UNIX™ System V

DOCUMENTER'S WORKBENCH™ Software
Preprocessor Reference

Western Electric

January 1984
307-153 Issue 1

UNIX™ System V Release 2.0

DOCUMENTER'S WORKBENCH™ Software Preprocessor Reference

Western Electric

CONTENTS

Chapter 1

INTRODUCTION

This book is a guide and reference manual for the text preprocessors that are provided in the UNIX* System DOCUMENTER'S WORKBENCH† software. This system provides an integrated set of text processing tools for easy, flexible, and professional documentation production. Books that describe other aspects of the DOCUMENTER'S WORKBENCH software are:

- *Introduction and Reference Manual*—Select Code 307-150
- *Text Formatters Reference*—Select Code 307-151
- *Macro Packages Reference*—Select Code 307-152

Each of the chapters in this book is a user guide to a specific text preprocessor. Information is provided in each chapter that will allow the user to understand and use the preprocessors. Numerous examples are included that will provide the user a base to build on when learning to use the preprocessors. The beginning user should refer to the DOCUMENTER'S WORKBENCH software *Introduction and Reference Manual* for a better overall description of the text processing tools available on the UNIX system.

1. Using the Preprocessors

A preprocessor allows a user to produce complicated formatted output such as tables and pictures from an input language that is easier to use than the formatter language. For example, a complicated, multi-column table that is boxed on all sides, with each item properly aligned and boxed can be produced with only a few lines of input text and the table data using the **tbl** preprocessor. To produce the same output using only the formatter requests would be much more difficult and time consuming.

* UNIX is a trademark of Bell Laboratories.

† DOCUMENTER'S WORKBENCH is a trademark of Western Electric Company.

To use the preprocessors, the unformatted text file is written using macros or formatter requests with the input destined for the preprocessor set off by delimiting macros or characters. The preprocessor works on the unformatted file first, replacing the text between the delimiters with the text formatter requests that produce the desired output. The output from the preprocessor is then processed by a text formatter. Normally, this is done using the piping mechanism of the UNIX system. For example:

```
tbl file | eqn | troff
```

would be the command line used to format with troff a file containing tables and equations. Note that several preprocessors can be used in the same process because each has its own language and each one only expands input found between its own delimiters.

2. Preprocessors Covered

The following preprocessors are covered in this book.

- Chapter 2—*Table Formatting Program* (tbl)
- Chapter 3—*Picture Graphics Language* (pic)
- Chapter 4—*Mathematics Typesetting Program* (eqn)

Chapter 2

TABLE FORMATTING PROGRAM

PAGE

Chapter 2

TABLE FORMATTING PROGRAM

1. Introduction

The **tbl** program is a document formatting preprocessor for the **nroff** and **troff** formatters that makes fairly complex tables easy to specify and enter. Tables consist of columns which may be independently centered, right-adjusted, left-adjusted, or aligned by decimal points. Headings may be placed over single columns or groups of columns. A table entry may contain equations or consist of several rows of text. Horizontal or vertical lines may be drawn as desired in the table, and any table or element may be enclosed in a box.

A description of a table is translated by the **tbl** program into a list of **nroff/troff** formatter requests that will produce the table. The **tbl** program isolates a portion of a job that it can successfully handle (text between the **.TS** and **.TE** delimiting macros) and leaves the remainder for other programs. Thus, **tbl** may be used with the equation formatting program (**eqn**), the graphics formatting program (**pic**), and/or various formatter layout macro packages without function duplication.

2. Usage

On the UNIX system, the **tbl** program can be run on a simple table with the command

```
tbl filename|troff
```

When there are several input files containing tables, equations, pictures, and *mm* macro requests, the normal command is

```
tbl file1 file2...|eqn|troff -mm
```

The usual options may be used on the **troff** formatter. Usage of the **nroff** formatter is similar to that of **troff**. If a file name is "−", the standard input is read at that point.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special *-TX* command-line option to **tbl** which produces output that does not have fractional line motions.

When both **tbl** and **eqn** programs operate on the same file, **tbl** should be called first. If there are no equations within tables, either sequence works. It is usually faster to execute **tbl** first since **eqn** normally produces a larger expansion of the input. However, if there are equations within tables (using the *delim* statement in **eqn**), **tbl** must be executed first or the output will be scrambled. Use of equations in **n**-style (numeric) columns should be avoided since **tbl** attempts to split numerical format items into two parts. The *delim (xy)* global option prevents splitting numerical columns within delimiters. For example, if the **eqn** delimiters are "$$", a *delim ($$)* statement causes a numerical column such as

```
1245 $± 16$
```

to be divided after 1245, not after 16.

The **tbl** program accepts up to 35 columns; the actual number that can be processed may be smaller depending on availability of **troff** formatter number registers. Number register names used by **tbl** must be avoided within tables. These include 2-digit numbers from 31 to 99 and strings of the form 4*x*, 5*x*, #*x*, *x*+, *x*|, ^*x*, and *x*-, where *x* is any lowercase letter. The names ##, #-, and #^ are also used in certain circumstances. To conserve register names, the **n** and **a** key letters (key letters are introduced in paragraph 3.2) share a register. Hence, the restriction that they may not be used in the same column.

As an aid in writing layout macros, **tbl** defines a number register *TW* which is the table width. The *TW* number register is defined by the time that the **.TE** macro is invoked and may be used in the expansion of that macro. More importantly, to assist in laying out multipage boxed tables, the macro **T#** is defined to produce the bottom lines and side lines of a boxed table and then be invoked at its end. By use of this macro in the page footer, a multipage table can be boxed. In particular, thd *mm* macros can be used to print a multipage boxed table with a repeated heading by giving the argument *H* to the **.TS** macro. If the table start macro is written

```
    .TS H
```

then, a line of the form

```
    .TH
```

must be given in the table after any table heading (or at the start if none). Material up to the **.TH** is placed at the top of each page of the table. The remaining lines in the table are placed on several pages as required. This is not a feature of **tbl** but of the *mm* macros.

3. Input Commands

Input to **tbl** is text for a document with tables preceded by a **.TS** (table start) command and followed by a **.TE** (table end) command. The **tbl** program processes the tables, generates formatting requests, and leaves the text unchanged. The **.TS** and **.TE** lines are copied so that **troff** formatter layout macros (such as memorandum formatting macros) can use these lines as delimiters. Arguments on the **.TS** or **.TE** lines are copied, but otherwise ignored, and may be used by document layout macro requests.

The general format of the input is

```
text
.TS
table
.TE
text
.TS
table
.TE
text
```

The format of each table is

```
.TS
options;
format.
data
.TE
```

Each table is independent and contains:

- Global options {3.1}
- A format section describing individual columns and rows of the table {3.2}
- Data to be printed {3.3}.

The format section and data are always required but not the options.

3.1 Global Options

There may be a single line of options affecting the whole table. If present, this line must immediately follow the .TS line and must contain a list of option names separated by spaces, tabs, or commas and must be terminated by a semicolon. Allowable options are:

- **center** - center table (default is left-adjust)
- **expand** - make table as wide as current line length
- **box** - enclose table in a box
- **allbox** - enclose each item of table in a box
- **doublebox** - enclose table in two boxes
- **tab** (*x*) - separate data items by using *x* instead of tab
- **linesize** (*n*) - set lines or rules (e.g., from **box**) in *n*-point type
- **delim** (*xy*) - recognize *x* and *y* as **eqn** delimiters.

The **tbl** program tries to keep boxed tables on one page by issuing appropriate **.ne** (need) requests. These requests are calculated from the number of lines in the tables. If there are spacing requests embedded in the input, the **.ne** requests may be inaccurate. Normal **troff** formatter procedures, such as keep-release macros, are used in that case. If a multipage boxed table is required, macros designed for this purpose (.TS H and .TH) should be used.

3.2 Format Section

The format section of the table specifies the layout of the columns. Each line in the format section corresponds to one line of table data (except the last format line corresponds to all following data lines up to any additional **.T&** command line). Each format line contains a **key letter** for each column of the table. Key letters may be separated by spaces or tabs for readibility purposes. Key letters are:

L or **l**	Indicates a left-adjusted column entry.
R or **r**	Indicates a right-adjusted column entry.
C or **c**	Indicates a centered column entry.
N or **n**	Indicates a numerical column entry. Numerical entries are aligned so that the units digits of numbers line up.
A or **a**	Indicates an alphabetic subcolumn. All corresponding entries are aligned on the left and positioned so that the widest entry is centered within the column.
S or **s**	Indicates a spanned heading. The entry from the previous column continues across this column (not allowed for the first column of the table).
^	Indicates a vertically spanned heading. The entry from the previous row continues down through this row (not allowed for the first row of the table).

When numerical column alignment (**n**) is specified, a location for the decimal point is sought. The rightmost dot (**.**) adjacent to a digit is used as a decimal point. If there is no dot adjoining a digit, the rightmost digit is used as a units digit. If no alignment is indicated, the item is centered in the column. However, the special nonprinting character string **\&** may be used to override dots and digits or to align alphabetic data. This aligns the dots and the **\&** disappears from the final output.

In the following example, items shown in the **INPUT** column will be aligned (in a numerical column) as shown in the **OUTPUT** column.

```
INPUT:               OUTPUT:
.TS
center;
n.
13                        13
4.2                        4.2
26.4.12                 26.4.12
abcdefg               abcdefg
abcd\&efg              abcdefg
abcdefg\&           abcdefg
43\&3.22                  433.22
749.12                  749.12
.TE
```

If numerical data are used in the same column with wider **L** (the capital **L** key letter is used instead of lowercase for readability) or **r** type table entries, the widest number is centered relative to the wider **L** or **r** items. Alignment within the numerical items is preserved. This is similar to the behavior of **a** type data. Alphabetic subcolumns (requested by the **a** key letter) are always slightly indented relative to **L** items. If necessary, the column width is increased to force this. This is not true for **n** type entries.

