representing the second column—would make column two equal to argument one.) The **thru** instruction is the typical **grap** mechanism for plotting files that are not stored as time series or as *x*, *y* pairs. Its use will be illustrated with the file, **states.cp**, which contains data on the fifty states:

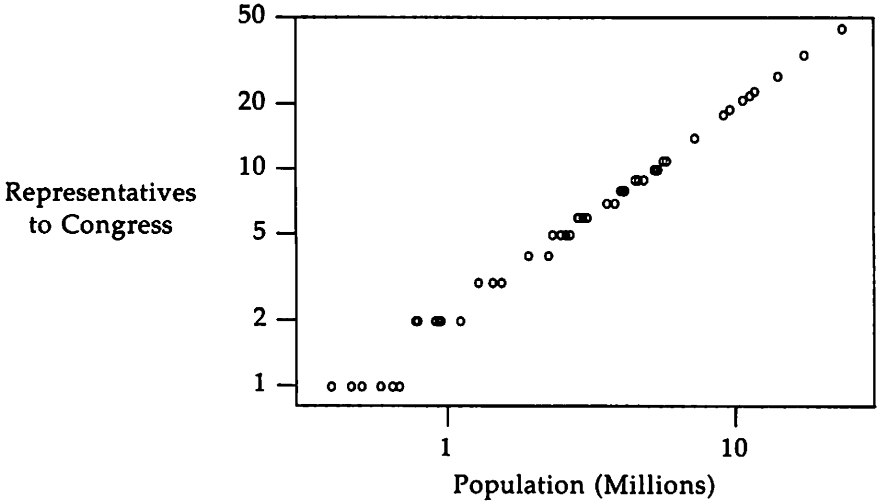
```
AK    1     401851
WY    1     469557
VT    1     511456
DE    1     594338
ND    1     652717
...
CA    45    23667902
```

The first field is the postal abbreviation of the state's name (Alaska, Wyoming, Vermont, ...), the second field is the number of Representatives to Congress from the state after the 1981 reapportionment, and the third field is the population of the state as measured in the 1980 Census. The states appear in increasing order of population.

First, the graph will be given in population, representative pairs. (In the **coord** statement, **log log** is a synonym for **log x log y.**)

```
.G1
label left "Representatives" "to Congress" left .3
label bot "Population (Millions)"
coord x .3,30 y .8,50 log log
define PlotState X circle at ($3/1e6,$2) X
copy "states.cp" thru PlotState
.G2
```

The **copy** instruction performs its usual task, reading in **states.cp** for its information. This time, however, the file was interpreted **thru** the defined macro, **PlotState**. The third column (**$3**) is divided by (**/**) one million (1×10^6, written in **grap** exponential notation as **1e6**). Because **$3** is given first, column three is the first argument for **grap**'s coordinate system. The second column, **$2**, will be the second argument. Here is the finished product:



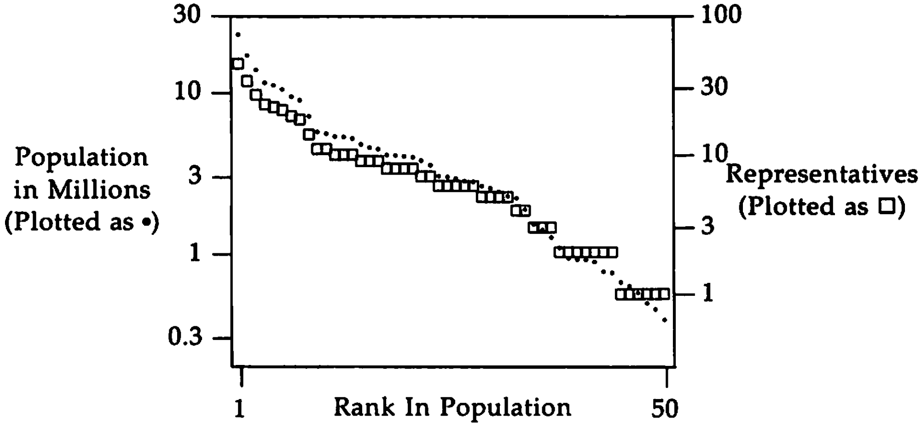
Using **circle** as a plotting symbol displays overlapping points that are obscured when the data is plotted with bullets. The representation of a state is roughly proportional to its population, except in the very small states.

Multiple Coordinates

The next plot will use the state's rank in population as the *x*-coordinate and two different *y*-coordinates: population and number of representatives. Consequently, you would use two **coord** instructions to define the two coordinate systems **pop** and **rep**. You then explicitly give the coordinate system whenever you refer to a point, both in constructing axes and plotting data.

```
.G1
frame ht 1.8 wid 2.3
label left "Population" "in Millions" "(Plotted as \(bu)"
label bot "Rank In Population" up .2
label right "Representatives" "(Plotted as \(sq)"
coord pop x 0,51 y .2,30 log y
coord rep x 0,51 y .3,100 log y
ticks left out at pop .3, 1, 3, 10, 30
ticks bot out at pop 1, 50
ticks right out at rep 1, 3, 10, 30, 100
thisrank=50
copy "states.cp" thru X
  bullet at pop thisrank,$3/1e6
  square at rep thisrank,$2
  thisrank=thisrank-1
X
.G2
```

The **copy** statement in the program uses an "immediate macro" enclosed in Xs and thus avoids having to name a macro for this task. **copying states.cp thru** the lines between Xs is functionally identical to reading the file through a defined macro. Because the program assumes that the states are sorted in increasing order of population, it generates **thisrank** internally as a **grap** variable. The program produces the following graph:



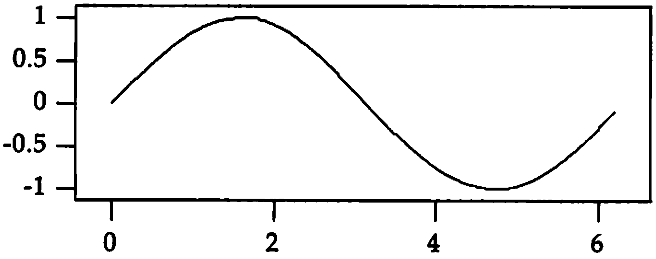
The plotting symbols were chosen for contrast in both shape and shading. This graph also indicates that representation is proportional to population. Once you see this graph, though, you should realize that you don't really need two coordinate systems: you can relate the two by dividing the population of the U.S. — about 226,000,000 — by the number of representatives — 435 — to see that each representative should count as 520,000 people. If the purpose of this graph were to tell a story about American politics rather than to illustrate multiple coordinate systems, it would be redrawn with a single coordinate system.

for Loops and if Statements

Many graphs plot both observed data and a function that (theoretically) describes the data. There are many ways to draw a function in **grap**: a series of **next** instructions is tedious but works, as does writing a simple program to write a data file that is subsequently read and plotted by the **grap** program. The **for** statement often provides a better solution. This **grap** program

```
.G1
frame ht 1 wid 3
draw solid
pi=atan2(0,-1)
for i from 0 to 2*pi by .1 do X next at i, sin(i) X
.G2
```

produces



The **for** statement uses the same syntax as the **ticks** statement, but the **from** instruction can be replaced by =, (which will look more familiar to programmers). It varies the index variable over the specified range, and for each value, executes all statements inside the delimiter characters, which use the same rules as macro delimiters. It is, of course, useful for many tasks beyond plotting functions.

The **if** statement provides a simple mechanism for conditional execution. If a file contains data on both cities and states (and lines describing states have "S" in the first field), it could be plotted by statements like

```
if "$1" == "S" then X
    PlotState($2,$3,$4)
X else X
    PlotCity($2,$3,$4,$5,$6)
X
```

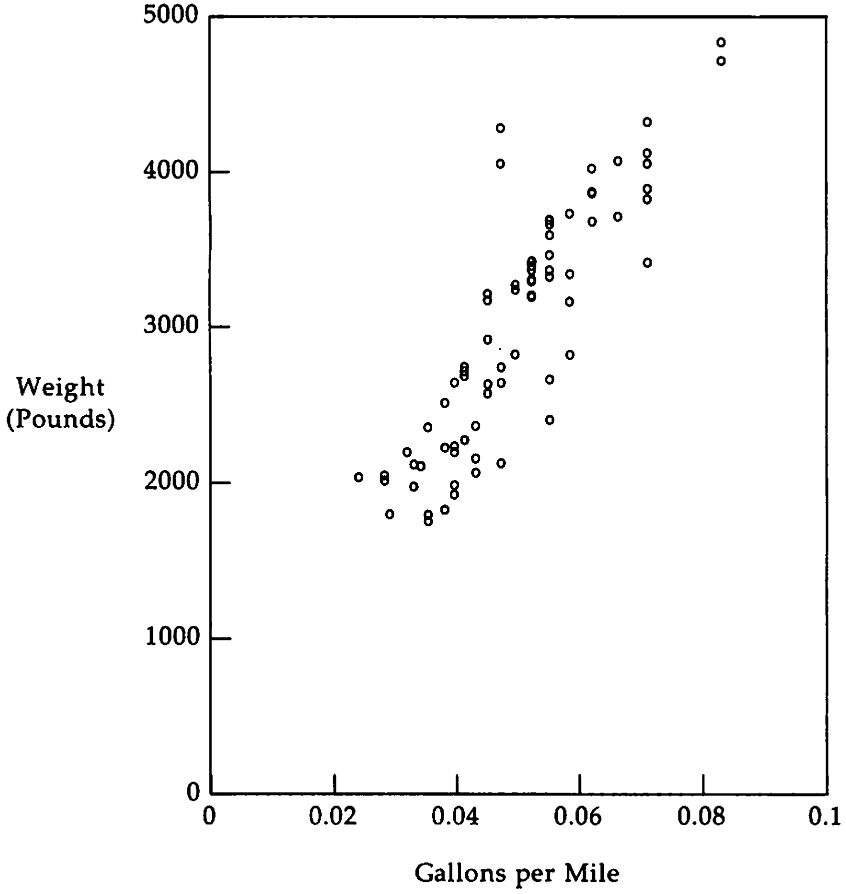
The **else-clause** is optional; delimiters use the same rules as macros and **for** statements.

grap Sampler

```
.G1
frame ht 4 wid 3.2
label left "Weight" "(Pounds)" left .3
label bot "Gallons per Mile"
coord x 0,.10 y 0,5000
ticks left in from 0 to 5000 by 1000
ticks bot in from 0 to .10 by .02
copy "cars.cp" thru X circle at 1/$1, $2 X
.G2
```

Excerpt from file **cars.cp**:

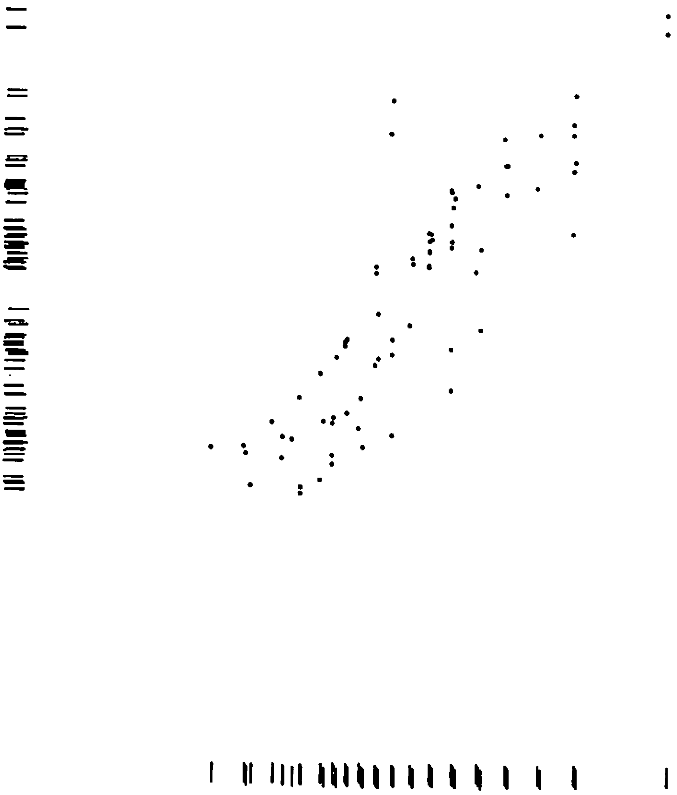
```
22 2930
17 3350
22 2640
17 2830
23 2070
...
17 3170
```


