NAME

grap – Kernighan and Bentley's language for typesetting graphs

SYNOPSIS

grap [**-d** *defines_file*] [**-D**] [**-M** *include path*] [**-v**] [**-u**] [**-C**] [**-h**] [*filename ...*]

DESCRIPTION

grap is an implementation of Kernighan and Bentley's language for typesetting graphs, as described in "Grap-A Language for Typesetting Graphs, Tutorial and User Manual," by Jon L. Bentley and Brian W. Kernighan, revised May 1991, which is the primary source for information on how to use **grap**. As of this writing, it is available electronically at http://www.kohala.com/start/troff/cstr114.ps

This version is a black box implementation of **grap**, and some inconsistencies are to be expected. The remainder of this manual page will briefly outline the **grap** language as implemented here.

grap is a pic(1) pre-processor. It takes commands embedded in a troff(1) source file which are surrounded by **.G1** and **.G2** macros, and rewrites them into pic commands to display the graph. Other lines are copied. Output is always to the standard output, which is usually redirected. Input is from the given *filename*s, which are read in order. A *filename* of **-** is the standard input. If no *filename*s are given, input is read from the standard input.

Because **grap** is a pic preprocessor, and gnu pic will output TeX, it is possible to use **grap** with TeX.

The **-d** option specifies a file of macro definitions to be read at startup, and defaults to /usr/local/share/grap/grap.defines. The **-D** option inhibits the reading of any initial macros file. The defines file can also be given using the GRAP_DEFINES environment variable. (See below).

-v prints the version information on the standard output and exits. **--version** is a synonym for **-v**.

-u makes labels unaligned by default. This version of **grap** uses new features of gnu pic to align the left and right labels with the axes, that is that the left and right labels run at right angles to the text of the paper. This may be useful in porting old **grap** programs.

-M is followed by a colon-separated list of directories used to search for relative pathnames included via **copy**. The path is also used to locate the defines file, so if the **-d** changes the defines file name to a relative name, it will be searched for in the path given by **-M**. The search path always includes the current directory, and by default that directory is searched last.

All **grap** commands are included between **.G1** and **.G2** macros, which are consumed by **grap**. The output contains pic between **.PS** and **.PE** macros. Any arguments to the **.G1** macro in the input are arguments to the **.PS** macro in the output, so graphs can be scaled just like pic diagrams. If **-C** is given, any macro beginning with .G1 or .G2 is treated as a .G1 or .G2 macro, for compatibility with old versions of troff. Using **-C** also forces pure troff syntax on embedded font change commands when strings have the **size** attribute.

The **-h** flag prints a brief help message and exist. **--help** is a synonym for **-h**.

It is possible for someone to cause **grap** to fail by passing a bad format string and data to the **sprintf** command. If **grap** is integrated as part of the printing system, this could conceivably provided a path to breaching security on the machine. If you choose to use **grap** as part of a printing system run by the super-user, you should disable **sprintf** commands. This can be done by calling **grap** with the **-S** flag, setting the GRAP_SAFER environment variable, or compiling with the GRAP_SAFER preprocessor symbol defined. (The GNU configure script included with **grap** will define that preprocessor symbol if the **--with-grap-safe** option is given.)

The **grap** commands are sketched below. Refer to Kernighan and Bentley's paper for the details.

New versions of groff(1) will invoke **grap** if **-G** is given.

Commands

Commands are separated from one another by newlines or semicolons (;).

frame [*line_description*] [**ht** *height* | **wid** *width*] [[(**top**|**bottom**|**left**| **right**) *line_description*] ...]

frame [**ht** *height* | **wid** *width*] [*line_description*] [[(**top**|**bottom**|**left**| **right**) *line_description*] ...]

This describes how the axes for the graph are drawn. A *line_description* is a pic line description, e.g., dashed 0.5, or the literal solid. If the first *line_description* is given, the frame is drawn with that style. The default is solid. The height and width of the frame can also be specified in inches. The default line style can be over-ridden for sides of the frame by specifying additional parameters to **frame**.