> ***Note:*** The **n** and **a** items should not be used in the same column.

The end of the format section is indicated by a period. The layout of key letters in the format section resembles the layout of the actual data in the table. Thus, a simple 3-column format might appear as

```
css
lnn.
```

The first line of the table contains a heading centered across all three columns. Each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data.

A sample table in this format is:

OVERALL TITLE		
Item-a	34.22	9.1
Item-b	12.65	.02
Item-c	23	5.8
Total	69.87	14.92

Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats may be given on the same line, separated by commas. The format for the above example could be written:

```
c s s, l n n.
```

Additional features of the key letter system are:

- ***Horizontal lines*** - A key letter may be replaced by underscore (_) to indicate a horizontal line in place of the column entry or equal (=) to indicate a double horizontal line. If an adjacent column contains a horizontal line or if there are vertical lines adjoining this column, the horizontal line is extended to meet nearby lines. If any data entry is provided for this column, it is ignored and a warning message is printed.

- ***Vertical lines*** - A vertical bar (|) placed between column key letters will cause a vertical line between the corresponding columns of the table. A vertical bar to the left of the first key letter or to the right of the last one produces a line at the edge of the table. If two vertical bars appear between key letters, a double vertical line is drawn.

- ***Space between columns*** - A number may follow the key letter indicating the amount of separation between this column and the next column. The number specifies the separation in *ens.* One *en* is about the width of the letter "n". More precisely, an *en* is the number of points (1 point = 1/72 inch) equal to half the current type size. If the *expand* option is used, these numbers are multiplied by a constant such that the table is as wide as the current line length. The default column separation number is 3. If the separation is changed, the worst case (largest space requested) governs.

- ***Vertical spanning*** - Vertically spanned items extending over several rows of the table are centered in their vertical range. If a key letter is followed by **t** or **T**, any corresponding vertically spanned item will begin at the top line of its range.

- ***Font changes*** - A key letter followed by a string containing a font name or number preceded by the letter **f** or **F** indicates that the corresponding column should be in a different font from the default font (usually Roman). All font names are one or two letters. A 1-letter font name should be separated from whatever follows by a space or tab. The single letters **B**, **b**, **I**, and **i** are shorter synonyms for **fB** and **fI**. Font-change requests given with the table entries override these specifications.

- ***Point size changes*** - A key letter followed by **p** or **P** and a number indicates the point size of the corresponding table entries. If the number is a signed digit, it is taken as an increment or decrement from the current point size. If both a point size and a column separation value are given, one or more blanks must separate them.

- ***Vertical spacing changes*** - A key letter followed by **v** or **V** and a number indicates the vertical line spacing used within a multiline table entry. The number may be a signed digit, in which case it is taken as an increment or decrement from the current vertical spacing. A column separation value must be separated by blanks or some other specification from a vertical spacing request. This request has no effect unless the corresponding table entry is a text block.

- ***Column width indication*** - A key letter followed by **w** or **W** and a width value in parentheses indicates minimum column width. If the largest element in the column is not as wide as the width value given after the **w**, the largest element is assumed to be that wide. If the largest element in the column is wider than the specified value, its width is used. The width is also used as a default line length for included text blocks. Normal **troff** formatter units can be used to scale the width value. The default value is *ens* if none are used. If the width specification is a unitless integer, the parentheses may be omitted. If another width value is given in a column, the last one controls the width.

- ***Equal-width columns*** - A key letter followed by **e** or **E** indicates equal-width columns. All columns whose key letters are followed by **e** or **E** are made the same width. This permits a group of regularly spaced columns.

- ***Staggered columns*** - A key letter followed by **u** or **U** indicates that the corresponding entry is to be moved up one-half line. This makes it easy to have a column of differences between numbers in an adjoining column. The *allbox* option does not work with staggered columns.

- ***Zero-width item*** - A key letter followed by **z** or **Z** indicates that the corresponding data item is to be ignored in calculating column widths. This may be useful in allowing headings to run across adjacent columns where spanned headings would be inappropriate.

- ***Default*** - Column descriptors missing from the end of a format line are assumed to be **L**. The longest line in the format section, however, defines the number of columns in the table. Extra columns in the data are ignored.

The order of the features is immaterial. They need not be separated by spaces except as indicated to avoid ambiguities involving point size and font changes. Thus, a numerical column entry in italic font and 12-point type with a minimum width of 2.5 inches and separated by 6 ens from the next column could be specified as

```
np12w(2.5i)fI 6
```

3.3 Data To Be Printed

Data for the table are input after the format section. Each table line is typed as one line of data. Very long input lines can be broken. Any line whose last character is a backslash (\) is combined with the following line; i.e., the backslash vanishes. Data for different columns (table entries) are separated by tabs or by whatever character has been specified in the **tab** global option {3.1}.

There are a few special cases of data entries:

- **troff** *commands within tables* - An input line beginning with a dot and followed by anything but a number (*.xx*) is assumed to be a request to the formatter and is passed through unchanged retaining its position in the table. For example, a space within a table may be produced with the **.sp** request in the data.

- *Full width horizontal lines* - An input line containing only the _ (underscore) character or = (equal sign) is taken to be a single or double line, respectively, extending the full width of the table.

- *Single column horizontal lines* - An input table entry containing only the _ character or the = is taken to be a single or double line extending the full width of the column. Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain these characters explicitly in a column, they should be preceded by a \**&** or followed by a space before the usual tab or newline character.

- *Short horizontal lines* - An input table entry containing only the string \_ is assumed to be a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

- *Repeated characters* - An input table entry containing only a string of the form \**R***x*, where *x* is any character, is replaced by repetitions of the character *x* as wide as data in the column. The sequence is not extended to meet adjoining columns.

- *Vertically spanned items* - An input table entry containing only the \\^ character string indicates that the table entry immediately above spans downward over this row. It is equivalent to a table format key letter of ^.

- *Text blocks* - In order to include a block of text as a table entry, precede it by **T{** and follow it by **T}**. Thus, the sequence

```
... T{
block of
text
T} ...
```

is the way to enter as a single entry in the table something that cannot conveniently be typed as a simple string between tabs. The **T}** (end delimiter) must begin a line. Additional columns of data may follow after a tab on the same line. Text blocks are pulled out from the table, processed separately by the formatter, and replaced in the table as a solid block.

Various limits in the **troff** program are likely to be exceeded if 30 or more text blocks are used in a table. This produces diagnostic messages such as "too many string/macro names" or "too many number registers".

If no line length is specified in the block of text or in the table format, the default is to use

$$L \times C / (N + 1)$$

where L is the current line length, C is the number of table columns spanned by the text, and N is the total number of columns in the table.

Other parameters (point size, font, etc.) used in typesetting the text block are:

(a) those in effect at the beginning of the table (including the effect of the **.TS** macro)

(b) any table format specifications of size, spacing, and font using the **p**, **v**, and **f** modifiers to the column key letters

(c) **troff** requests within the text block itself (requests within the table data but not within the text block do not affect that block).

Although any number of lines may be present in a table, only the first 200 lines are used in setting up the table. A multipage table may be arranged as several single-page tables if this proves to be a problem.

When calculating column widths, all table entries are assumed to be in the font and size being used when the **.TS** command was encountered. This is true except for font and size changes indicated in the table format section or within the table data (as in the entry **\s+3Data\s0**). Because arbitrary **troff** requests may be sprinkled in a table, care must be taken to avoid confusing width calculations. It is not possible to change the number of columns, the space between columns, the global options such as *box*, or the selection of columns to be made equal in width.

4. Additional Command Lines

To change the format of a table after many similar lines, as with subheadings or summarizations, the **.T&** (table continue) command is used to change column parameters. It is not recognized after the first 200 lines of a table. The outline of such a table input is

```
.TS
options;
format.
data
...
.T&
format.
data
.T&
format.
data
.TE
```

Using this procedure, each table line can be close to its corresponding format line.

5. Examples

Figures 2-1 through 2-6 are included to show input and output information that illustrate the basic concepts of the **tbl** program. The Ⓣ symbol in the input data represents a tab character. Although each figure has a title that indicates an option or feature, other examples of use may be gleaned from them. For instance, Figure 2-5 also indicates the requesting of bold type print in the format area.

INPUT:

```
.TS
box;
c c c
l l l.
LanguageⓉAuthorsⓉRuns on
.sp
_
FortranⓉManyⓉAlmost anything
Pl/1ⓉIBMⓉ360/370
CⓉBTLⓉ11/45,H6000,370
BLISSⓉCarnegie-MellonⓉPDP-10,11
IDSⓉHoneywellⓉH6000
PascalⓉStanfordⓉ370
.TE
```

OUTPUT:

Language	Authors	Runs on
Fortran	Many	Almost anything
PL/1	IBM	360/370
C	BTL	11/45,H6000,370
BLISS	Carnegie-Mellon	PDP-10,11
IDS	Honeywell	H6000
Pascal	Stanford	370

Figure 2-1. Table Using "box" Option

INPUT:

```
.TS
allbox;
c s s
c c c
n n n.
AT&T Common Stock
YearⓉPriceⓉDividend
1971Ⓣ41-54Ⓣ$2.60
2Ⓣ41-54Ⓣ2.70
3Ⓣ46-55Ⓣ2.87
4Ⓣ40-53Ⓣ3.24
5Ⓣ45-52Ⓣ3.40
6Ⓣ51-59Ⓣ.95*
.TE
* (first quarter only)
```

OUTPUT:

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

Figure 2-2. Table Using "allbox" Option

INPUT:

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

OUTPUT:

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

Figure 2-3. Table Using "vertical bar" Key Letter Feature

INPUT:

```
.TS
box;
L L L
L L _
L L | LB
L L _
L L L.
januaryⓉfebruaryⓉmarch
aprilⓉmay
juneⓉjulyⓉMonths
augustⓉseptember
octoberⓉnovemberⓉdecember
.TE
```