5000
4000
3000
2000
1000
0
Weight
(Pounds)
0
0.02
0.04
0.06
0.08
0.1
Gallons per Mile

```
.G1
frame invis ht 4 wid 4
coord x 0,.10 y 0,5000
copy "cars.cp" thru X
  tx=1/$1; ty=$2
  bullet at tx,ty
  tick bot at tx " "
  tick left at ty " "
X
.G2
```

Excerpt from file **cars.cp**:

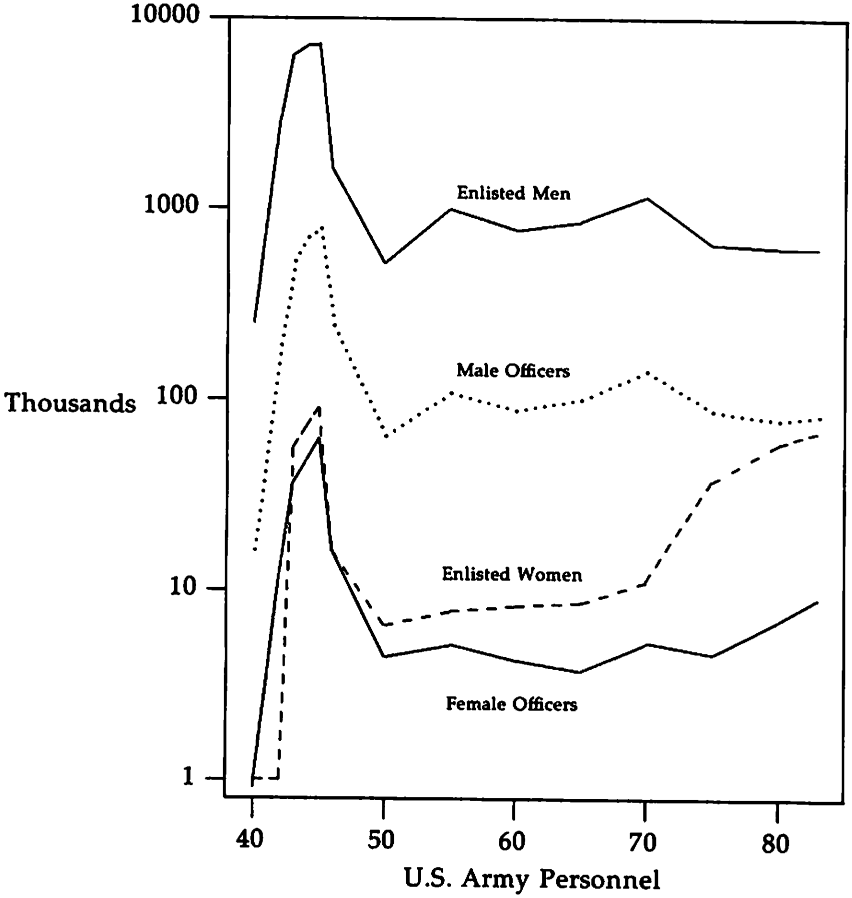
```
22  2930
17  3350
22  2640
17  2830
23  2070
...
17  3170
```



```
.G1
frame ht 4 wid 3.2
coord x 38,85 y .8,10000 log y
label bot "U.S. Army Personnel"
label left "Thousands" left .3
draw of solid   # Officers Female
draw ef dashed  # Enlisted Female
draw om dotted  # Officers Male
draw em solid   # Enlisted Male
copy "army.cp" thru X
  next of at $1,$3
  next ef at $1,$5
  next om at $1,$2
  next em at $1,$4
X
copy thru % "$1 $2" size -3 at 60,$3 % until "XXX"
Enlisted Men 1200
Male Officers 140
Enlisted Women 12
Female Officers 2.5
XXX
.G2
```

Contents of file **army.cp**:

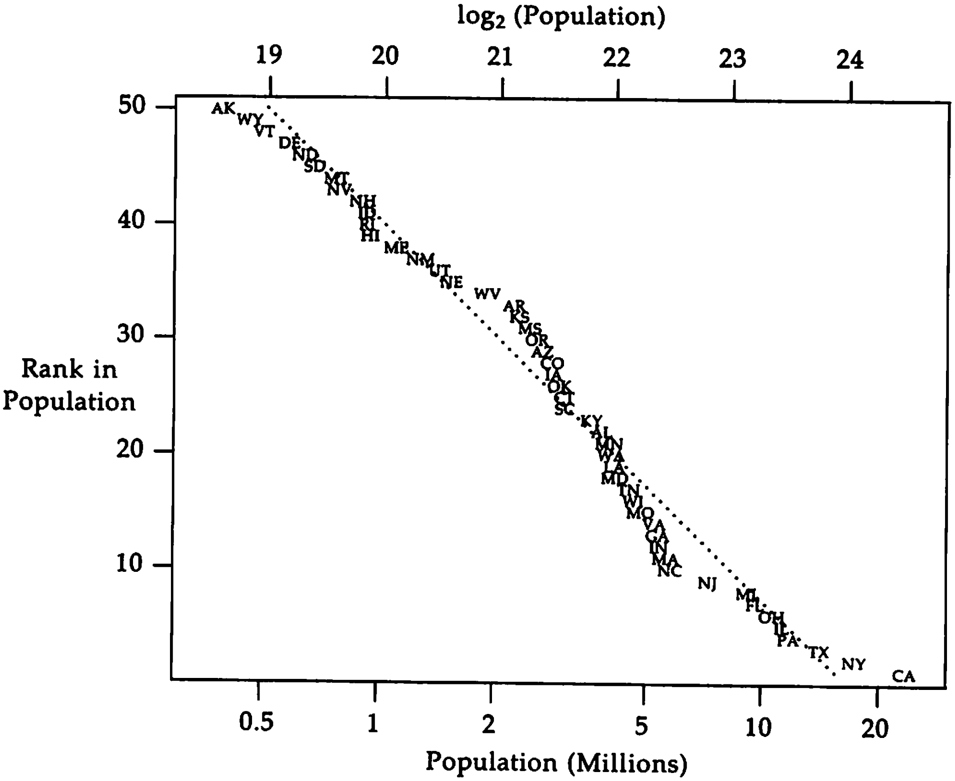
```
40  16 .9  249  1
42 190 12 2867  1
43 521 36 6358 55
44 692 47 7144 71
45 772 62 7283 90
46 240 16 1605 16
50 63 4.4 512 6.5
55 106 5.1 977 7.7
60 86 4.2 761 8.2
65 98 3.7 846 8.5
70 138 5.2 1141 11
75 85 4.5 640 37
80 76 6.8 608 58
83  80  9  606 67
```


10000
1000
100
10
1
Thousands
Enlisted Men
Male Officers
Enlisted Women
Female Officers
40
50
60
70
80
U.S. Army Personnel

```
.EQ
delim $$
.EN
.G1
frame ht 3 wid 4
label left "Rank in" "Population"
label bot "Population (Millions)"
label top "$log sub 2$ (Population)"
coord x .3,30 y 0,51 log x
define L % exp($1*log(2))/1e6 "$1" %
ticks bot out at .5, 1, 2, 5, 10, 20
ticks left out from 10 to 50 by 10
ticks top out at L(19), L(20), L(21), L(22), L(23), L(24)
thisy=50
copy "states.cp" thru X
  "$1" size -4 at ($3/1e6, thisy)
  thisy=thisy-1
X
line dotted from 15.3,1 to .515,50
.G2
.EQ
delim off
.EN
```

Excerpt from file **states.cp**:

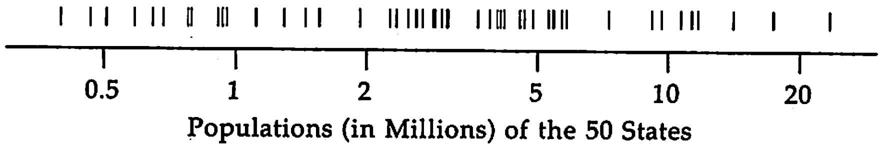
```
AK    1     401851
WY    1     469557
VT    1     511456
DE    1     594338
ND    1     652717
...
CA    45    23667902
```


log₂ (Population)
19
20
21
22
23
24
50
40
30
20
10
Rank in Population
AK
WY
VT
DE
ND
SD
MT
NV
NH
ID
RI
HI
ME
NM
UT
NE
WV
AR
KS
MS
OR
AZ
CO
IA
OK
CT
SC
KY
AL
MN
WA
LA
MD
TN
WI
MO
VA
GA
IN
MA
NC
NJ
MI
FL
OH
IL
PA
TX
NY
CA
0.5
1
2
5
10
20
Population (Millions)

```
.G1
frame invis ht .3 wid 4.5 bottom solid
label bot "Populations (in Millions) of the 50 States"
coord x .3,30 y 0, 1 log x
ticks bot out at .5, 1, 2, 5, 10, 20
ticks left off
copy "states.cp" thru X vtick at ($3/1e6,.5) X
.G2
```

Excerpt from file **states.cp**:

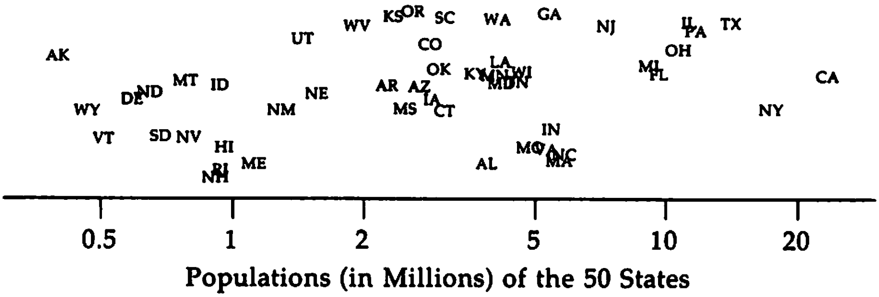
```
AK    1     401851
WY    1     469557
VT    1     511456
DE    1     594338
ND    1     652717
...
CA    45    23667902
```


0.5
1
2
5
10
20
Populations (in Millions) of the 50 States

```
.G1
frame invis ht 1 wid 4.5 bottom solid
label bot "Populations (in Millions) of the 50 States"
coord x .3,30 y 0,1000 log x
ticks bot out at .5, 1, 2, 5, 10, 20
tick left off
copy "states.cp" thru X "$1" size -4 at ($3/1e6, 100+rand(900)) X
.G2
```

Excerpt from file **states.cp**:

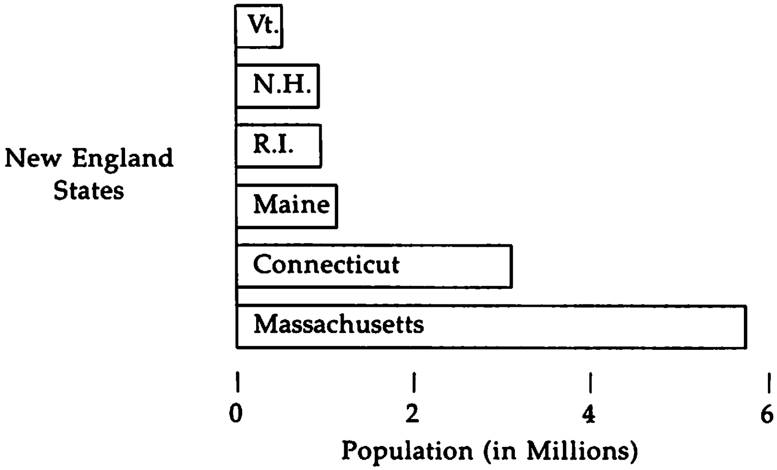
```
AK    1     401851
WY    1     469557
VT    1     511456
DE    1     594338
ND    1     652717
...
CA    45    23667902
```


KS
OR
SC
WA
GA
WV
NJ
IL
PA
TX
UT
CO
OH
AK
LA
OK
KY
MN
WI
MI
FL
MD
TN
CA
MT
ND
DE
ID
NE
AR
AZ
WY
NM
MS
IA
CT
NY
IN
VT
SD
NV
HI
MO
VA
NC
MA
AL
ME
RI
NH
0.5
1
2
5
10
20
Populations (in Millions) of the 50 States

```
.G1
ticks left off
cury=0
barht=.7
copy "newengl.cp" thru X
  line from 0,cury to ($1/1e6,cury)
  line from ($1/1e6,cury) to ($1/1e6,cury-barht)
  line from 0,cury-barht to ($1/1e6,cury-barht)
  "  $2" ljust at 0,cury-barht/2
  cury=cury-1
X
line from 0,0 to 0,cury+1-barht
bars=-cury
frame invis ht bars/3 wid 3
label left "New England" "States"
label bot "Population (in Millions)"
.G2
```