If no plotting commands have been given before the **frame** command is issued, the frame will be output at that point in the plotting stream relative to embedded troff or pic commands. Otherwise the frame is output before the first plotted object (even invisible ones).

coord [*name*] [**x** *expr*, *expr*] [**y** *expr*, *expr*] [**log x** | **log y** | **log log**]

The **coord** command specifies a new coordinate system or sets limits on the default system. It defines the largest and smallest values that can be plotted, and therefore the scale of the data in the frame. The limits for the x and y coordinate systems can be given separately. If a *name* is given, that coordinate system is defined, if not the default system is modified.

A coordinate system created by one **coord** command may be modified by subsequent **coord** commands. A **grap** program may declare a coordinate space using **coord**, **copy** a file of data through a macro that plots the data and finds its maxima and minima, and then define the size of the coordinate system with a second **coord** statement.

This command also determines if a scale is plotted logarithmically. **log log** means the same thing as **log x log y**.

draw [*line_name*] [*line_description*] [*plot_string*]

The **draw** command defines the style with which a given line will be plotted. If *line_name* is given, the style is associated with that name, otherwise the default style is set. *line_description* is a pic line description, and the optional *plot_string* is a string to be centered at each point. The default line description is invis, and the default plotting string is a centered bullet, so by default each point is a filled circle, and they are unconnected. If points are being connected, each **draw** command ends any current line and begins a new one.

When defining a line style, that is the first **draw** command for a given line name, specifying no plot string means that there are to be no plot strings. Omitting the plot string on subsequent **draw** commands addressing the same named line means not to change the plot string. If a line has been defined with a plot string, and the format is changed by a subsequent **draw** statement, the plot string can be removed by specifying "" in the **draw** statement.

The plot string can have its format changed through several string_modifiers. String_modifiers are described in the description of the **plot** command.

new is a synonym for **draw**.

next [*line_name*] **at** [*coordinates_name*] *expr*, *expr* [*line_description*]

The **next** command plots the given point using the line style given by *line_name*, or the default if none is given. If *line_name* is given, it should have been defined by an earlier **draw** command, if not a new line style with that name is created, initialized the same way as the default style. The two expressions give the point's x and y values, relative to the optional coordinate system. That system should have been defined by an earlier **coord** command, if not, grap will exit. If the optional *line_description* is given, it overrides the style's default line description. You cannot override the plotting string. To use a different plotting string use the **plot** command.

The coordinates may optionally be enclosed in parentheses: (*expr*, *expr*)

quoted_string [*string_modifiers*] [, *quoted_string* [*string_modifiers*]] ... **at** [*coordinates_name*] *expr*, *expr*

plot *expr* [*format_string*] **at** [*coordinates_name*] *expr*, *expr*

These commands both plot a string at the given point. In the first case the literal strings are stacked above each other. The string_modifiers include the pic justification modifiers (**ljust**, **rjust**, **above**, and **below**), and absolute and relative size modifiers. See the pic documentation for the description of the justification modifiers. **grap** also supports the **aligned** and **unaligned** modifiers which are briefly noted in the description of the **label** command.

size *expr* sets the string size to *expr* points. If *expr* is preceded by a + or -, the size is increased or decreased by that many points.

In the second version, the *expr* is converted to a string and placed on the graph. *format_string* is a printf(3) format string. Only formatting escapes for printing floating point numbers make sense. The format string is only respected if the **sprintf** command is also active. See the description of **sprintf** for the various ways to disable it. **Plot** and **sprintf** respond differently when **grap** is running safely. **Sprintf** ignores any arguments, passing the format string through without substitution. **plot** ignores the format string completely, plotting *expr* using the "%g" format.

Points are specified the same way as for **next** commands, with the same consequences for undefined coordinate systems.