OUTPUT:

january	february	march
april	may	
june	july	**Months**
august	september	
october	november	december

Figure 2-4. Table Using Horizontal Lines In Place Of Key Letters

INPUT:

```
.TS
box;
cfB s s s.
Composition of Foods
_
.T&
c | c s s
c | c s s
c | c | c | c.
FoodⓉPercent by Weight
\^Ⓣ_
\^ⓉProteinⓉFatⓉCarbo-
\^Ⓣ\^Ⓣ\^Ⓣhydrate
_
.T&
l | n | n | n.
ApplesⓉ.4Ⓣ.5Ⓣ13.0
HalibutⓉ18.4Ⓣ5.2Ⓣ...
Lima beansⓉ7.5Ⓣ.8Ⓣ22.0
MilkⓉ3.3Ⓣ4.0Ⓣ5.0
MushroomsⓉ3.5Ⓣ.4Ⓣ6.0
Rye breadⓉ9.0Ⓣ.6Ⓣ52.7
.TE
```

OUTPUT:

Composition of Foods			
Food	Percent by Weight		
	Protein	Fat	Carbo-hydrate
Apples	.4	.5	13.0
Halibut	18.4	5.2	...
Lima beans	7.5	.8	22.0
Milk	3.3	4.0	5.0
Mushrooms	3.5	.4	6.0
Rye bread	9.0	.6	52.7

Figure 2-5. Table Using Additional Command Lines

INPUT:

```
.TS
allbox;
cfI s s
cw(1i) cw(1.75i) cw(1.75i)
l l l.
New York Area Rocks
.sp
EraⓉFormationⓉAge (years)
PrecambrianⓉReading ProngⓉ>1 billion
PaleozoicⓉManhattan ProngⓉ400 million
MesozoicⓉT{
.na
Newark Basin, incl.
Stockton,Lockatong, and Brunswick
formations
.ad
T}Ⓣ200 million
CenozoicⓉCoastal PlainⓉT{
.na
On Long Island 30,000 years;
Cretaceous sediments redeposited
by recent glaciation
.ad
T}
.TE
```

OUTPUT:

New York Area Rocks		
Era	Formation	Age (years)
Precambrian	Reading Prong	>1 billion
Paleozoic	Manhattan Prong	400 million
Mesozoic	Newark Basin, incl. Stockton, Lockatong, and Brunswick formations	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years; Cretaceous sediments redeposited by recent glaciation

Figure 2-6. Table Using Text Blocks

Chapter 3

PIC GRAPHICS LANGUAGE

Chapter 3

PIC GRAPHICS LANGUAGE

1. Introduction

Pic is a language for drawing simple pictures. It operates as yet another **troff** preprocessor, (in the same style as **eqn** and **tbl**), with pictures marked by `.PS` and `.PE`. **Pic** is a procedural language—a picture is drawn by specifying the motions that one goes through to draw it.

This document is primarily a user's manual for **pic**. Part 2 shows how to use **pic** in the most simple way. Subsequent parts describe how to change the sizes of objects when the defaults are inappropriate, and how to change their positions when the standard positioning rules are inappropriate. Part 3, *Reference Manual,* describes the **pic** language precisely.

1.1 TROFF Interface

Pic is usually run as a **troff** preprocessor using a command line as shown below:

pic [*options*] *file* | troff

Run it before **eqn** and **tbl** if they are also present.

The command line options to **pic** are:

−**T***xxx*	Output is being prepared for the device *xxx*. The default is *xxx*=**aps** for the APS-5 phototypesetter. This is the only phototypesetter currently supported by device-independent **troff**. Supported laser printers are the Imagen 10 (xxx=**i10**) and the Xerox 9700 (xxx=**x97**). Any unrecognized value is taken as the resolution in units per inch of the output device.

−D Draw all lines using the "**D**" escape sequence of the device-independent **troff** formatter. This can be used to correct problems with characters that do not align properly when used to draw lines.

−d Sets a debug mode where some useful information is output with the picture code.

If the `.PS` line looks like

.PS <*file*

then the contents of *file* are inserted in place of the `.PS` line (whether or not the file contains `.PS` or `.PE`).

Other than this file inclusion facility, **pic** copies the `.PS` and `.PE` lines from input to output intact, except that it adds two things right on the same line as the `.PS`:

```
.PS h w
```

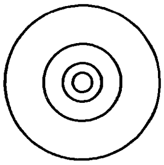
Arguments `h` and `w` are the picture height and width in units.

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** command that makes sense at that point; it is copied to the output at that point in the document. Requests for spaces or changing the line spacing is not recommended here. They may confuse the **pic** preprocessor. Point size and font changes are acceptable. So, for example,

```
.ps 24
circle radius .4i at 0,0
.ps 12
circle radius .2i at 0,0
.ps 8
circle radius .1i at 0,0
.ps 6
circle radius .05i at 0,0
.ps 10    \" don't forget to restore size
```

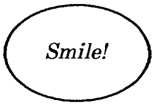
gives



Point sizes, fonts, and local motions can be modified 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, in size 8, use

```
ellipse "\s8\fISmile!\fP\s0"
```

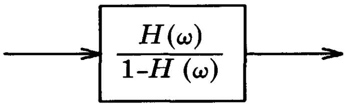
This produces



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`:

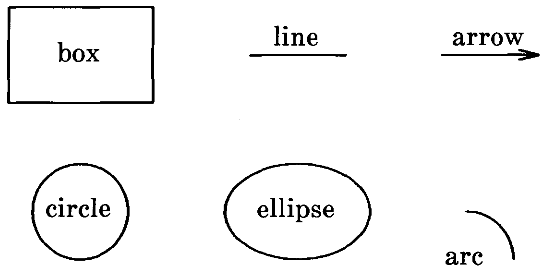
```
arrow
box "$space 0 {H( omega )} over {1 - H( omega )}$"
arrow
```



2. PIC User Manual

2.1 Basics

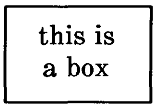
Pic provides boxes, lines, arrows, circles, ellipses, arcs, and splines (arbitrary smooth curves), plus facilities for positioning and labeling them. The picture below shows all of the fundamental objects (except for splines) in their default sizes:



Each picture begins with `.PS` and ends with `.PE`; between them are commands to describe the picture. Each command is typed on a line by itself. For example

```
.PS
box "this is" "a box"
.PE
```

creates a standard box (¾ inch wide, ½ inch high) and centers the two pieces of text in it:

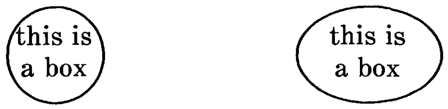


Each line of text is a separate quoted string. Quotes are mandatory, even if the text contains no blanks. (Of course there needn't be any text at all.) Each line will be printed in the current size and font, centered horizontally, and separated vertically by the current **troff** line spacing. **Pic** does not center the drawing itself.

The definitions of the `.PS` and `.PE` macros for centering pictures would be:

```
.de PS
.if t .sp .3
.in (\\n(.lu-\\$2u)/2u
.ne \\$1u
..
.de PE
.in
.if t .sp .6
..
```

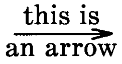
You can use `circle` or `ellipse` in place of `box`:



Text is centered on lines and arrows; if there is more than one line of text, the lines are centered above and below:

```
.PS
arrow "this is" "an arrow"
.PE
```

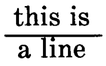
produces



and

```
line "this is" "a line"
```

gives

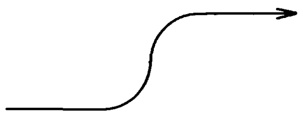


Boxes and lines may be dashed or dotted; just add the word `dashed` or `dotted` after `box` or `line`.

Arcs by default turn 90 degrees counterclockwise from the current direction; you can make them turn clockwise by saying `arc cw`. So

```
line; arc; arc cw; arrow
```

produces



A spline might well do this job better; we will return to that shortly.

As you might guess,

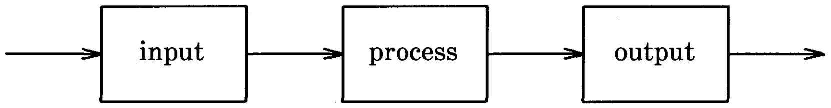
```
arc; arc; arc; arc
```

draws a circle, though not very efficiently.

Objects are normally drawn one after another, left to right, and connected at the obvious places. Thus the input

```
arrow; box "input"; arrow; box "process"; arrow; box "output"; arrow
```

produces the figure



If you want to leave a space at some place, use `move`:

```
box; move; box; move; box
```

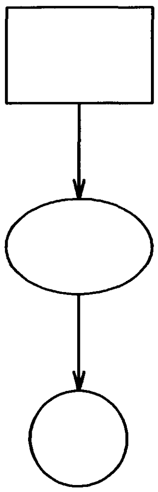
produces

Notice that several commands can be put on a single line if they are separated by semicolons.

Although objects are normally connected left to right, this can be changed. If you specify a direction (as a separate object), subsequent objects will be joined in that direction. Thus

```
down; box; arrow; ellipse; arrow; circle
```

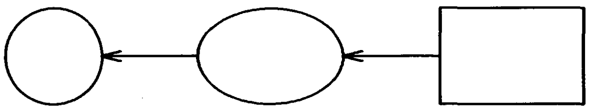
produces



and

```
left; box; arrow; ellipse; arrow; circle
```

produces



Each new picture begins going to the right.

Normally, figures are drawn at a fixed scale, with objects of a standard size. It is possible, however, to arrange that a figure be expanded to fit a particular width. If the `.PS` line contains a number, the drawing is forced to be that many inches wide, with the height scaled proportionately. Thus

```
.PS 3.5i
```

causes the picture to be 3.5 inches wide.