Contents of file **newengl.cp**:

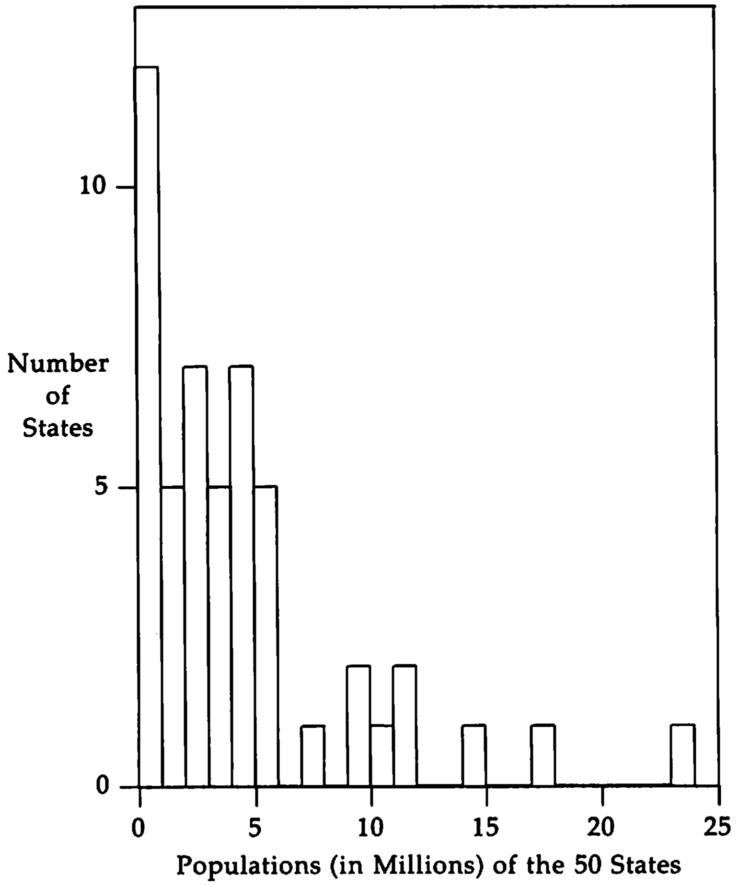
```
511456 Vt.
920610 N.H.
947154 R.I.
1124660Maine
3107576Connecticut
5737037Massachusetts
```


Vt.
N.H.
R.I.
Maine
Connecticut
Massachusetts
New England States
0
2
4
6
Population (in Millions)

```
.G1
frame ht 4 wid 3 solid
label bot "Populations (in Millions) of the 50 States"
label left "Number" "of" "States"
ticks bot out from 0 to 25 by 5
coord x 0,25 y 0,13
copy "statepop.cp" thru X
  line from $1,0 to $1,$2
  line from $1,$2 to $1+1,$2
  line from $1+1,$2 to $1+1,0
X
.G2
```

Contents of file **statepop.cp**:

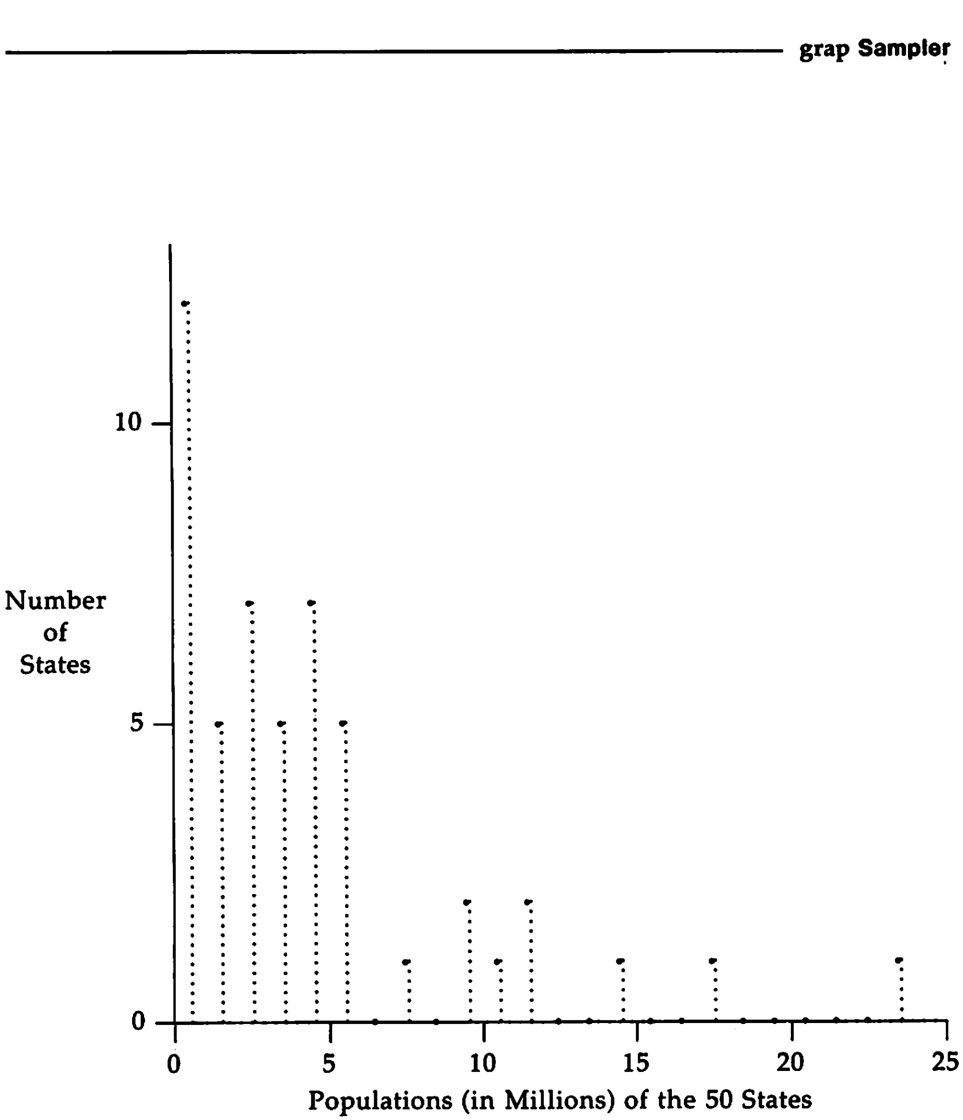
```
0 12
1 5
2 7
3 5
4 7
5 5
6 0
7 1
8 0
9 2
10 1
11 2
12 0
13 0
14 1
15 0
16 0
17 1
18 0
19 0
20 0
21 0
22 0
23 1
```


10
Number
of
States
5
0
0
5
10
15
20
25
Populations (in Millions) of the 50 States

```
.G1
frame invis ht 4 wid 4 bot solid left solid
label bot "Populations (in Millions) of the 50 States"
label left "Number" "of" "States" left .2
ticks bot out from 0 to 25 by 5
coord x 0,25 y 0,13
copy "statepop.cp" thru X
  line dotted from $1+.5,0 to $1+.5,$2
  "\(bu" size -3 at $1+.5,$2
X
.G2
```

Contents of file **statepop.cp**:

```
0 12
1 5
2 7
3 5
4 7
5 5
6 0
7 1
8 0
9 2
10 1
11 2
12 0
13 0
14 1
15 0
16 0
17 1
18 0
19 0
20 0
21 0
22 0
23 1
```


Number
of
States
10
5
0
0
5
10
15
20
25
Populations (in Millions) of the 50 States

Index

Table of Contents

Introduction

mv is a package of UNIX System V macros that format text for two types of transparencies (also called foils): viewgraphs (in a variety of dimensions), and 35 mm slides or 2x2 super-slides. The **troff**ed output of the **mv** macros is not the transparency or slide itself, but the text on opaque paper that will later be used to make the transparency or slide. Because the text of the viewgraph is stored in a UNIX system file, and the output is on paper, you derive the additional benefit of being able to check the output, make changes, or correct errors, before you go to the time and expense of making the final transparency.

The prerequisites to benefit from this tutorial are as follows:

- You should know what a file and directory are, and know how to create them. See the *UNIX System V User Guide.*
- You should know how to use a UNIX system text editor (**ed, vi,** and **ex** are examples). See the *UNIX System V User Guide.*
- You should be familiar with **troff**. See the tutorial in this book "The Formatter **troff**: A Tutorial." Also, the "**nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual* discusses **troff** in detail.

After reading this tutorial, you will be able to make transparencies in a variety of styles, in different fonts, and with oversize titles, and you will be able to use the reference material provided later in this chapter. (See the section titled "Reference Material.")

Some Examples

The following two examples show you some of the macros that you can use to format text for transparencies with **mv**. As you look them over you will see that the macros in the **mv** package are much like other formatting macros you may have used: they are named with a one or two letter upper-case string, they begin on a new line, and the first character is always a dot (.).

The macros shown in these examples are explained in greater detail in later sections of the tutorial. The output for these two examples, and for other examples used in this tutorial, is in the section "**mv** Sampler."

Example 1: A Simple Transparency

Suppose you have a file named **steps.in**, which contains the following lines:

```
.VS
The Primary Steps in Preparing Documents
.B
Planning
.B
Writing
.B
Editing
.B
Typesetting
```

.VS is one of several foil-start macros. (There are no foil-end macros.) Among other things, foil-start macros tell the typesetter what size the transparency should be. Because it begins with **.VS**, this one will be 7x7 inches. **.B**, as we've seen, is a level macro; it specifies a level of indentation and a bullet (•) to mark the text that follows.

You can process the foil by typing one of the following command lines:

mvt steps.in |*phototypesetter*
or
troff −mv steps.in | *phototypesetter*

Ask your system administrator which phototypesetter is appropriate to process your output.

Example 2: A More Elaborate Transparency

Suppose you wanted to make the transparency from Example 1 more elaborate:

```
.Vw
There are four steps in preparing documents
.B
Planning
.C
What documents exist?
Can they be revised?
.C
Are new documents needed?
.B
Writing
.B
Editing
.C
By peers and supervision
.B
Typesetting
.C
Using DOCUMENTER'S WORKBENCH Software
```

.Vw is another of the foil-start macros; it produces text that will fit on a 7x5 inch transparency. **.C** specifies that another level be added beyond **.B**.

The Foil Start Macros (.VS, .Vw, .Vh, .VW, .VH, .Sw, .Sh, .SW, .SH)

Every **mv** file must contain a foil-start macro. There are nine of them; each generates output for a different sized foil.

Macro Name	Size of Frame Opening and Type of Foil
.VS	7x7 inch viewgraph or 2x2 inch super-slide
.Vw	7x5 inch viewgraph
.Vh	5x7 inch viewgraph
.VW	9x7 inch viewgraph
.VH	7x9 inch viewgraph
.Sw	7x5 inch 35-mm slide
.Sh	5x7 inch 35-mm slide
.SW	9x7 inch 35-mm slide
.SH	7x9 inch 35-mm slide

NOTE

Note that .VW and .SW actually produce text for a 7x5.4 transparency (because typesetter paper is commonly less than 9 inches wide). The output from these two macros will have to be enlarged by a factor of 9/7 before they can be used to make 9x7 inch transparencies.

The first character of a foil-start macro's name distinguishes between viewgraphs (V) and slides (S), and the second character specifies the dimensions of the foil. Varieties of sizes are as follows:

S	square
w	small and wide (7x5)
h	small and high (5x7)
W	big and wide (9x7)
H	big and high (7x9)

By default, each foil-start macro puts out three right-justified lines of identification information at the top of each foil.

The current date in the form *mo/dy/yr*
AT&T
FOIL *n*

where *n* is the sequence number in the current series of transparencies you are preparing. You can change the default heading information by using three optional arguments to the foil-start macro:

.*XX* [*n*] [*id*] [*date*]

where *XX* stands for one of the foil-start macros, *n* is the foil identifier (typically a number), *id* is other identifying information, such as your initials or your company name, and *date* is the date. Example 6 shows the use of these options in an input file, and the "**mv** Sampler" shows the output for this example.