The second form of this command is because the first form can be used with a **grap sprintf** expression (See **Expressions**).

ticks (**left**|**right**|**top**|**bottom**)[(**in**|**out**) [*expr*]] [**on**|**auto** *coord_name*]

ticks (**left**|**right**|**top**|**bottom**) (**in**|**out**) [*expr*] [**up** *expr* | **down** *expr* | **left** *expr* | **right** *expr*] **at** [*coord_name*] *expr* [*format_string*] [[, *expr* [*format_string*]] ...]

ticks (**left**|**right**|**top**|**bottom**) (**in**|**out**) [*expr*] [**up** *expr* | **down** *expr* | **left** *expr* | **right** *expr*] **from** [coord_name] *start_expr* **to** *end_expr* [**by** [+|-|*|/] *by_expr*] [format_string]

ticks [**left**|**right**|**top**|**bottom**] **off**

This command controls the placement of ticks on the frame. By default, ticks are automatically generated on the left and bottom sides of the frame.

The first version of this command turns on the automatic tick generation for a given side. The **in** or **out** parameter controls the direction and length of the ticks. If a *coord_name* is specified, the ticks are automatically generated using that coordinate system. If no system is specified, the default coordinate system is used. As with **next** and **plot**, the coordinate system must be declared before the **ticks** statement that references it. This syntax for requesting automatically generated ticks is an extension, and will not port to older **grap** implementations.

The second version of the **ticks** command overrides the automatic placement of the ticks by specifying a list of coordinates at which to place the ticks. If the ticks are not defined with respect to the default coordinate system, the *coord_name* parameter must be given. For each tick a `printf`(3) style format string can be given. The *format_string* defaults to "%g". The format string can also take string modifiers as described in the **plot** command. To place ticks with no labels, specify *format_string* as "".

If **sprintf** is disabled, **ticks** behaves as **plot** with respect to the format string.

The labels on the ticks may be shifted by specifying a direction and the distance in inches to offset the label. That is the optional direction and expression immediately preceding the **at**.

The third format of the **ticks** command over-rides the default tick generation with a set of ticks ar regular intervals. The syntax is reminiscent of programming language for loops. Ticks are placed starting at *start_expr* ending at *end_expr* one unit apart. If the **by** clause is specified, ticks are *by_expr* units apart. If an operator appears before *by_expr* each tick is operated on by that operator instead of +. For example

```
ticks left out from 2 to 32 by *2
```

will put ticks at 2, 4, 8, 16, and 32. If *format_string* is specified, all ticks are formatted using it.

The parameters preceding the **from** act as described above.

The **at** and **for** forms of tick command may both be issued on the same side of a frame. For example:

```
ticks left out from 2 to 32 by *2
ticks left in 3, 5, 7
```

will put ticks on the left side of the frame pointing out at 2, 4, 8, 16, and 32 and in at 3, 5, and 7.

The final form of **ticks** turns off ticks on a given side. If no side is given the ticks for all sides are cancelled.

tick is a synonym for **ticks**.

grid (**left**|**right**|**top**|**bottom**) [`ticks off`] [*line_description*] [**up** *expr* | **down** *expr* | **left** *expr* | **right** *expr*] [**on**|**auto** [*coord_name*]]

grid (**left**|**right**|**top**|**bottom**) [`ticks off`] [*line_description*] [**up** *expr* | **down** *expr* | **left** *expr* | **right** *expr*] **at** [*coord_name*] *expr* [*format_string*] [[, *expr* [*format_string*]] ...]

grid (**left**|**right**|**top**|**bottom**) [`ticks off`] [*line_description*] [**up** *expr* | **down** *expr* | **left** *expr* | **right** *expr*] **from** [coord_name] *start_expr* **to** *end_expr* [**by** [+|-|*|/] *by_expr*] [format_string]

The **grid** command is similar to the **ticks** command except that **grid** specifies the placement of lines in the frame. The syntax is similar to **ticks** as well.

By specifying `ticks off` in the command, no ticks are drawn on that side of the frame. If ticks appear on a side by default, or have been declared by an earlier **ticks** command, **grid** does not cancel them unless `ticks off` is specified.

Instead of a direction for ticks, **grid** allows the user to pick a line description for the grid lines. The usual `pic` line descriptions are allowed.

Grids are labelled by default. To omit labels, specify the format string as "".

If **sprintf** is disabled, **grid** behaves as **plot** with respect to the format string.

label (**left**|**right**|**top**|**bottom**) *quoted_string* [*string_modifiers*] [, *quoted_string* [*string_modifiers*]] ... [**up** *expr* | **down** *expr* | **left** *expr* | **right** *expr*]

The **label** command places a label on the given axis. It is possible to specify several labels, which will be stacked over each other as in pic. The final argument, if present, specifies how many inches the label is shifted from the axis.