Pic cannot produce output when the size of text is specified in relation to the size of boxes, circles, and so on. There is as yet no way to say "make a box that just fits around this text" or "make this text fit inside this circle" or "draw a line as long as this text." Tight fitting of text can generally only be done by trial and error.

If you make a grammatical error in the way you describe a picture, **pic** will complain and try to indicate where. For example, the invalid input

```
box arrow box
```

will print the message

```
pic: syntax error near line 5, file -
 context is
        box arrow ^  box
```

The caret ^ marks the place where the error was first noted; it typically *follows* the word in error.

2.2 Controlling Sizes

This section deals with how to control the sizes of objects when the "default" sizes are not what is wanted. The next section deals with positioning them when the default positions are not right.

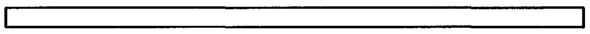
Each object that **pic** knows about (boxes, circles, etc.) has associated dimensions, like height, width, radius, and so on. By default, **pic** tries to choose sensible default values for these dimensions, so that simple pictures can be drawn with a minimum of fuss and bother. All of the figures and motions shown so far have been in their default sizes.

box	¾″ wide × ½″ high
circle	½″ diameter
ellipse	¾″ wide × ½″ high
arc	½″ radius
line or arrow	½″ long
move	½″ in the current direction

When necessary, you can make any object any size you want. For example, the input

```
box width 3i height 0.1i
```

draws a long, flat box



3 inches wide and 1/10 inch high. There must be no space between the number and the "`i`" that indicates a measurement in inches. In fact, the "`i`" is optional; all positions and dimensions are taken to be in inches.

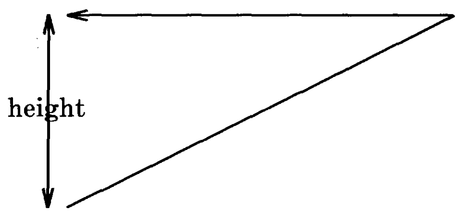
Giving an attribute like `width` changes only the one instance of the object. You can also change the default size for all objects of a particular type, as discussed later.

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.

Lines and arrows are most easily drawn by specifying the amount of motion from where one is right now, in terms of directions. Accordingly the words `up`, `down`, `left` and `right` and an optional distance can be attached to `line`, `arrow`, and `move`. For example,

```
.PS
line up 1i right 2i
arrow left 2i
move left 0.1i
line <-> down 1i "height"
.PE
```

draws

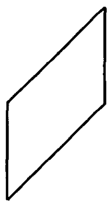


The notation <-> indicates a two-headed arrow; use -> for a head on the end and <- for one on the start. Lines and arrows are really the same thing; in fact, `arrow` is a synonym for `line ->`.

If you don't put any distance after `up`, `down`, etc., **pic** uses the standard distance. So

```
line up right; line down; line down left; line up
```

draws the parallelogram



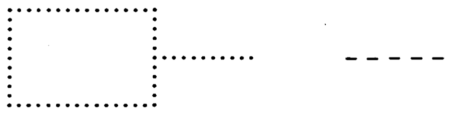
Warning: a very common error is to say

```
line 3i
```

A direction is needed.

```
line right 3i
```

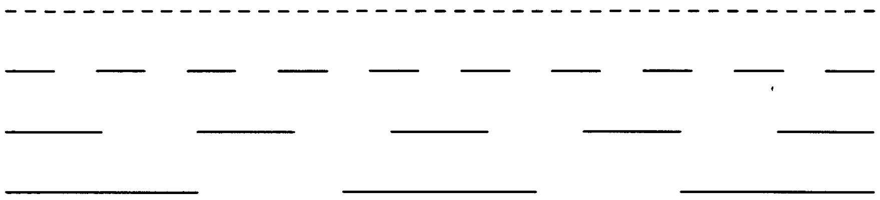
Boxes and lines may be dotted or dashed:



comes from

```
box dotted; line dotted; move; line dashed
```

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 inputs (as separate pictures)

```
line right 4.5i dashed
line right 4.5i dashed 0.25i
line right 4.5i dashed 0.5i
line right 4.5i dashed 1i
```

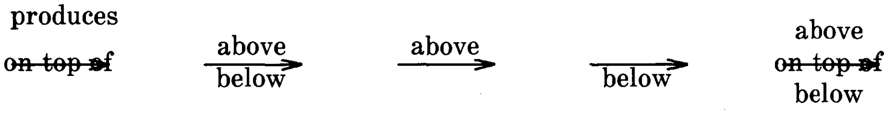
Circles and arcs cannot be dotted or dashed.

You can make any object invisible by adding the word `invis(ible)` after it. This is particularly useful for positioning things correctly near text, as we will see later.

Text may be positioned on lines and arrows:

```
.PS
arrow "on top of"; move
arrow "above" "below"; move
arrow "above" above; move
arrow "below" below; move
arrow "above" "on top of" "below"
.PE
```

produces



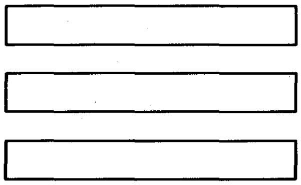
The "width" of an arrowhead is the distance across its tail; the "height" is the distance along the shaft. The arrowheads in this picture are default size.

As we said earlier, arcs go 90 degrees 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 one of <-, -> or <->.

In all cases, unless an explicit dimension for some object is specified, you will get the default size. If you want an object to have the same size as the previous one of that kind, add the word **same**. Thus in the set of boxes given by

```
down; box ht 0.2i wid 1.5i; move down 0.15i
box same; move same; box same
```



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.

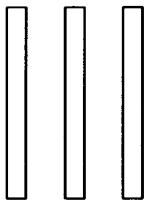
It is possible to change the default sizes of objects by assigning values to certain variables:

```
boxwid, boxht
linewid, lineht
dashwid
circlerad
arcrad
ellipsewid, ellipseht
movewid, moveht
arrowwid, arrowht     (These refer to the arrowhead.)
```

So if you want all your boxes to be long and skinny, and relatively close together,

```
boxwid = 0.1i; boxht = 1i
movewid = 0.2i
box; move; box; move; box
```

gives



Pic works internally in what it thinks are inches. Setting the variable `scale` to some value causes all dimensions to be scaled down by 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.

2.3 Controlling Positions

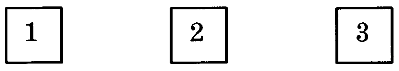
You can place things anywhere you want; **pic** provides a variety of ways to talk about places. **pic** uses a standard Cartesian coordinate system, so any point or object has an *x* and *y* position. The first object is placed with its start at position 0,0 by default. The *x,y* position of a box, 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.

You can also use `up`, `down`, `right`, and `left` with `line` and `move`. Thus

```
.PS 2
box ht 0.2 wid 0.2 at 0,0 "1"
move to 0.5,0         # or "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:



Note the use of `same` to repeat the previous dimensions instead of reverting to the default values.

Comments can be used in pictures; they begin with a `#` and end at the end of the line.

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.

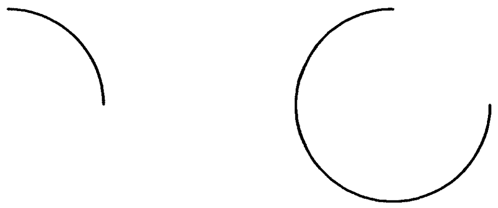
The `from` and `to` attributes are particularly useful with arcs, to specify the endpoints. By default, arcs are drawn counterclockwise,

```
arc from 0.5i,0 to 0,0.5i
```

is the short arc and

```
arc from 0,0.5i to 0.5i,0
```

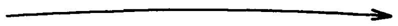
is the long one:



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 made large to provide flat arcs:

```
arc -> cw from 0,0 to 2i,0 rad 15i
```

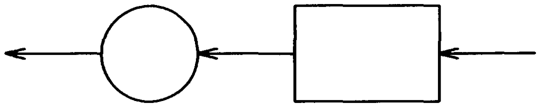
produces



We said earlier that objects are normally connected left to right. This is an over-simplification. The truth is that objects are connected together in the direction specified by the most recent `up`, `down`, `left` or `right` (either alone or as part of some object). Thus, in

```
arrow left; box; arrow; circle; arrow
```

the `left` implies connection towards the left:



This could also be written as

```
left; arrow; box; arrow; circle; arrow
```

Objects are joined in the order determined by the last `up`, `down`, etc., with the entry point of the second object attached to the exit point of the first. Entry and exit points for boxes, circles and ellipses are on opposite sides, and the start and end of lines, motions and arcs. It's not entirely clear that this automatic connection and direction selection is the right design, but it seems to simplify many examples.

If a set of commands is enclosed in braces {...}, the current position and direction of motion when the group is finished will be exactly where it was when entered. Nothing else is restored. There is also a more general way to group objects, using `[` and `]`, which is discussed in a later section.

2.4 Labels and Corners

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
```

Place names have to begin with an upper case letter (to distinguish them from variables, 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:

```
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
```

The reserved name `Here` may be used to record the current position of some object, for example as

```
Box1:  Here
```

Labels are variables — they can be reset several times in a single picture, so a line of the form

```
Box1:  Box1 + 1i,1i
```

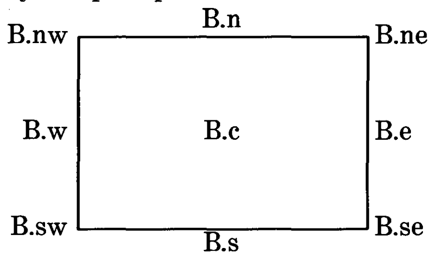
is perfectly legal.

You can also refer to previously drawn objects of each type, using the word `last`. For example, given the 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 have "corners" named by 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 alter the default positioning of text, and of a blank with some strings to help space them away from a vertical line.