Cross-hairs (+) mark the area of a transparency that will show through the opening of the cardboard frame to help you center the transparency in the frame. All foils other than the square (.VS) also have a set of horizontal or vertical crop marks that show how much of the transparency will be seen if it is made into a slide rather than into a viewgraph.

The Level Macros (.A, .B, .C, .D)

Example 3: The Level Macros

The text of this transparency explains **mv**'s four levels of indentation. The "**mv** Sampler" shows the formatted output.

```
.Vh
.T "Foil Levels and Level Marks"
.A
This is the .A level.
.B
This is the .B level,
.B
and so is this;
.C
this is the .C level
.C
and so is this;
.D
and this is the .D level,
.D
and so is this.
.A
The large bullet, the dash, and the small bullet are the default "marks"
for levels .B, .C, and .D respectively.
However, you may arbitrarily mark these three levels:
.B 2.
like this (.B level);
.C c.
like this (.C level);
.D *
like this (.D level);
.D iv.
or like this (.D level again).
.A
You cannot mark the .A level.
.B
You may include as many lines of text as you wish in any item at any
level; \f3troff\fP fills the text,  but it does not adjust the margin or
hyphenate words (as with this .B level item).
```

troff processes text after every level macro with line filling turned on. That is, the number of lines input do not necessarily match the number output.

Each of the foil-start macros automatically calls the **.A** level macro, that is

```
.Vw
There are four steps in preparing documents
```

and

```
.Vw
.A
There are four steps in preparing documents
```

give you the same results.

.A precedes the line following it with a half-line of space. This blank half-line can be suppressed by giving an argument to **.A**. In fact any argument—any character or string of characters—will do. In the following example the character **x** is used as an argument to **A** (though it could have been **N** or **22** and still would have had the same effect):

```
.A x
There are four steps in preparing documents
```

Thus, the line following **.A x** will not be preceded by a blank half-line.

The text that follows a **.B** macro is marked by a bullet, indented three spaces to the right, and is one-half a **troff** vertical space (one blank line) down from the preceding text. The mark that a **.B** macro produces can be changed, and the point size of the mark can be changed as well, by using the arguments shown below:

.B [*mark* [*size*]]

.C level text indents farther to the right and also spaces one-half a **troff** vertical space (one blank line) down from the preceding text. The default mark it produces is a long (em) dash (—).

.C [*mark* [*size*]]

The text following a **.D** level-macro is marked by a small bullet, indented even farther to the right, and begins on a new line (but no blank line precedes the text of a **.D** macro.

.D [*mark* [*size*]]

Example 4: Repeated Level Marks

If you call the same level-macro without any intervening text, you generate a corresponding number of marks:

```
.VH
.T "Repeated Level Marks"
.A
Because the .A macro has no label, repeated .A macro calls are
ignored.
.A
.A
.A
.A
These are indentation level requests:
.B
.B
.B
Notice the mark.
.D
.D
Notice the mark.
.B
.B
Notice the mark.
.B
.A
This text is back to the left margin.
```

Look at the output of this example in the "**mv** Sampler" section to clarify this principle.

Other Macros

Titles (.T)

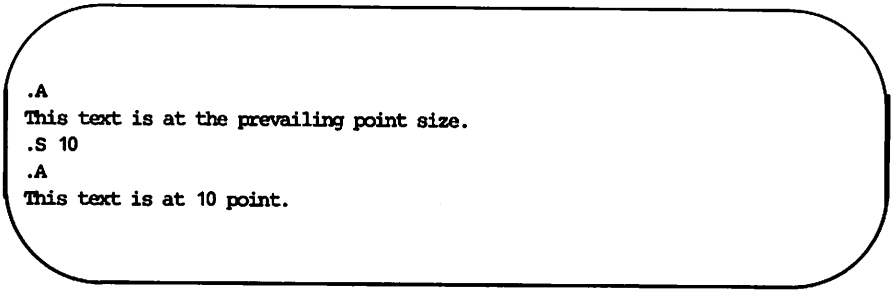
Example 4 also shows how to use .T to create a centered title for your foil:

```
.VH
.T "Repeated Level Marks"
```

On output, the size of the text that is the argument to a .T macro is four points larger than the prevailing point size. You must enclose the argument to .T in double quotes if it contains blank spaces.

Point Sizes and Line Lengths (.S)

To change the point size of your text, use the .S macro.



```
.A
This text is at the prevailing point size.
.S 10
.A
This text is at 10 point.
```

If you give .S a null first argument (that is, .S ""), you restore the previous point size.

```
.A
The text is at the prevailing point size.
.S 10
.A
This text is at 10 point.
.S ""
.A
This text is back at the prevailing point size.
```

A negative argument to .S (for example, −2), reduces the default point size by the amount specified. If the argument is positive (+2), it is added to the default, and if there is a null argument, as above, it becomes the new point size.

Change your foil's line length by giving .S a second argument, which becomes the prevailing length. You may specify the line length in any metric that you choose (ems, inches, and so on). If you do not specify the metric of this second argument and it is less than 10, **troff** interprets it as inches; if the second argument is greater than 10, it is taken as **troff** units (see the "**nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual*).

Global Indents (.I)

Shift the entire text (except titles) of the foil right or left with the .I macro:

```
        .I +10 a x
```

The first argument specifies the amount of indentation **mv** will use to establish a new left margin. This argument may be signed positive or negative, indicating right or left movement from the current left margin. If unsigned, the argument specifies a new margin, relative to the initial default margin. If you do not specify units, **mv** assumes that you mean inches (see the "**nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual* for legal **troff** formatter units). If the argument is null or omitted, **mv** assumes 0i (zero inches), causing **troff** to revert to the initial default margin.

If you specify a second argument, **.I** calls the **.A** macro before exiting. The third argument, if you use it, is passed to the **.A** macro to suppress vertical spacing. Example 5 illustrates this point.

Changing Fonts (.DF)

Hȩlvetica Regular, mounted in position 1 on your typesetter, is the default font for viewgraphs and slides formatted with **mv**. You can mount additional fonts and change the default font with the **.DF** macro. The arguments to the **.DF** macro in the following example load roman font in position 1 (so that it becomes default), italic in position 2, bold in position 3, and Helvetica in position 4:

```
.DF 1 R 2 I 3 B 4 H
```

After you enter this line in your file, using the escape sequence for changing fonts (\f*n*) loads the fonts you specified for each position.

```
.B
\f1roman\f2italic\f3bold\f4Helvetica
```

If you use **.DF**, you must use it immediately before a foil-start macro.

Using the following arguments to the **.DF** macro is equivalent to using **mv**'s default loading of fonts:

```
.DF 1 H 2 I 3 B 4 S
```

Changing Vertical Spacing (.DV)

Change the default vertical spacing of the four level macros with **.DV**. **mv** calculates vertical spacing in **troff** vertical units (v). This control line changes the spacing for the **.A** level to .7v, spacing for the **.B** level to .4v, and the spacing for the **.C** level to 0.

```
.DV .7v .4v 0v
```

The default setting is equivalent to the setting shown in this line:

```
.DV .5v .5v .5v 0v
```

Underlining (.U)

To underline a line of text, precede it with the .U macro, like this:

```
.U "DOCUMENTER'S WORKBENCH Software"
```

The .U macro takes one or two arguments. The first argument is the string to be underlined, the second, if you include it, is not underlined but concatenated to the first argument.

Synonyms

In general, you should not intermix **troff** formatter requests arbitrarily with the **mv** macros, because this often leads to undesirable (and sometimes surprising) results. The **mv** package recognizes the following upper-case text synonyms for the corresponding lower-case **troff** requests.

Synonym	Meaning
.AD	turn on line adjustment
.BR	line break
.CE	center the line
.FI	turn on line filling
.HY	turn on hyphenation
.NA	turn off hyphenation
.NF	turn off line filling
.NH	turn off hyphenation
.NX	next file
.SO	switch source file
.SP	space
.TA	set tab stops
.TI	temporary indent

troff requests that are safe to use correspond to these upper-case synonyms. The "**nroff/troff** Technical Discussion" in the *Technical Discussion and Reference Manual* explains in detail how the lower-case equivalents to these upper-case synonyms work.

You should use other **troff** formatter requests sparingly (if at all). Be particularly careful when using requests that affect point size, indentation, page offset, line and title lengths, and vertical spacing between lines. In cases such as these, use the .I and .S macros instead, because they do more work for you. For example, the .S macro changes point size and adjusts vertical spacing appropriately.

Line Breaks

The .S, .DF, .DV, and .U macros do not cause a line break. The .I macro causes a break only if you call it with more than one argument. All other **mv** macros always cause a break. The **troff** synonyms .AD, .BR, .CE, .FI, .NA, .NF, .SP, and .TI also cause a break.

Line Filling, Adjusting, and Hyphenation

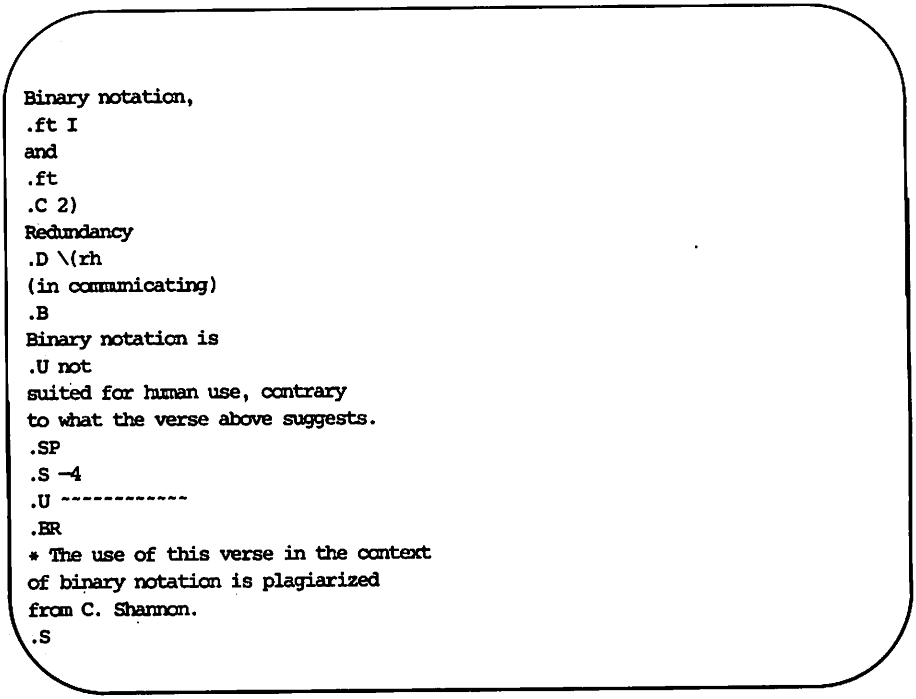
By default, the **mv** macros turn on line filling, but they neither adjust nor hyphenate lines. These defaults can, of course, be changed by using the **troff** synonyms for filling, adjusting and hyphenation.

Example 5: A Complicated Transparency

The following input combines a variety of **mv** features, specifying a title, changing point size and fonts, and using some of the **troff** synonyms. The output of this example is in the "**mv** Sampler" section.

```
.DF 1 R
.VS 5 Complex
.T "Of Bits & Bytes & Words"
.S -4
.I 3 A x
.ft I
But let your communication be, Yea, yea;
Nay, nay: for whatsoever is more than these
cometh of evil.*
.ft
.I +1 a nospace
Matthew 5:37
.BR
.S
.I 0
.A
Binary notation has been around for a
.S +6
long
.S
time.
.B
The above verse tells us to use:
.C 1)
```

Example 5 continued:



```
Binary notation,
.ft I
and
.ft
.C 2)
Redundancy
.D \(rh
(in communicating)
.B
Binary notation is
.U not
suited for human use, contrary
to what the verse above suggests.
.SP
.S -4
.U -------------
.BR
* The use of this verse in the context
of binary notation is plagiarized
from C. Shannon.
.S
```

Using the Preprocessors with mv

You can use various preprocessors to typeset information that requires more powerful formatting capabilities.

Use **tbl** to set up columns of data within a viewgraph or slide. The **.TS** and .TE macros are not defined in the **mv** macro package, but are merely flags to **tbl**. You can learn about **tbl** by reading "The Preprocessor **tbl**" in this User's Guide. You can find details about **tbl** in the "**tbl** Technical Discussion" in the *Technical Discussion and Reference Manual.*

Use the **eqn** preprocessor to typeset mathematical expressions and formulas on transparencies. The chapter "The Preprocessor **eqn**" in this guide provides details about using this preprocessor. .EQ and .EN are flags to **eqn**, and are not defined in the **mv** macro package.