By default the labels on the left and right labels run parallel to the frame. You can cancel this by specifying unaligned as a *string_modifier*.

circle at [*coordinate_name*] *expr*, *expr* [**radius** *expr*] [*linedesc*]

This draws an circle at the point indicated. By default, the circle is small, 0.025 inches. This can be over-ridden by specifying a radius. The coordinates of the point are relative to the named coordinate system, or the default system if none is specified.

This command has been extended to take a line description, e.g., dotted. It also accepts the filling extensions described below in the **bar** command.

line [*line_description*] **from** [*coordinate_name*] *expr*, *expr* **to** [*coordinate_name*] *expr*, *expr* [*line_description*]

arrow [*line_description*] **from** [*coordinate_name*] *expr*, *expr* **to** [*coordinate_name*] *expr*, *expr* [*line_description*]

This draws a line or arrow from the first point to the second using the given style. The default line style is solid. The *line_description* can be given either before the **from** or after the **to** clause. If both are given the second is used. It is possible to specify one point in one coordinate system and one in another, note that if both points are in a named coordinate system (even if they are in the same named coordinate system), both points must have *coordinate_name* given.

copy ["*filename*"] [**until** "*string*"] [**thru** *macro*]

The **copy** command imports data from another file into the current graph. The form with only a filename given is a simple file inclusion; the included file is simply read into the input stream and can contain arbitrary **grap** commands. The more common case is that it is a number list; see **Number Lists** below.

The second form takes lines from the file, splits them into words delimited by one or more spaces, and calls the given macro with those words as parameters. The macro may either be defined here, or be a macro defined earlier. See **Macros** for more information on macros.

The *filename* may be omitted if the **until** clause is present. If so the current file is treated as the input file until *string* is encountered at the beginning of the line.

copy is one of the workhorses of **grap**. Check out the paper and /usr/local/share/doc/grap/examples for more details.

print (*expr*|*string*)

Prints its argument to the standard error.

sh *block*

This passes *block* to sh(1). Unlike K&B **grap** no macro or variable expansion is done. I believe that this is also true for gnu pic version 1.10. See the **Macros** section for information on defining blocks.

pic *pic_statement*

This issues the given `pic` statements in the enclosing **.PS** and **.PE** at the point where the command is issued.

Statements that begin with a period are considered to be `troff`(statements) and are output in the enclosing **.PS** and **.PE** at the point where the command appears.

For the purposes of relative placement of `pic` or `troff` commands, the frame is output immediately before the first plotted object, or the **frame** statement, if any. If the user specifies `pic` or `troff` commands and neither any plotable object nor a **frame** command, the commands will not be output.

graph *Name pic_commands*

This command is used to position graphs with respect to each other. The current graph is given the `pic` name *Name* (`pic` names begin with capital letters). Any `pic` commands following the graph are used to position the next graph. The frame of the graph is available for use with `pic` name `Frame.` The following places a second graph below the first:

```
graph Linear
[ graph description ]
graph Exponential with .Frame.n at \
        Linear.Frame.s - (0, .05)
[ graph description ]
```

name = *expr*

This assigns *expr* to the variable *name*. **grap** has only numeric (double) variables.

Assignment creates a variable if it does not exist. Variables persist across graphs. Assignments can cascade; `a = b = 35` assigns 35 to `a` and `b`.

bar (**up**|**right**) [*coordinates_name*] *offset* **ht** *height* [**wid** *width*] [**base** *base_offset*] [*line_description*]

bar [*coordinates_name*] *expr*, *expr*, [*coordinates_name*] *expr*, *expr*, [*line_description*]

The **bar** command facilitates drawing bar graphs. The first form of the command describes the bar somewhat generally and has **grap** place it. The bar may extend up or to the right, is centered on *offset* and extends up or right *height* units (in the given coordinate system). For example

```
bar up 3 ht 2
```

draws a 2 unit high bar sitting on the x axis, centered on x=3. By default bars are 1 unit wide, but this can be changed with the **wid** keyword. By default bars sit on the base axis, i.e., bars directed up will extend from y=0. That may be overridden by the **base** keyword. (The bar described above has corners (2.5, 0) and (3.5, 2).)