Lines and arrows have a `start`, an `end` and a center in addition to corners. (Arcs have only a center, a start, and an end.) There are a host of (i.e., too 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 '`.`' notation, as in

```
last box.ne
```

you can instead say

```
upper right of last box
```

A longer statement like

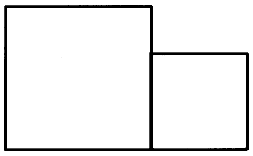
```
line from upper left of 2nd last box to bottom of 3rd last ellipse
```

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,

```
box ht 0.75i wid 0.75i
box ht 0.5i wid 0.5i with .sw at last box.se
```

produces

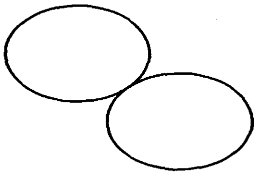


Notice that the corner after `with` is written `.sw`.

As another example, consider

```
ellipse; ellipse with .nw at last ellipse.se
```

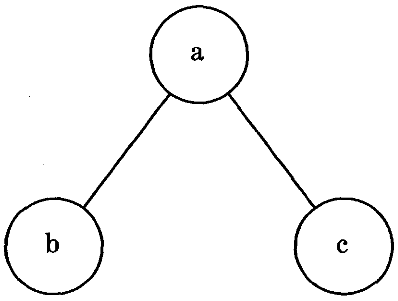
which makes



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.

```
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
```

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*

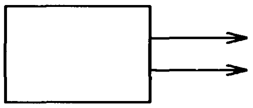
fraction is any expression, and *position1* and *position2* are any positions. You can abbreviate this phrase; "of the way" is optional, and the whole thing can be written instead as

fraction < *position1* , *position2* >

As an example,

```
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>
```

produces



Naturally, the distance given by *fraction* can be greater than 1 or less than 0.

2.5 Variables and Expressions

It's generally a bad idea to write everything in absolute coordinates if you are likely to change things. **pic** variables let you parameterize your picture:

```
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)
```

Expressions may use the standard operators `+`, `-`, `*`, `/`, and `%`, and parentheses for grouping.

Probably the most important variables are the predefined ones for controlling the default sizes of objects, listed in Section 3. 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:

```
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
```

Any pair of expressions enclosed in parentheses defines a position; furthermore such positions can be added or subtracted to yield new positions:

$(x_1,y_1)+(x_2,y_2)$

are positions, then

(p_1,p_2)

refers to the point

$(p_{1.x},p_{2.y})$

2.6 More on Text

Normally, text is centered at the geometric center of the object it is associated with. 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.

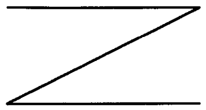
At the moment you can *not* compound text attributes. It is illegal to say `"..." above ljust.`

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
```

2.7 Lines and Splines

A "line" may actually be a path, that is, it may consist of connected segments like this:



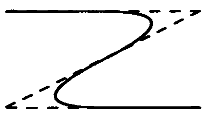
This line was produced by

```
line right 1i then down .5i left 1i then right 1i
```

A spline is a smooth curve guided by a set of straight lines just like the line above. It begins at the same place, ends at the same place, and in between is tangent to the mid-point of each guiding line. The syntax for a spline is identical to a (path) line except for using `spline` instead of `line`. Thus:

```
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
```

produces



(Long input lines can be split by ending each piece with a backslash.)

The elements of a path, whether for line or spline, are specified as a series of points, either in absolute terms or by `up`, `down`, etc. If necessary to disambiguate, the word `then` can be used to separate components, as in

```
spline right then up then left then up
```

which is not the same as

```
spline right up left up
```

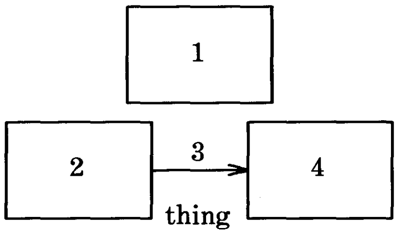
At the moment, arrowheads may only be put on the ends of a line or spline; splines may not be dotted or dashed.

2.8 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

```
box "1"
[ box "2"; arrow "3" above; box "4" ] with .n at last box.s - (0,0.1)
"thing" at last [].s
```

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: ...;  ]
```

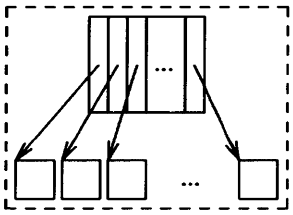
When combined with **`define`** statements (next section), blocks provide a reasonable simulation of a procedure mechanism.

Although blocks nest, it is currently 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.

```
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 []
```

This produces



2.9 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** is any character that does not appear in the replacement. 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

```
define square X box ht $1 wid $1  $2 X
```

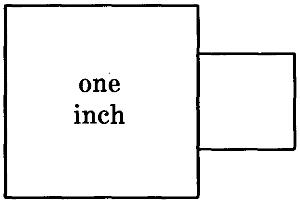
Then

```
square(1i, "one" "inch")
```

calls for a one-inch square with the obvious label, and

```
square(0.5i)
```

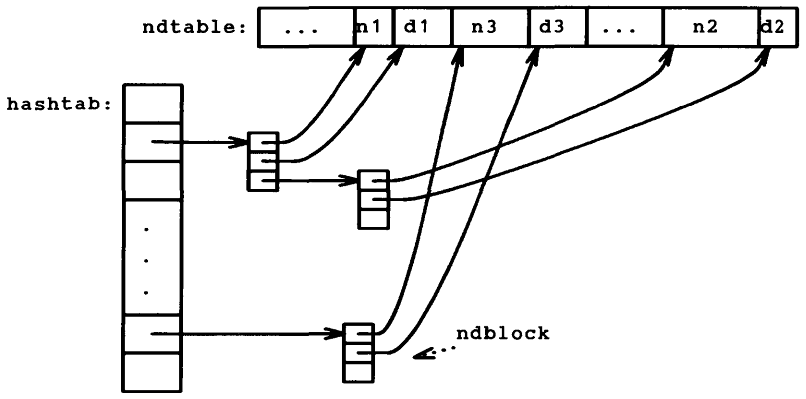
calls for a square with no label:



Coordinates like *x,y* may be enclosed in parentheses, as in (*x,y*), so they can be included in a macro argument.

2.10 Some Examples

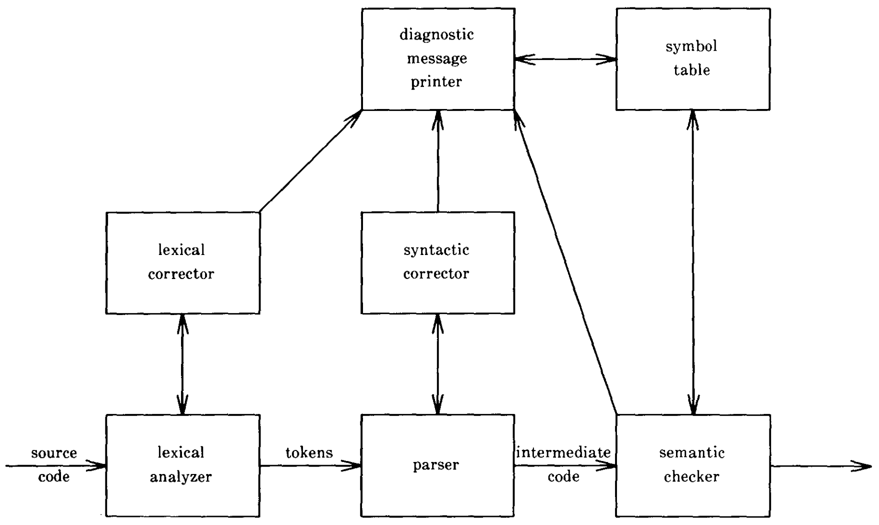
Here are a few larger examples:



The input for the picture above was:

```
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
```

The second example follows.



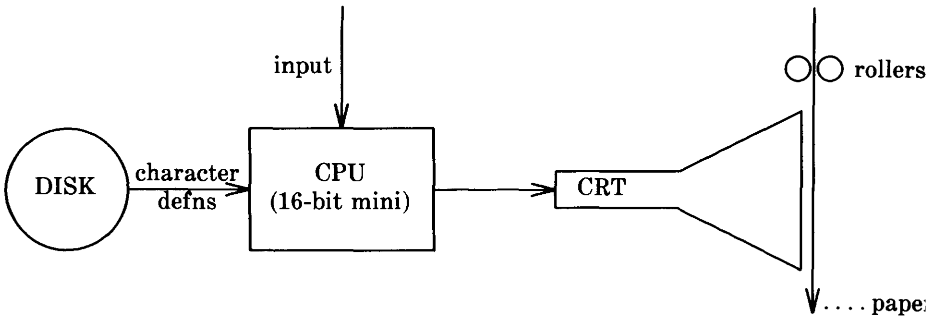
This input will generate a picture something like the above:

```
.PS 6
.ps 8
     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
.PE
```

There are eighteen objects (boxes and arrows) in the second example, and one line of **pic** input for each; this seems like an acceptable level of verbosity.

The next example is the following:



Basic Digital Typesetter

This input will generate a picture like that in example 3:

```
.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
.ce
Basic Digital Typesetter
```

3. PIC Reference Manual

3.1 Pictures

The top-level object in **pic** is the "picture":

picture:
        .PS *optional-width*
        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.

If instead the line is

```
.PS <f
```

the file `f` is inserted in place of the `.PS` line.

If `.PF` is used instead of `.PE`, the position after printing is restored to what it was upon entry.

3.2 Elements

An *element-list* is a list of elements. The elements are

element:
        primitive attribute-list
        placename : element
        placename : position
        variable = expression
        direction
        troff-command
        { *element-list* }
        [*element-list*]

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 are local to that block.