You can also use the **pic** and **grap** preprocessors with **mv**. This guide provides tutorials on both of them. Their delimiters are flags to their programs, and are not defined in the **mv** macro package.

Example 6: A Transparency with Tables and an Equation

Here is an example of a foil containing an equation and a table:

```
.VS 6 "The Works: Output"
.EQ
delim $$
gsize 14
.EN
.I 0 a
.SP
.TS
center doublebox ;
Cip+4 ¦ Cip+4 S S
^ ¦ L L L
^ ¦ C ¦ C ¦ C
^ ¦ C ¦ C ¦ C
Li ¦ C ¦ C ¦ N .
User's<TAB>Hardware
<TAB>_<TAB>_<TAB>_
<TAB>UNIX\*(Tm<TAB>Model<TAB>Serial
<TAB>System<TAB>\^<TAB>Number
=
OS Dev.<TAB>A<TAB>VAX<TAB>54
SGS Dev.<TAB>B<TAB>11/70<TAB>3275
Low-End<TAB>C<TAB>11/23<TAB>221
_
And now . . .<TAB>T{
.NA
Some text and an equation:
T}<TAB>T{
$ zeta (s) = prod from k=1 to inf k sup -s $
.AD
T}<TAB>1.2
.sp
.TE
.EQ
delim off
gsize 10
.EN
```

Using Constant Width Font

Your phototypesetter may have a constant width font available. Use the .DF macro to define the font position. For example:

```
.DF 1 R 2 I 3 CW
```

Then, define the .CW and .CN macros to include the font change, like this:

```
.de CW
.NF
.ft 3
..
.de CN
.FI
.ft 1
..
```

Reference Material

Phototypesetter Output

Obtain typeset output with the **mvt** command.

mvt [*options*] [*file* ...] | *phototypesetter*

The *file* argument names a file that contains the text (and macros) to be typeset. The *options* argument can be one or more of the following:

−a	previews output on a terminal (other than a TEKTRONIX 4014)
−e	calls **eqn**
−t	calls **tbl**
−p	calls **pic**
−g	calls **grap**
−T*tty_type*	formats output for *tty_type,* where *tty_type* refers to a phototypesetter. **mv** supports the *tty_type* value 4014 (TEKTRONIX 4014).
−D*dest*	directs output to device *dest.* **mv** supports the *dest* value 4014 (TEKTRONIX 4014).
−z	directs output to the standard output.

If you use a hyphen (−) in place of *file,* **mvt** reads the standard input (rather than a file).

The *phototypesetter* argument is the device name of a printer or terminal capable of producing phototypeset output. Your system administrator should know what name is appropriate.

Output Approximation on a Terminal

You can obtain an approximation of the typeset output with the **mvt** option **−a**.

mvt −a [*file* ...]

Use this option to preview the formatted text on a VDT screen or on a typewriter-like printer. It approximates the final output except that:

- You cannot see point-size changes.
- You cannot see font changes.
- Titles that are too long appear proper.
- All horizontal motions are reduced to one horizontal space to the right.
- All vertical motions are reduced to one vertical space down.

For example, it will appear that lines of text following a **.B**, **.C**, or **.D** macro are not aligned properly (even though they will be in the typeset output).

Although you cannot determine alignment from this approximation, you can observe line breaks and the amount of vertical space required by the text. If the foil is not full, the macro package prints the number of blank lines (in the current point size) that remain on the foil; if the foil is full, **mv** prints a warning. If the text overflows the foil, **mv** prints the text after the cross hairs.

Names Reserved by mv

The **mv** macro package uses certain names internally. All 2-character names starting with either "**)**" or "**]**" are reserved. Experienced users of the UNIX system or the **mv** macro package may want to define strings, or write additional macros. You cannot use names that are the same as those of the **mv** macros, strings that are described in this section of the tutorial, or names that are the same as **troff** names. Furthermore, if you use any of the preprocessors, you also must avoid their reserved names.

Miscellaneous Information

The .S macro changes the point size and vertical spacing immediately, but a line-length change requested with that macro does not take effect until the next level-macro call. Specifying a third argument to the .S macro usually results in a disaster.

The *(Tm string generates a trademark symbol.

The tilde (~) is interpreted by the **mv** macros as an unpaddable space; that is, the tilde may be used wherever you desire a fixed-size (non adjustable) space. To override this condition, the following line should be included in the input file:

```
.tr ~~
```

Dimensional Details

For each size of viewgraph, the following table shows the default point size, the maximum number of lines of text (at the default point size), and the height, width, and aspect ratio, both nominal and actual.

Macro (Note 1)	Point Size	Maximum Lines (Note 2)	Nominal				Actual (TEXT)			
			W	H	AR	1/AR	W	H	AR	1/AR
			— (Note 3) —				— (Note 3) —			
.VS	18	21	7	7	1	1	6	6.8	1.13	.88
.Vw	14	19	7	5	.71	1.4	6	4.8	.8	1.25
.Vh	14	27	5	7	1.4	.71	4.2	6.8	1.6	.62
.VW	14	21	7	5.4	.77	1.3	6	5.2	.87	1.15
.VH	18	28	7	9	1.3	.77	6	8.8	1.5	.68
.Sw	14	18	7	4.6	.67	1.5	6	4.4	.73	1.4
.Sh	14	27	4.6	7	1.5	.67	3.8	6.8	1.8	.56
.SW	14	18	7	4.6	.67	1.5	6	4.4	.73	1.4
.SH	18	28	6	9	1.5	.67	5	8.8	1.76	.57

NOTE

If used as viewgraphs, the .SW macro and .VW macro generated foils must be enlarged by a factor of 9/7.

Maximum number of lines of text at the default point size.

W-Width in inches, H-Height in inches, AR-Aspect Ratio (H/W).

The Production of Viewgraphs and Slides

The phototypesetter produces output on mechanical paper, which is white, opaque, photographic paper. There are several simple processes (for example, Thermofax, Bruning) for making transparencies from opaque paper. Because some of these processes involve heat, and because mechanical paper is heat sensitive, you should first make copies of the phototypesetter output on a good-quality office copier, and then use these copies for making your transparencies.

Making slides is a more complicated photographic process than making viewgraphs. It is possible to make both positive (opaque letters on transparent background) and negative (transparent letters on opaque background) slides, as well as colored-background slides, etc. It is probably best to consult a professional in cases like these.

General Guidelines for Using mv

- The most useful foil sizes are **.VS** and **.Vw** (or **.Sw**). This is because most projection screens are either square or wider than they are tall, and also because the resulting foils are smaller, easier to carry, and require no enlargement before use.
- You should avoid reducing point size below the default value. Default point size for each type of foil (Table 3) is the smallest point size that produces a foil that is legible by an audience of more than a dozen people. If there is more text than will comfortably fit on one foil, you should use two or more foils instead of reducing the point size.
- You should avoid numerous font changes. A foil with more than two typefaces looks cluttered and distracts the viewer.
- You should avoid underlined typeset text. Even though this package contains a macro for underlining, you probably should not use it. Underlined typeset text almost always looks bad; instead, use a different typeface.
- The Helvetica sans-serif font is thicker and easier to read than the Times Roman serif font normally used for typesetting. On the other hand, the Times Roman font permits more text to be squeezed onto a foil. If you intend to use italic and/or bold typefaces, either the Helvetica regular, italic, and medium (which is actually a bold typeface):

```
.DF 1 H 1 HI 3 HM
```

or the Times Roman regular, italic, and bold:

```
.DF 1 R 2 I 3 B
```

should be mounted via the **.DF** macro. Bold typefaces tend to be a bit overwhelming. Choice of fonts is primarily a matter of personal choice.

- You can use the **.SP** macro to insert a bit of additional white space (for instance, .5v or 1v) at the top of each foil (that is, increase the top margin).
- Normal upper-case and lower-case text is more legible than upper-case text only. (In smaller point sizes, however, lower-case characters may be hard to read.) Upper-case and lower-case alphabets have evolved because they result in more legible text. Furthermore, such text is less bulky than upper-case text only, so you can put more information onto a foil without crowding.
- You should make foils for a presentation as consistent as possible. Changing fonts, typefaces, point sizes, etc., from foil to foil tends to distract the viewer. While it is possible to emphasize and draw the viewer's attention to particular items with such changes, this works only if it is done purposefully and sparingly. Overuse of these techniques is almost always counter-productive.

In summary, the dictum that "the medium is the message" does not apply to foil making. When in doubt:

- Do not change point sizes.
- Do not change fonts or typefaces.
- Do not underline.
- Use many sparse foils rather than a few dense ones.
- Use fewer words rather than more words.
- Use larger point sizes rather than smaller point sizes.

- Use larger top and bottom margins rather than smaller ones.
- Use normal upper-case and lower-case text rather than upper-case text only.

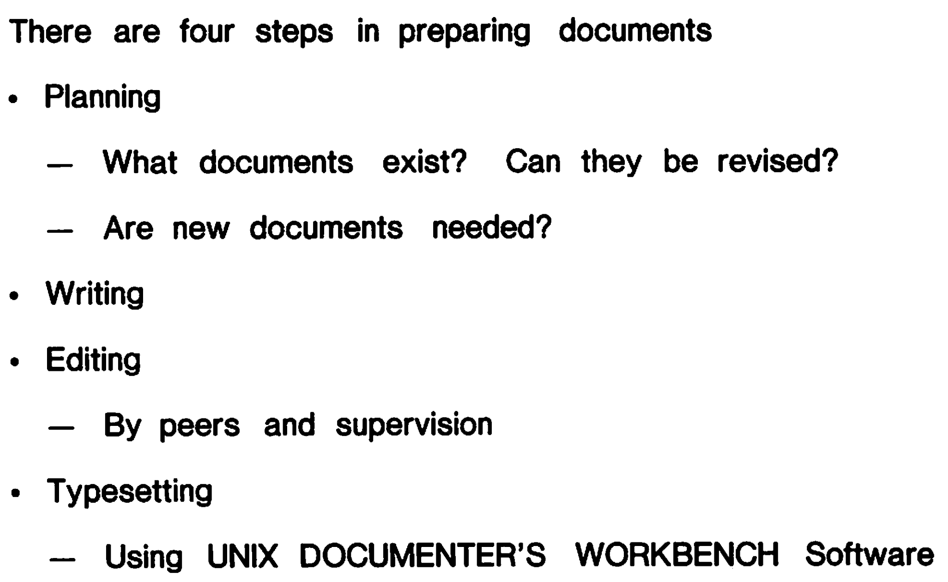
The mv Sampler

This Sampler contains the formatted output of the examples shown in this tutorial. The output has been reduced in order to fit the page size of this guide.

The Primary Steps in Preparing Documents

- Planning
- Writing
- Editing
- Typesetting

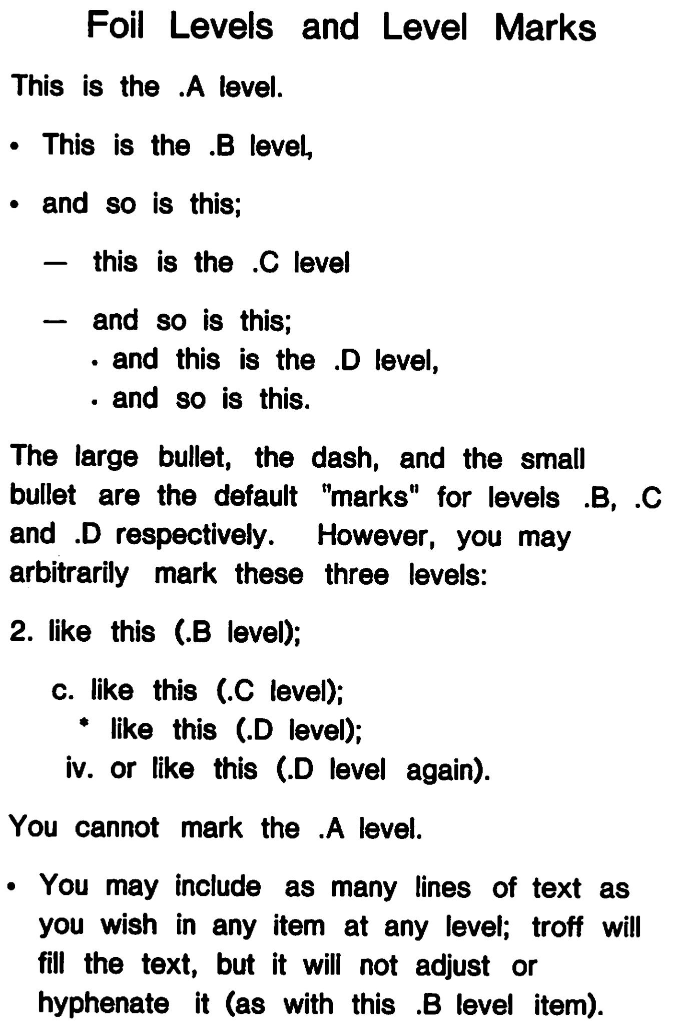
Example 1: A Simple Transparency



There are four steps in preparing documents

- Planning
  - What documents exist? Can they be revised?
  - Are new documents needed?
- Writing
- Editing
  - By peers and supervision
- Typesetting
  - Using UNIX DOCUMENTER'S WORKBENCH Software

Example 2: A More Elaborate Transparency



Foil Levels and Level Marks

This is the .A level.