The *troff-command* is any line that begins with a period. Such lines are assumed to make sense in the context where they appear.

3.3 Primitives

The primitive objects are

primitive:
```
    box
    circle
    ellipse
    arc
    line
    arrow
    move
    spline
    "any text at all"
```

`arrow` is a synonym for `line` ->.

3.4 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*	`->` `<-` `<->`
`same`	`invis`
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, dotted, dashed, invis, same, text
circle and ellipse:
     radius, diameter, height, width, at, invis, same, text
arc:
     up, down, left, right, height, width, from, to, at, radius,
     invis, same, cw, <-, ->, <->, text
line,arrow
     up, down, left, right, height, width, from, to, by, then,
     dotted, dashed, invis, same, <-, ->, <->, text
spline:
     up, down, left, right, height, width, from, to, by, then,
     invis, <-, ->, <->, text
move:
     up, down, left, right, to, by, same, text
"text...":
     at, text
```

The attribute `at` implies placing the geometrical center at the specified place. For lines, splines and arcs, `height` and `width` refer to arrowhead size.

3.5 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.

3.6 Positions and Places

A position is ultimately an *x,y* coordinate pair, but it may be specified in other ways.

position:

e, e
`place` ± *e, e*
`(` *position, position* `)`
e `[of the way]` `between` *position* `and` *position*
e < *position* , *position* >

The pair *e, e* may be enclosed in parentheses.

place:

`placename` *optional-corner*
corner `placename`
`Here`
corner `of` *nth primitive*
nth primitive optional-corner

A *corner* is one of the eight compass points or the center or the start or end of a primitive. (Not text!)

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:

~~nth~~
`nth last`

Legal input includes `1th`, as well as synonyms like `1st` and `3st`.

3.7 Variables

The built-in variables and their default values are:

`boxwid 0.75i`	`boxht 0.5i`
`circlerad 0.25i`	
`ellipsewid 0.75i`	`ellipseht 0.5i`
`arcrad 0.25i`	
`linewid 0.5i`	`lineht 0.5i`
`movewid 0.5i`	`movewid 0.5i`
`arrowht 0.1i`	`arrowwid 0.05i`
`dashwid 0.1i`	
`scale 1`	

These may be changed at any time, and the new values remain in force until changed again. Dimensions are divided by `scale` during output.

3.8 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`

3.9 Definitions

The `define` statement is not part of the grammar.

```
define:
     define name x  replacement text x
```

Occurrences of `$1` through `$9` in the 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.

Chapter 4

MATHEMATICS TYPESETTING PROGRAM

Chapter 4

MATHEMATICS TYPESETTING PROGRAM

1. Introduction

Mathematical text is known in the publishing trade as "penalty copy" because it is slower, more difficult, and more expensive to set in type than any other kind of copy normally occurring in books and journals.

- One difficulty is the multiplicity of characters, sizes, and fonts. Many mathematical expressions require an intimate mixture of Roman, italic, and Greek letters (in three sizes) and a number of special characters. Typesetting such expressions by traditional methods is essentially a manual operation.

- A second difficulty is the 2-dimensional character of mathematics. This is illustrated by the following example which shows line-drawing, built-up characters (such as braces and radicals), and a spectrum of positioning problems:

$$\int \frac{dx}{ae^{mx} - be^{-mx}} = \begin{cases} \dfrac{1}{2m\sqrt{ab}} \log \dfrac{\sqrt{a}\,e^{mx} - \sqrt{b}}{\sqrt{a}\,e^{mx} + \sqrt{b}} \\ \dfrac{1}{m\sqrt{ab}} \tanh^{-1}(\dfrac{\sqrt{a}}{\sqrt{b}} e^{mx}) \\ \dfrac{-1}{m\sqrt{ab}} \coth^{-1}(\dfrac{\sqrt{a}}{\sqrt{b}} e^{mx}) \end{cases}$$

The **eqn** software for typesetting mathematics has been designed to be easy to learn and to use by people (for example, secretaries and mathematical typists) who know neither mathematics nor typesetting. The language can be learned in an hour or so since it has few rules and fewer exceptions. It interfaces directly with the phototypesetting language so mathematical expressions can be embedded in the running text of a manuscript, and the entire document produced in one process. Typical mathematical expressions include size and font changes, positioning, line drawing, and other necessary functions to print according to mathematical conventions, and are done automatically. The syntax of the language is specified by a small context-free grammar; a compiler-compiler is used to make a compiler that translates this language into typesetting commands. Output may be produced on either a typesetter or on a terminal with forward and reverse half-line motions.

2. Usage

On the UNIX system, the typesetter is driven by a text formatting program, **troff**, which was designed for typesetting text. Facilities needed for printing mathematical expressions, such as arbitrary horizontal and vertical motions, line drawing, and font size changing are also provided. Syntax for describing these special operations is difficult to learn and difficult even for experienced users to type correctly. For this reason, the **troff** formatter is used as an assembly language by the **eqn** program which describes and compiles mathematical expressions.

To typeset mathematical text stored in *files*, the following command is issued:

```
eqn files | troff
```

The vertical bar connects the output of one **eqn** process to the input of another **troff** process. Any **troff** formatter options are located following the **troff** formatter part of the command. For example:

```
eqn files | troff -mm
```

Eqn can also be used on devices which have half-line forward and reverse capabilities. Input language is identical, but **neqn** and the **nroff** formatter are used instead of **eqn** and the **troff** formatter. Some things will not look as good because terminals do not provide the variety of characters, sizes, and fonts that a typesetter does, but the output is usually adequate for proofreading.

To use a specific terminal as the output device, the following command is used:

```
neqn files | nroff -Tx
```

where *x* is the terminal type being used, such as 300 or 300S.

The **eqn** and **neqn** programs can be used with the **tbl** program for typesetting tables that contain mathematics

```
tbl files | eqn | troff
tbl files | neqn | nroff
```

Missing delimiters and some equation errors can be detected early with program aids. Using these troubleshooting devices described in paragraph 5 should be considered as an initial step in formatting a document.

3. Language

3.1 Design

The fundamental principle upon which the **eqn** language design is based is that the language should be easy to use by those who know neither mathematics nor typesetting. This principle implies:

- Normal mathematical conventions about operator precedence, such as parentheses, cannot be used. To give special meaning to such characters means that the user has to understand what is being typed. The language should not assume that parentheses are always balanced.

- There should be few rules, keywords, special symbols, and operators. This keeps the language easy to learn and remember. Furthermore, there should be few exceptions to the rules that do exist. If something works in one situation, it should work everywhere. If a variable can have a subscript, then a subscript can have a subscript, etc., without limit.

- Standard things should happen automatically. When "x=y+z+1" is typed, "x=y+z+1" should be the result. Subscripts and superscripts should be printed automatically (with no special intervention) in appropriately smaller size. Fraction bars should be made the right length and positioned at the correct height. A mechanism for overriding default actions should exist, but its application is the exception, not the rule.

A secondary, but still important, design goal is that the system should be easy to build and to change. To this end and to guarantee regularity, the language is defined by a context-free grammar.

The typist should have a reasonable picture (a 2-dimensional representation) of the desired final form, such as might be handwritten by the author of a paper. It is also assumed that the input is to be typed on a computer terminal much like an ordinary typewriter. This implies an input alphabet of perhaps 100 characters, none of them special.

The **troff** processor performs work for the mathematics typesetting function. It is a powerful program, with a macro facility, text and arithmetic variables, numerical computation and testing, and conditional branching. Text strings are passed to the **troff** formatter omitting the need for a separate storage management package. The user need not be concerned with most details of the particular device and character set currently in use. For example, the **troff** formatter computes the widths of all strings of characters; the user does not need to know about them.

3.2 Structure

The basic structure of the language is not original. Equations are pictured as a set of boxes, pieced together in various ways. For example, something with a subscript is a box followed by another box moved downward and shrunk an appropriate amount. A fraction is a box centered above another box, at the right altitude, with a line of correct length drawn between them.

3.3 Mode of Operation

Since the **eqn** program is useful for typesetting mathematics only, it interfaces with the underlying typesetting language in order to get intermingled mathematics and text. The standard mode of operation is that when a document is typed, mathematical expressions are input as part of the text but marked by delimiters, **.EQ** and **.EN**. The program reads this input and treats as comments those things which are not mathematics passing them through untouched. At the same time, it converts mathematical inputs into **troff** formatter commands. The resulting output is passed directly to the formatter where comments and mathematical parts become text and/or formatter commands.

4. User's Guide

4.1 Delimiters

The **eqn** preprocessor reads intermixed text and equations and passes its output to the **troff** formatter. Since the formatter uses lines beginning with a period as control words (**.ce** means "center the next output line"), **eqn** uses the **.EQ** macro to mark the beginning of an equation and the **.EN** macro to mark the end. By default **.EQ** and **.EN** are ignored by the **troff** formatter, so equations are printed in-line.

The **.EQ** and **.EN** macros can be supplemented by **troff** commands as desired. A centered display equation can be produced with the input

```
.ce
.EQ
x sub i = y sub i ...
.EN
```

The **.EQ** and **.EN** delimiters are passed through to the formatter untouched, so they can be used to center equations, number them automatically, etc. The **troff** and **nroff** formatter macro package, −**mm**, allows equations to be left-justified and numbered. Any argument to the **.EQ** macro will be placed at the right margin as an equation number.

Warning: When using the* -mm *macro package, always use a break-producing request such as* .br *or* .sp *immediately before the* .EQ *macro.

For example, the input

```
.EQ (4.1a)
x = f(y/2) + y/2
.EN
```

produces the output

$$x=f(y/2)+y/2 \qquad (4.1a)$$

Since it is tedious to type **.EQ** and **.EN** around very short expressions (e.g., single letters), two characters can be defined to serve as the left and right delimiters of expressions. These characters are recognized anywhere in subsequent text {4.16}.

4.2 Spaces and New Lines

4.2.1 Input Spaces

Input is free form. Space and newline characters in the input are used by **eqn** to separate pieces of the input; they are not used to create space in the output.

Thus an input