- This is the .B level,
- and so is this;
  - — this is the .C level
  - — and so is this;
    - · and this is the .D level,
    - · and so is this.

The large bullet, the dash, and the small bullet are the default "marks" for levels .B, .C and .D respectively. However, you may arbitrarily mark these three levels:

2. like this (.B level);

   c. like this (.C level);

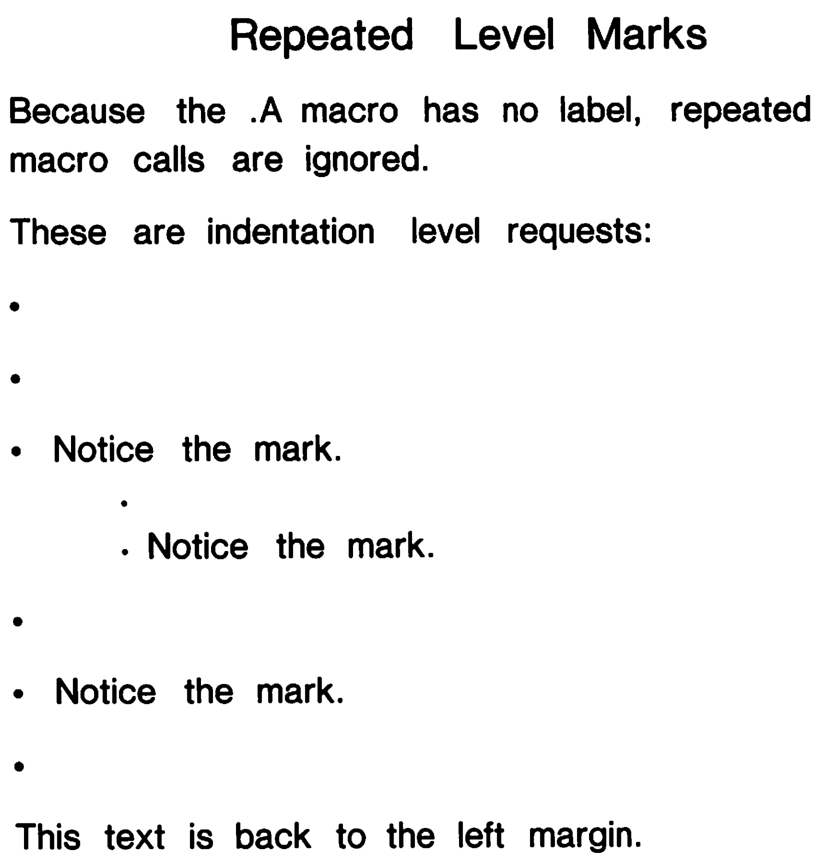
     * like this (.D level);

   iv. or like this (.D level again).

You cannot mark the .A level.

- You may include as many lines of text as you wish in any item at any level; troff will fill the text, but it will not adjust or hyphenate it (as with this .B level item).

Example 3: The Level Macros



Repeated Level Marks

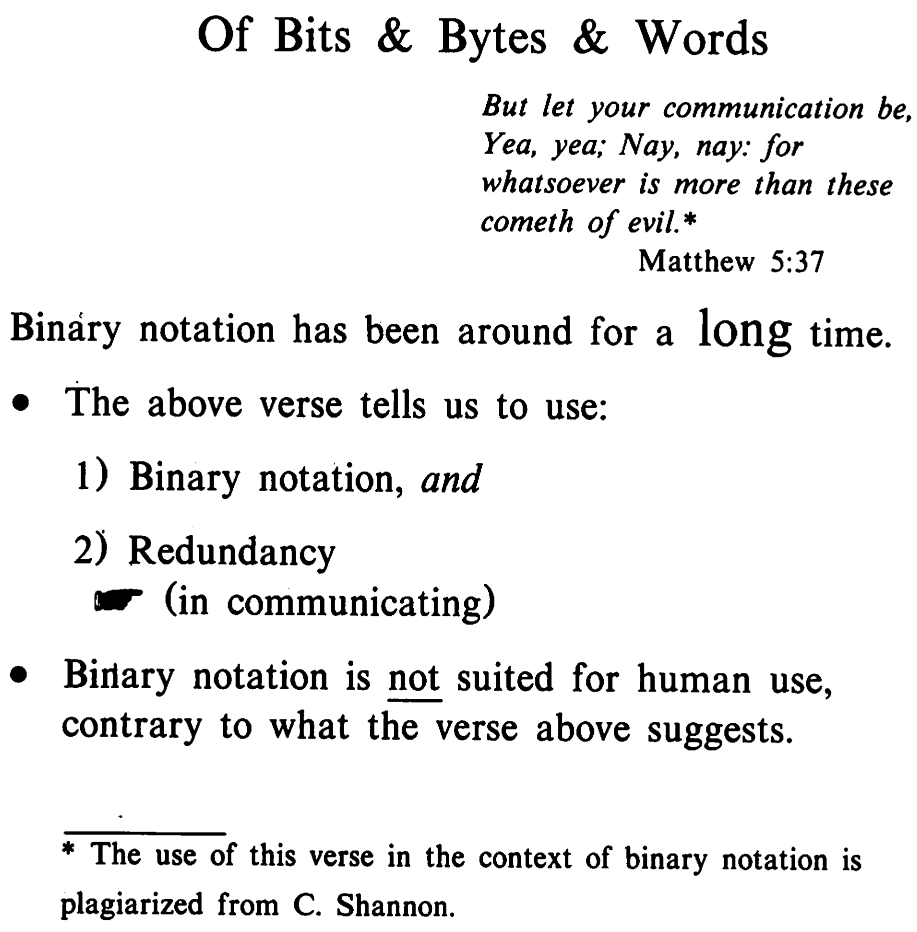
Because the .A macro has no label, repeated macro calls are ignored.

These are indentation level requests:

-
-
- Notice the mark.
  -
  - Notice the mark.
-
- Notice the mark.
-

This text is back to the left margin.

Example 4: Repeated Level Marks



Of Bits & Bytes & Words

> *But let your communication be,*
> *Yea, yea; Nay, nay: for*
> *whatsoever is more than these*
> *cometh of evil.**
> Matthew 5:37

Binary notation has been around for a long time.

- The above verse tells us to use:

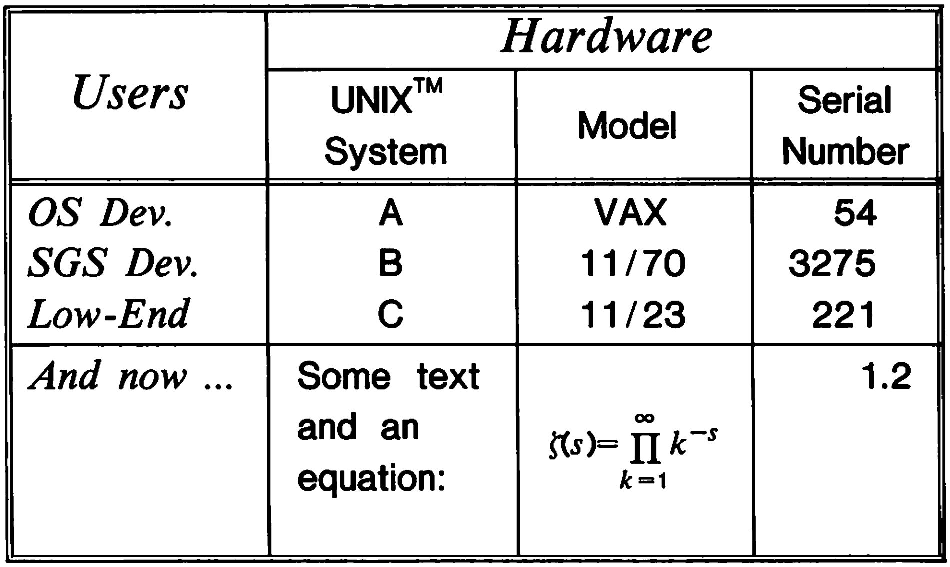
  1) Binary notation, *and*

  2) Redundancy
  ☛ (in communicating)

- Binary notation is <u>not</u> suited for human use, contrary to what the verse above suggests.

* The use of this verse in the context of binary notation is plagiarized from C. Shannon.

Example 5: A Complicated Transparency



Users	*Hardware*		
	UNIX™ System	Model	Serial Number
OS Dev. *SGS Dev.* *Low-End*	A B C	VAX 11/70 11/23	54 3275 221
And now ...	Some text and an equation:	$\zeta(s) = \prod_{k=1}^{\infty} k^{-s}$	1.2

Example 6: A Transparency with Tables and an Equation

Index

Table of Contents

Introduction

This tutorial is intended to give you a working knowledge of **checkmm**, **diffmk**, **hyphen**, **ndx**, **ptx**, and **subj**. When you have prepared a document draft, **hyphen** will help you to proofread it. **checkmm** will check your macro usage for possible errors. **diffmk** will make later revisions of your work more manageable. When you are satisfied that you have produced a polished document, **subj**, **ndx**, and **ptx** will help you to make an index.

You should be familiar with the following concepts and tools to fully benefit from this tutorial.

- You should know how to use a UNIX System V text editor (**ed**, **vi**, and **ex** are examples). See the *UNIX System V User Guide*.
- You should know what a file and directory are and know how to manipulate them. See the *UNIX System V User Guide.*
- You should know how to redirect input and output using pipes. See the *UNIX System V User Guide.*
- You should be familiar with a UNIX System V formatter (**nroff** or **troff**). See the tutorials in this book that discuss **nroff** or **troff**.
- You should be familiar with the **mm** macro package in order to understand the explanation of **checkmm**. See the tutorial in this book that discusses **mm**.

Checking your File

This section shows you how to check your file before it is formatted or sent to a printer. The tools discussed here will help you catch formatting errors before you print, saving time, paper, and computing resources.

Using checkmm

checkmm examines your file to identify potential **mm** usage errors. For example, if you used **.DS** to start a display but forgot to close the display with .DE, **checkmm** will find the mistake. If you prepared a formal memorandum and began it with inappropriate requests or macros, **checkmm** will complain.

checkmm is easy to use: you simply type

```
checkmm my.file
```

and errors (providing you have any) will come up on your screen. If you'd like a more permanent record of these errors, type

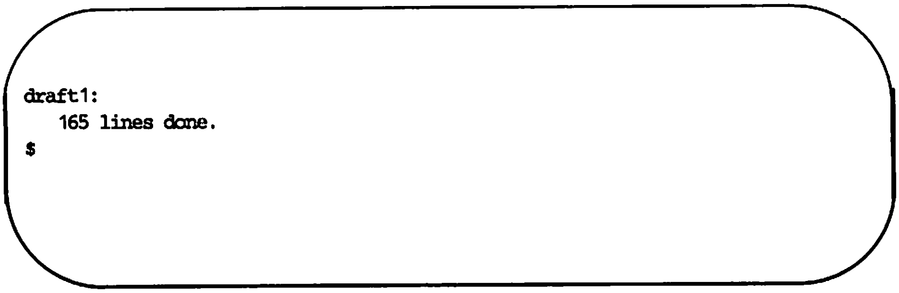
```
checkmm my.file > errors
```

and read the contents of the file, **errors**, for **checkmm**'s error messages.

If you have no errors **checkmm** will tell you so. For instance, if you had typed 165 lines into the file **draftl** you could check it for **mm**-correctness as follows. Assuming there were no **mm** errors, typing

```
checkmm draftl
```

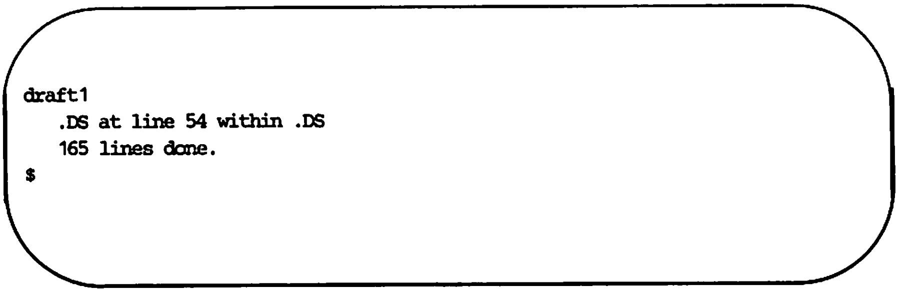
would cause the following message to print across your screen:



```
draft1:
   165 lines done.
$
```

That is, "No errors."

But if you had forgotten to close a display with a .DE macro, you might have gotten the following message:

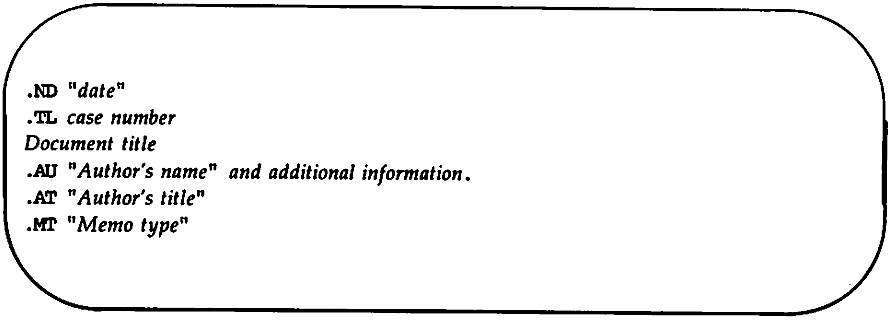


```
draft1
   .DS at line 54 within .DS
   165 lines done.
$
```

In this case, **checkmm** read **draftl,** saw two **.DS** macros and no intervening .DE macro, and complained that you were trying to put a display inside a display (an illegal action in **mm**). In other words, it saw that you had forgotten to end the first display.

This example suggests that **checkmm** is inclined to give you the benefit of the doubt. It will take note of a possible error, but only when it has processed to the point where something must be wrong will it complain. Another example of this is its handling of formal memoranda macros.

Here are the macros you would use to begin a memo:

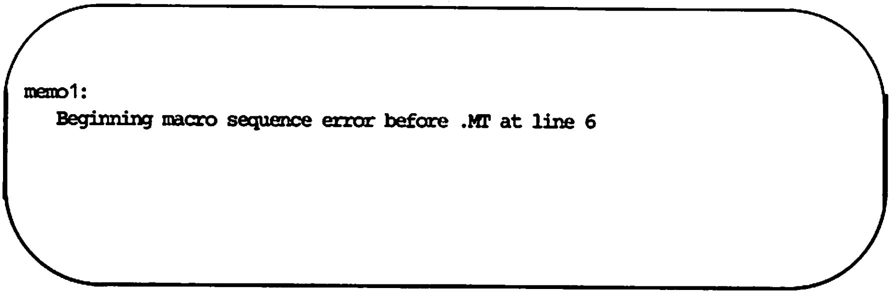


```
.ND "date"
.TL case number
Document title
.AU "Author's name" and additional information.
.AT "Author's title"
.MT "Memo type"
```

checkmm will accept the absence of unessential macros like .ND and .AT, or it will allow you to intersperse other requests, such as those for adjusting line length or indent, for centering text, and so on. But it will complain if you enter any of these macros out of the order shown above. Had you, for example, entered the .ND macro after the .TL in a file called **memo1**, then typing

checkmm memo1

would produce these results:



```
memo1:
   Beginning macro sequence error before .MT at line 6
```

checkmm reads through to the .MT macro before deciding you've committed an error; you must search upward in the file from this point to find what caused the actual error.

Proofreading with hyphen

nroff and **troff** provide you with a fine level of control over hyphenation. You can make precise decisions about the way words break across the boundary from the end of one line to the beginning of the next. As the "**nroff/troff** Technical Discussion" shows you (see the *Technical Discussion and Reference Manual*), you can turn on automatic hyphenation with the request **.hy**, you can turn off all hyphenation with the request **.nh**, and/or you can specify a selective version of automatic hyphenation with **.hw**. After your document is formatted, you might like to go back and see the results, that is, see all hyphenated words with the **hyphen** command.

hyphen is straightforward. You simply type

```
hyphen out.file
```

and you get a list of all words that have been hyphenated in the file you named on the command line. (Don't forget that only formatted files reflect **nroff**'s or **troff**'s automatic hyphenation.)

Here is another way to check your file for hyphenation:

```
mm −Tlp −rW72 memo1 | hyphen > memo1.hy
```

This command line formats the file and pipes the output (in this case, a file processed by **nroff** and **mm**, prepared for printing on the line printer, and specified to have lines of 72 characters) through **hyphen**. The results are sent to a file called **memo1.hy**. This last file doesn't contain the formatted document; it contains all words in that document that were hyphenated across a line break. To inspect the hyphenations that will appear in the document, you must give **hyphen** the same formatted file that you will send to the printer. Different formats—longer lines or fewer lines, for example—will result in different hyphenated words. **hyphen** ignores hyphenated numbers, such as phone numbers.

Revising with diffmk

diffmk is a valuable tool for comparing successive versions of a document. It allows you to see two things at once: the most up-to-date version of a document, and those lines that are different from the previous version.

When you are revising a document it is good practice to copy the original and edit the copy. Once these alterations are complete, you can see your changes against the background of the original with **diffmk**. By typing

```
diffmk original.file revised.file diffmk.file
```

you form **diffmk.file,** based on the differences from **original.file** and **revised.file.** Note that the first two files are not changed as a consequence of **diffmk**'s analysis.

When you format this **diffmk**'d version, using **nroff** or **troff,** a vertical bar in the margin will identify those lines you have revised. An asterisk in the margin will indicate places where lines of text have been removed. An unformatted version of **diffmk.file** is identical to **revised.file** with the addition of **.mc** (margin character) requests.

Making Indices with subj, ndx, ptx, and mptx

Replacing the tedious and error-prone manual task of making an index, DOCUMENTER'S WORKBENCH Software provides the tools, **subj** and **ndx**, for generating indices automatically. They are easy to use and are compatible with the DOCUMENTER'S WORKBENCH formatters and macro packages. **subj** analyzes your text for words that appear to be subjects, and **ndx** produces an index of those words and the page numbers on which they appear.

To use these tools, you should know how to use a UNIX text editor, and you should be familiar with DOCUMENTER'S WORKBENCH Software command lines. (See the table at the conclusion of the *User's Guide* "Preface.")

The Subject-List Maker: subj

subj decides which words in your file are keywords: the ones that suggest what your file is about. It looks for capitalized words, assuming they are proper nouns, and for modifier-noun sequences. It also pays special attention to the words you use in abstracts, headings, introductory paragraphs, and the topic sentence of each paragraph.

subj remembers these keywords and examines your document in increments of two-sentence sections (first and second, second and third, and so forth) attempting to tie relevant keywords together. In this way function words and phrases, such as prepositional phrases, don't isolate potential subjects and subject-modifiers from each other.

Perhaps you can see already that **subj** has its own expectations of your writing. It assumes that each sentence begins on a new line, so it can disregard capitalized words that begin sentences. It assumes that you will present your text in neat and relevant blocks of information, and that these blocks will have headings or subheadings. Finally, it expects you to observe the classic thesis essay form: thesis sentence or sentences at the beginning of the text and a topic sentence at the start of each paragraph. Should you go off on a tangent or include an irrelevant name, you might have a subject-list that better reflects a document's problems of exposition than its real purpose or thesis. Normally you would want to edit a subject-list before using it, but first let's see how to produce one.

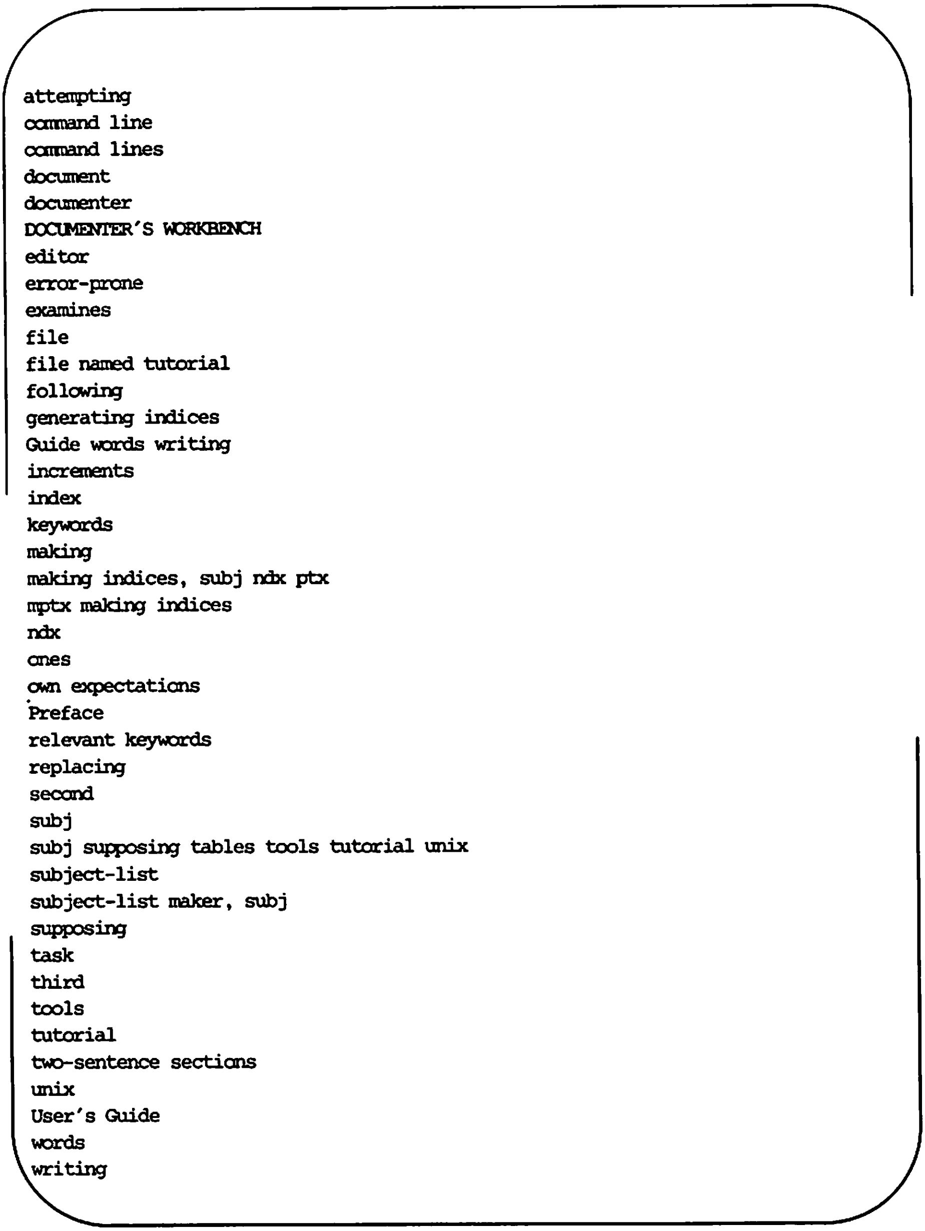
The command line you would type to use **subj** is the following:

```
subj my.file
```

Supposing you wanted a subject-list of this section of the tutorial (stored in a file named **tutorial**), you would type

```
subj tutorial
```

which would produce the following output:



```
attempting
command line
command lines
document
documenter
DOCUMENTER'S WORKBENCH
editor
error-prone
examines
file
file named tutorial
following
generating indices
Guide words writing
increments
index
keywords
making
making indices, subj ndx ptx
mptx making indices
ndx
ones
own expectations
Preface
relevant keywords
replacing
second
subj
subj supposing tables tools tutorial unix
subject-list
subject-list maker, subj
supposing
task
third
tools
tutorial
two-sentence sections
unix
User's Guide
words
writing
```

Notice that it recorded both headings in full and that it focused on the opening sentence, mentioning the **mm** tutorial. Had **mm** been mentioned down in the middle of a paragraph, it would have been ignored. Thus, presenting unimportant information in the opening paragraph of an essay or report will be given disproportionate importance in the subject-list. (In fact, this list suggests that the file contains information about **mm**. But although the file mentions **mm**, it is not the topic of the file, and might be deleted from the final list.)

subj works especially well with files that contain requests and macros. Knowing how these formatting commands are normally used, **subj** takes cues from text prepared with DOCUMENTER'S WORKBENCH Software that conventional text cannot offer. This does not compromise the other criteria **subj** uses to choose subjects, but rather complements them. While a subject-list might be valuable in a variety of ways, making indices is probably its most useful function.