```
x   =   y
  + z + 1
```

produces

$$x=y+z+1$$

Free-form input is easier to type initially. Space and newline characters should be freely used to make input equations readable and easy to edit. Very long lines are hard to correct if a mistake is made.

4.2.2 Output Spaces

Extra white space can be forced into the output by several characters of various sizes. A tilde (~) gives a space equal to the normal word spacing in text, a circumflex (^) gives half this much, and a tab character spaces to the next tab stop (tab stops must be set by **troff** commands). Spaces, tildes, circumflexes, and tabs also serve to delimit pieces of input. For example:

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

produces

$$x \ = \ y \ + \ z$$

4.3 Symbols, Special Names, and Greek Alphabet

Mathematical symbols, mathematical names, and the Greek alphabet are known by **eqn**. For example:

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

produces

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

Spaces in the input are necessary to indicate that *sin*, *pi*, *int*, and *omega* are separate entities and should get special treatment. The **eqn** program looks up each string of characters in a table, and if found, gives it a translation. Digits, parentheses, brackets, punctuation marks, and the following mathematical words are converted to Roman font:

```
sin cos tan sinh cosh tanh arc
max min lim log ln exp
Re Im and if for det
```

In the previous example, *pi* and *omega* become their Greek equivalents (π and ω), *int* becomes the integral sign (which is moved down and enlarged), and *sin* is output in Roman font, following conventional mathematical practice. Parentheses, digits, and operators are output in Roman font.

Spaces should be put around separate parts of the input. A common error is to type "f(pi)" without leaving spaces on both sides of the "pi". As a result, **eqn** does not recognize *pi* as a special word, and it in the output. A list of **eqn** names appears in Figure 4-1.

INPUT NAME	*OUTPUT CHARACTER*
>=	≥
<=	≤
==	≡
!=	≠
+-	±
->	→
<-	←
<<	≪
>>	≫
inf	∞
partial	∂
half	½
prime	′
approx	≈
nothing	
cdot	·
times	×
del	∇
grad	∇
...	· · ·
,...,	, · · · ,
sum	Σ
int	∫
prod	Π
union	⋃
inter	⋂
DELTA	Δ
GAMMA	Γ
LAMBDA	Λ
OMEGA	Ω

Figure 4-1. Names Recognized by eqn (Sheet 1 of 2)

INPUT NAME	*OUTPUT CHARACTER*
PHI	Φ
PI	Π
PSI	Ψ
SIGMA	Σ
THETA	Θ
UPSILON	Υ
XI	Ξ
alpha	α
beta	β
chi	χ
delta	δ
epsilon	ε
eta	η
gamma	γ
iota	ι
kappa	κ
lambda	λ
mu	μ
nu	ν
omega	ω
omicron	o
phi	ϕ
pi	π
psi	ψ
rho	ρ
sigma	σ
tau	τ
theta	θ
upsilon	υ
xi	ξ
zeta	ζ

Figure 4-2. Names Recognized by eqn (Sheet 2 of 2)

Four-character **troff** names can also be used for anything **eqn** does not recognize, e.g., “\(pl” for the + sign.

The only way **eqn** can deduce that some sequence of letters may be special is if that sequence is separated from the letters on either side of it. This can be done by surrounding a special word by ordinary space, tab, or newline characters. Special words can also be made to stand out by surrounding them with tildes or circumflexes, e.g.:

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

is much the same as the previous example, except tildes separate words like *sin*, *omega*, etc., and also add an extra space per tilde. The output of this example is:

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

4.4 Subscripts and Superscripts

Subscripts and superscripts are introduced by the keywords “sub” and “sup”:

$$x^2 + y_k$$

is produced by

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

The **eqn** program takes care of all size changes and vertical motions needed to make the hard copy look right. The words “sub” and “sup” must be surrounded by spaces. A space or tilde is used to mark the end of a subscript or superscript. Return to the original base line is automatic.

Multiple levels of subscripts or superscripts are allowed. Subscripted subscripts and superscripted superscripts such as:

```
x sub i sub 1
```

produce

x_{i_1}

A subscript and superscript on the same thing are printed one above the other if the subscript comes first.

```
x sub i sup 2
```

is

x_i^2

Other than this special case, "sub" and "sup" group to the right

```
x sup y sub z
```

generates

x^{y_z}

not

${x^y}_z$

A common erroneous expression is of the form

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

which causes

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

instead of the intended

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

The error is in omitting a delimiting space. The correct input expression is

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

4.5 Braces

Complicated expressions can be formed by using braces ({ }) to keep objects together in unambiguous groups. Braces indicate what goes over what or what terms are to be grouped before applying another mathematical function.

Normally, the end of a subscript or superscript is marked by a space, tilde, circumflex, or tab. If the subscript or superscript is something that has to be typed with spaces in it, braces are used to mark the beginning and end. The input

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

produces

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

Braces can be used to force **eqn** to treat something as a unit or just to make the intent perfectly clear.

Braces can occur within braces if necessary. The statement

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

generates

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

A general rule is that an arbitrarily complicated string enclosed in braces can be used in place of a single character (such as *x*). The **eqn** program administers formatting details. In all cases, the correct number of braces must be used. Omitting one or adding an extra one causes **eqn** to complain.

The braces convention is an example of the power of using a recursive grammar to define the language. It is part of the language dictates that if a construct can appear in some context then any expression within braces can also occur in that context.

4.6 Fractions

Fractions are specified with the keyword *over*.

```
a+b over c+d+e = 1
```

produces

$$\frac{a+b}{c+d+e}=1$$

The line is made the correct length and positioned automatically. When there is both an “over” and a “sup” in the same expression, **eqn** performs the “sup” first.

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

is

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

4.7 Square Roots

There is a *sqrt* operator for making square roots of the appropriate size.

```
x =  {-b  +-  sqrt{b sup 2 -4ac}}  over 2a
```

yields

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Note: Since large radicals look poor on some typesetters, *sqrt* is not recommended for tall expressions.

4.8 Summations, Integrals, and Similar Constructions

Summations, integrals, and similar constructions are easy.

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

produces

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

Braces indicate where the upper part (**i= inf**) begins and ends. None are necessary for the lower part (**i=0**) because it contains no spaces. Braces will never hurt; but if the "from" and "to" parts contain any spaces, braces must be put around them.

The "from" and "to" parts of the construction are optional; but if both are used, they have to occur in that order.

Other useful characters can replace the *sum* in the above example. They are

```
int
prod
union
inter
```

which become, respectively

$\int$

Π

$\cup$

$\cap$

Since characters before the "from" can be anything, even something in braces, "from-to" can often be used in unexpected ways.

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

is

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

4.9 Size and Font Changes

Although **eqn** makes an attempt to use correct sizes and fonts, there are times when default assumptions are not what is wanted. Slides and transparencies often require larger characters than normal text. Thus size and font changing commands are also provided. By default, equations are set in 10-point type with standard mathematical conventions to determine what characters are in Roman and italic font. Size and font changes are made with *size n* and *roman, italic, bold,* or *fat* operations. As with the "sub" and "sup" keywords, size and font changes affect only the string that follows and revert to the normal situation afterward. Thus:

```
bold x y
```

is

$\mathbf{x}y$

Braces can be used if something more complicated than a single letter is to be affected.

bold {x y} z

produces

$\mathbf{xy}z$

If fonts other than Roman, italic, and bold are to be used, the *font X* statement (*X* is a 1-character **troff** name or number for the font) can be used. Since **eqn** is tuned for Roman, italic, and bold fonts, other fonts may not give as good an appearance.

The *fat* operation takes the current font and widens it by overstriking. For instance:

A = fat {pi r sup 2}

produces

$A = \boldsymbol{\pi r^2}$

Legal sizes which may follow *size* are

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

The size can also be changed by a given amount. For example:

size +2

makes the size two points larger. This has the advantage that knowledge of the current size is not necessary.

If an entire document is to be in a nonstandard size or font, it is a nuisance to write out a size and font change for each equation. Accordingly, a global size or font can be set that thereafter affects all equations. The following statements would appear at the beginning of any equation to set the size to 16 and the font to Roman:

```
.EQ
gsize 16
gfont R
. . .
.EN
```

In place of **R**, any of the **troff** font names may be used. The size after *gsize* can also be a relative change with + or −.

Generally, *gsize* and *gfont* appear at the beginning of a document. 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 size of equations should match the size of the footnote text. Footnote text is usually two points smaller than the main text. Global size should be reset at the end of the footnote.

4.10 Diacritical Marks

Diacritical marks, a problem in traditional typesetting, are straightforward in **eqn**. There are several words used to get marks

INPUT	*OUTPUT*
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 *bar* and *under* are made the right length for the entire construct. Other marks are centered. An example of an expression using diacritical marks is:

$$\underset{\cdot}{x} + \hat{x} + \tilde{y} + \hat{X} + \ddot{Y} = \overline{z + Z}$$

It is made by typing

```
x dot under + x hat + y tilde
+ X hat + Y dotdot = z+Z bar
```

4.11 Quoted Text

An input entirely within quotes (" . . . ") is not subject to font changes or spacing adjustments normally done by the typesetting program. This provides for individual spacing and adjusting if needed. For example:

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

produces

sin(x)+sin(*x*)

Quotes are also used to get braces and other **eqn** keywords printed.

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

prints

{ *size alpha* }

and

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

prints

```
    { size alpha }
```

The " " construction is often used as a place-holder when grammatically **eqn** needs something, but nothing is actually wanted on the output.

4.12 Aligning Equations

Sometimes it is necessary to align a series of equations at a horizontal position, often at an equals sign. This is done with two operations called *mark* and *lineup.*

The word *mark* may appear 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 at all possible. For example:

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

produces