Indexing Documents with ndx

The DOCUMENTER'S WORKBENCH Software index maker, **ndx**, reads the subject-list file, compares the words in it to the text, and records the subjects it finds and their corresponding page numbers. Because **ndx** focuses on the roots of words it finds in the subject-list, tenses and cases of words pose no problems to it. The following is an example of **ndx** at work. Notice that it uses a subject file, which you can make with **subj**:

```
ndx subject.file "mm –Tlp –rW72 my.file" > index.file
```

subject.file must contain a list of subjects drawn from the text or book you're indexing. You can use **subj** to make this list or you can select another method. It makes no difference to **ndx**. Next you give the command line you would use to generate the document you're indexing. Enclose it in double quotes. A few things should be said about using **ndx**:

- You are not actually sending anything to a printer. **ndx** makes a copy of the document, which you never see, then removes it.
- Be sure to give the same command line to **ndx** that you use when you format your document. **ndx** needs this information to make decisions about page numbers. A command line given to **ndx** that differs from the command line used to print the document will probably produce an inaccurate index.

- **ndx** will accept any DOCUMENTER'S WORKBENCH Software formatter with or without the **mm** macro package. The formatting command line must be enclosed in double quotes.
- **ndx** will not accept command line operators such as |, >, <, and &. Consequently, command lines that use preprocessors (e.g., **grap**, **pic**, **tbl**, and **eqn**) are disallowed on the **ndx** command line.

Should you need to index files that use one or more preprocessors, run the appropriate preprocessor before indexing and formatting:

```
grap my.file | pic | tbl | eqn > preprocessed.file
```

Then give the preprocessed file to **ndx**:

```
ndx subject.file "troff −mm preprocessed.file" > index.file
```

Last, make sure you use the same preprocessed file when printing:

```
troff −mm preprocessed.file | typesetter
```

The file, **index.file,** contains the finished index complete with alphabetical topic entries and corresponding page numbers.

Because **ndx** is intelligent enough to identify word-roots, you may find that it is too permissive in certain cases. To ensure that **ndx** literally matches a particular entry in your subject-list, begin the line on which the entry appears with a tilde (˜):

```
˜hyphenation
˜revision
˜preprocessing files
```

ndx will find only precise matches for these words. The tilde will not, however, identify phrases to be literally matched.

Given the brief subject list above, **ndx** would find "selection of hyphenated words" to be a miss but would identify "hyphenation" as a hit. You'll notice in the last example that each word following the tilde is designated as a literal entry. Thus, "preprocessed files" would not be matched, but "files that need preprocessing" would be. **ndx** has no provision for selectively isolating single words on a line. Either each word on the line is treated as a literal entry or none of them are.

Making Permuted Indices with ptx, mptx, and subj

Besides the subject/page-number index made with **ndx**, DOCUMENTER'S WORKBENCH Software has commands to make another type of index: the permuted index, made with **ptx** and **mptx**. Like a concordance, a permuted index is a catalog of keywords taken in their immediate contexts. The index is called "permuted" because it changes the word order, placing the keyword in the middle of the line. Index lines are presented alphabetically by keyword.

ptx, together with its eight options, reads the file you wish to index, selects keywords and contextual words, and prepares a file that will be processed to make the final index. **mptx** is a specialized macro package that interprets the file that **ptx** prepares.

The simplest statement of **ptx** use is as follows:

```
ptx my.file index.file
```

This gives **ptx**, respectively, the file you want it to read and the name of a file it can put its output in. Next, you simply give the output from **ptx** to the **mptx** macro package and an appropriate formatter to print your permuted index:

```
mm −mptx index.file > permuted.index
```

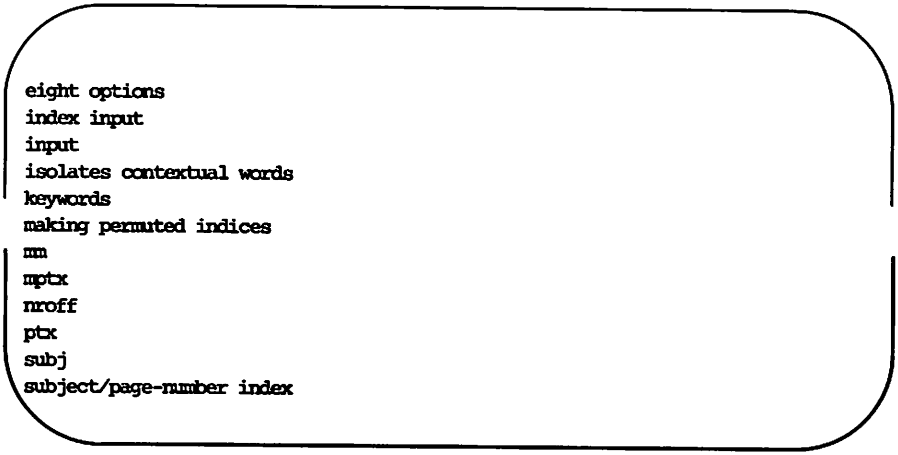
An appropriate formatter is one that will process whatever headings, titles, or other features you want to include with your permuted index. Remember, your permuted index is made from a formatted document. In the preceding case, the index was processed with **mptx** and **mm** (which invoked **nroff**).

ptx's default selection of keywords is generous, and the index produced by the above examples would probably be too large to be useful. You will probably want to give **ptx** a limited subject-list before it begins its work. You might, therefore, want to begin your session of indexing by using **subj**. (See the section of this tutorial that discusses **subj**.)

Assuming you wanted to make a permuted index of this section of the tutorial, you would proceed as follows. First, you would make a subject-list:

subj tutorial > subj.file

An edited version of the file, **subj.file,** is shown in the following figure.



```
eight options
index input
input
isolates contextual words
keywords
making permuted indices
mm
mptx
nroff
ptx
subj
subject/page-number index
```

Then, you would use **ptx** with its **−o** option:

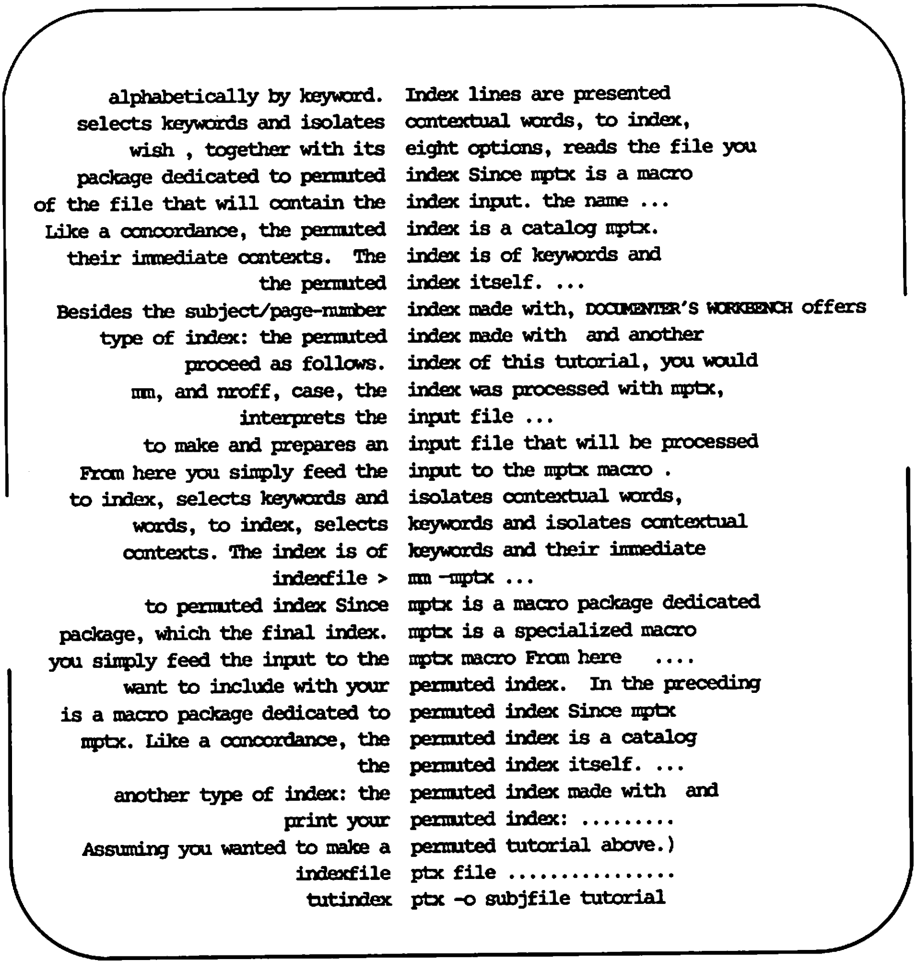
ptx −o subj.file tutorial tut.index

The **−o** option signifies "only," and is followed by an "only file." That is, only the words in the file following **−o** will be used by **ptx** as keywords. The file, **tutorial,** here is a formatted file. Once **ptx** has been given a subject-list, it expects to receive a formatted file, so it can get down to the business of extracting all keywords, including those in defined strings and defined macros.

Finally, you would process the file, **tut.index**, to produce the permuted index itself.

mm −mptx tut.index > permuted.index

The following is the contents of the file, **permuted.index.**



```
        alphabetically by keyword.  Index lines are presented
     selects keywords and isolates  contextual words, to index,
           wish , together with its  eight options, reads the file you
     package dedicated to permuted  index Since mptx is a macro
 of the file that will contain the  index input. the name ...
     Like a concordance, the permuted  index is a catalog mptx.
      their immediate contexts. The  index is of keywords and
                      the permuted  index itself. ...
   Besides the subject/page-number  index made with, DOCUMENTER'S WORKBENCH offers
        type of index: the permuted  index made with  and another
                 proceed as follows.  index of this tutorial, you would
           mm, and nroff, case, the  index was processed with mptx,
                     interprets the  input file ...
            to make and prepares an  input file that will be processed
      From here you simply feed the  input to the mptx macro .
     to index, selects keywords and  isolates contextual words,
           words, to index, selects  keywords and isolates contextual
          contexts. The index is of  keywords and their immediate
                        indexfile >  mm -mptx ...
            to permuted index Since  mptx is a macro package dedicated
    package, which the final index.  mptx is a specialized macro
   you simply feed the input to the  mptx macro From here    ....
          want to include with your  permuted index.  In the preceding
   is a macro package dedicated to  permuted index Since mptx
    mptx. Like a concordance, the  permuted index is a catalog
                               the  permuted index itself. ...
        another type of index: the  permuted index made with  and
                         print your  permuted index: .........
     Assuming you wanted to make a  permuted tutorial above.)
                          indexfile  ptx file ................
                           tutindex  ptx -o subjfile tutorial
```

The index's keywords are the alphabetically ordered, left adjusted words at the center. Contextual words read from left to right. Notice that the contextual phrase may begin at the far left, or it may begin at with the keyword itself wrapping around, on the same line, to the left. This word order depends on the syntax of the original sentence. If the keyword occurs early in the sentence, the context is likely to wrap in the index.

The permuted index used line adjustment (**.ad**) to make its contents easier to read. The left side is right adjusted, and the right side is left adjusted.

NOTE

For a complete listing of **ptx** and **mptx** options, see their manual entries in the *Technical Discussion and Reference Manual.*

Index