```
    x+y=z
      x=1
```

The *mark* and *lineup* operations do not work with centered equations. Also, *mark* does not look ahead.

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

is not going to work because there is not room for the $x+y$ part after the *mark* remembers where the x is.

4.13 Big Brackets

To get large brackets [ ], braces {}, parentheses (), and bars ¦¦ around information that exists on more than one line, the *left* and *right* keywords are used.

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

produces

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

The resulting brackets are made large enough to cover whatever they enclose. Other characters can be used besides these, but they are not likely to look very good. One exception is the *floor* and *ceiling* characters.

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

produces

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

Braces are larger than brackets and parentheses because they are made up of three, five, seven, etc., pieces while brackets can be made up of two, three, four, etc., pieces. Large left and right parentheses often look strange because of the design of the character set.

The *right* keyword may be omitted. A "left something" need not have a corresponding "right something". If the right part is omitted, braces are put around the thing that the left bracket is to encompass. Otherwise, resulting brackets may be too large. If the left part is to be omitted, things are more complicated because technically a *right* cannot exist without a corresponding *left*. Instead the following input will do:

```
left "" ... right)
```

The **left** "" means a " left nothing" which satisfies the rules without hurting the output.

4.14 Piles

Large braces, brackets, parenthesis, and vertical bars are often used with another facility (*piles*) which makes vertical piles of objects. Elements of the pile (there can be any number) are centered one above another, at the right height for most purposes. The keyword *above* is used to separate the pieces; braces are used around the entire list. Elements of a pile can be as complicated as needed, even containing more piles.

Three other forms of pile exist:

- *lpile* makes a pile with the elements left-justified
- *rpile* makes a right-justified pile
- *cpile* makes a centered pile, just like *pile.*

Vertical spacing between pieces is somewhat larger for *lpile, rpile,* and *cpile* than it is for ordinary piles. For example, to get

$$sign(x) \equiv \left\{ \begin{array}{rll} 1 & \text{if} & x>0 \\ 0 & \text{if} & x=0 \\ -1 & \text{if} & x<0 \end{array} \right.$$

the following is input.

```
sign (x) == left {
   rpile {1 above 0 above -1}
   ~~lpile {if above if above if}
   ~~lpile {x>0 above x=0 above x<0}
```

The **left {** construction makes a left brace large enough to enclose the **rpile {...}**, which is a right-justified pile of "above ... above ...". The **lpile** construction makes a left-justified pile.

4.15 Matrices

It is possible to make matrices. For example, to make a neat array like

$$\begin{matrix} x_i & x^2 \\ y_i & y^2 \end{matrix}$$

the following text is the input:

```
matrix {
  ccol { x sub i above y sub i }
  ccol { x sup 2 above y sup 2 }
}
```

This produces a matrix with two centered columns. Elements of the columns are then listed just as for a pile. Each element is separated by the word "above". The *lcol* or *rcol* keyword can also be used to left- or right-justify columns. Each column can be separately adjusted, and there can be as many columns as desired.

The reason for using a matrix instead of two adjacent piles is if the elements of the piles are not all the same height they will not line up properly. A matrix forces them to line up because it looks at the entire structure before deciding the spacing to use.

Note: Each column must have the same number of elements.

4.16 In-Line Equations

In a mathematical document, it is necessary to follow mathematical conventions in display equations and in text. Making variable names (such as *x*) italic is one instance. Although this could be done by surrounding the appropriate parts with **.EQ** and **.EN**, the continual repetition of **.EQ** and **.EN** is a nuisance. Furthermore, with −**mm**, **.EQ** and **.EN** imply a displayed equation.

The **eqn** program provides a shorthand notation for short in-line equations. Two characters can be defined to mark the left and right ends of an in-line equation, and then expressions can be typed in the middle of text lines.

```
.EQ
delim $$
.EN
```

The three lines added to the beginning of the document set both the left and right delimiter characters to dollar signs. A sample input is:

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

to produce:

> Let α_i be the primary variable, and let β be zero. Then it can be shown that x_1 is ≥ 0.

This works as expected—space characters, newline characters, etc., are significant in the input text, but not in the resultant equation. Multiple equations can occur in a single input line. Space is left before and after a line that contains in-line expressions so that a tall expression will not interfere with surrounding lines. To turn off the delimiters:

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

Note: The following should be observed when using the in-line equations format:

- Do not use braces, tildes, circumflexes, or double quotes as delimiters.
- In-line font changes must be closed before in-line equations are encountered.

4.17 Defines

There is a definition facility, so a user can say

```
define name '...'
```

at any time in the document. Henceforth, any occurrence of *name* in an expression will be expanded into whatever was inside the quotes in its definition. This lets users tailor the language to their own specifications. For example, if the sequence

```
x sub i sub 1 + y sub i sub 1
```

appears repeatedly throughout a paper; typing time can be saved each time the sequence is use by defining it:

```
define  xy  'x sub i sub 1 + y sub i sub 1'
```

This define makes *xy* a shorthand for whatever characters occur between the single quotes in the definition. Any character can be used instead of the quote to mark the ends of the definition as long as it does not appear inside the definition.

The above expression can now be input as follows:

```
.EQ
f(x) = xy ...
.EN
```

Each occurrence of *xy* will expand into its definition. Spaces (or their equivalent) are to be left around the name when used. The **eqn** program will identify it as special.

Although definitions can use previous definitions, as in:

```
    .EQ
    define  xi   ' x sub i '
    define  xi1  ' xi sub 1 '
    .EN
```

it is erroneous to define something in terms of itself. For instance:

```
    define  X  ' roman X '
```

Since **X** is now defined in terms of itself, problems will result. However, if the following expression is used, the quotes protect the second **X**, and everything works fine.

```
    define  X  ' roman " X"  '
```

The **eqn** keywords can be redefined. Making / mean *over* can be done with the following statement:

```
    define  /  ' over '
```

To redefine *over* as / use:

```
    define  over  ' / '
```

If different things are needed to be printed on a terminal and on the typesetter, symbols may be defined differently in **neqn** and **eqn**. This can be done with *ndefine* and *tdefine.* A definition made with *ndefine* takes effect when running **neqn**. When *tdefine* is used, the definition applies only for the **eqn** processor. Names defined with the *define* facility apply to both **eqn** and **neqn**.

4.18 Local Motions

Although the **eqn** formatter tries to position things correctly on the paper, it occasionally needs tuning to make the output just right. Small extra horizontal spaces can be obtained with tilde and circumflex. By using *back n* and *fwd n,* small amounts are moved horizontally, where *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, things can be moved up or down with an *up n* and a *down n.* As with *sub* or *sup*, local motions affect the next thing in the input, and this can be something arbitrarily complicated if it is enclosed in braces.

4.19 Precedence

Precedence rules resolve the ambiguity in a construction like

```
a sup 2 over b
```

The "sup" is defined to have a higher precedence than "over". A user can force a particular analysis by placing braces around expressions. If braces are not used to group functions, the **eqn** formatter will do operations in the following order:

```
dyad vec under bar tilde hat dot dotdot
fwd back down up
fat roman italic bold size
sub sup sqrt over
from to
```

The following operations group to the left:

```
over sqrt left right
```

All others group to the right.

5. Troubleshooting

If a mistake is made in an equation, such as omitting a brace, having one too many braces, or having a "sup" with nothing before it, the **eqn** formatter produces the following message:

```
syntax error between lines x and y, file z
```

where *x* and *y* are approximately the lines between which the trouble occurred, and *z* is the name of the file in question. There are also self-explanatory messages that arise when a quote is omitted or **eqn** is run on a nonexistent file. To check a document before printing

```
eqn files >/dev/null
```

discards the output but prints the message.

It is easy to leave out a dollar sign when used as delimiters. The **checkeq** program checks for misplaced or missing dollar signs (in-line delimiters) and similar troubles.

In-line equations can be only so big because of an internal buffer in the **troff** formatter. If a "word overflow" message is received, the limit has been exceeded. Printing the equation as a displayed equation usually causes the message to go away. The "line overflow" message indicates that an even bigger buffer has been exceeded. In this case, the equation must be broken into two separate ones, marking each with **.EQ/.EN** delimiters. The **eqn** program does not warn about equations that are too long for one line.

Your comments and suggestions are appreciated and will help us to provide the best documentation for your use.

1. How would you rate this document for COMPLETENESS? (Please Circle)

   Excellent Adequate Poor
   4 ----------------------3 ----------------------2 ----------------------1 ----------------------0

2. Identify any information that you feel should be included or removed.

   ____

   ____

3. How would you rate this document for ACCURACY of information? (Please Circle)

   Excellent Adequate Poor
   4 ----------------------3 ----------------------2 ----------------------1 ----------------------0

4. Specify page and nature of any error(s) found in this document.

   ____

   ____

5. How would you rate this document for ORGANIZATION of information? (Please Circle)

   Excellent Adequate Poor
   4 ----------------------3 ----------------------2 ----------------------1 ----------------------0

6. Describe any format or packaging problems you have experienced with this document.

   ____

   ____

7. Do you have any general comments or suggestions regarding this document?

   ____

   ____

8. We would like to know a little about your background as a user of this document:

A. Your job function ______________________

B. Number of years experience with computer hardware: operation _______ , maintenance _______ .

C. Number of years experience with computer software: user _______ , programmer _______ .

Your Name ______________________ Phone No. ______________
Company ____
Address ____
City & State ______________________ Zip Code _______

Western Electric

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 1999 GREENSBORO, N.C.

POSTAGE WILL BE PAID BY ADDRESSEE

DOCUMENTATION SERVICES
2400 Reynolda Road
Winston-Salem, N.C. 27106-9989

Do Not Tear—Fold Here and Tape