The line description has been extended to include a **fill** *expr* keyword that specifies the shading inside the bar. Bars may be drawn in any line style.

The second form of the command draws a box with the two points as corners. This can be used to draw boxes highlighting certain data as well as bar graphs. Note that filled bars will cover data drawn under them.

Control Flow

if *expr* **then** *block* [**else** *block*]

The **if** statement provides simple conditional execution. If *expr* is non-zero, the *block* after the **then** statement is executed. If not the *block* after the **else** is executed, if present. See **Macros** for the definition of blocks. Early versions of this implementation of **grap** treated the blocks as macros that were defined and expanded in place. This led to unnecessary confusion because explicit separators were sometimes called for. Now, **grap** inserts a separator (;) after the last character in *block*, so constructs like

```
if (x == 3) { y = y + 1 }
x = x + 1
```

behave as expected. A separator is also appended to the end of a **for** block.

for *name* **from** *from_expr* **to** *to_expr* [**by** [+|-|*|/] *by_expr*] **do** *block*

This command executes *block* iteratively. The variable *name* is set to *from_expr* and incremented by *by_expr* until it exceeds *to_expr*. The iteration has the semantics defined in the **ticks** command. The definition of *block* is discussed in **Marcos**. See also the note about implicit separators in the description of the **if** command.

An **=** can be used in place of **from**.

Expressions

grap supports a most standard arithmetic operators: + - / * ^. The carat (^) is exponentiation. In an **if** statement **grap** also supports the C logical operators ==, !=, &&, || and unary !. Also in an **if**, == and != are overloaded for the comparison of quoted strings. Parentheses are used for grouping.

Assignment is not allowed in an expression in any context, except for simple cascading of assignments. `a = b = 35` works as expected; `a = 3.5 * (b = 10)` does not execute.

grap supports the following functions that take one argument: **log**, **exp**, **int**, **sin**, **cos**, **sqrt**, **rand**. The logarithms are base 10 and the trigonometric functions are in radians. **eexp** returns Euler's number to the given power and **ln** returns the natural logarithm. The natural log and exponentiation functions are extensions and are probably not available in other **grap** implementations.

rand returns a random number uniformly distributed on [0,1). The following two argument functions are supported: **atan2**, **min**, **max**. **atan2** works just like `atan2`(3). The random number generator can be seeded by calling **srand** with a single parameter (converted internally to an integer). Because its return value is of no use, you must use **srand** as a separate statement, it is not part of a valid expression. **srand** is not portable.

Other than string comparison, no expressions can use strings. One string valued function exists: **sprintf** (*format*, [*expr* [*, expr*]]). It operates like `sprintf`(3), except returning the value. It can be used anywhere a quoted string is used. If **grap** is run with **−S**, the environment variable `GRAP_SAFER` is defined, or **grap** has been compiled for safer operation, the **sprintf** command will return the format string. This mode of operation is only intended to be used only if **grap** is being used as part of a super-user enabled print system.

Macros

grap has a simple but powerful macro facility. Macros are defined using the **define** command :

define *name block*
undefine *name*

Every occurrence of *name* in the program text is replaced by the contents of *block*. *block* is defined by a series of statements in nested { }'s, or a series of statements surrounded by the same letter. An example of the latter is

```
define foo  X coord x 1,3 X
```

Each time `foo` appears in the text, it will be replaced by `coord x 1,3`. Macros are literal, and can contain newlines. If a macro does not span multiple lines, it should end in a semicolon to avoid parsing errors.

Macros can take parameters, too. If a macro call is followed by a parenthesized, comma-separated list the values starting with $1 will be replaced in the macro with the elements of the list. A $ not followed by a digit is left unchanged. This parsing is very rudimentary, no nesting or parentheses or escaping of commas is allowed. Also, there is no way to say argument 1 followed by a digit (${1}0 in sh(1)).

The following will draw a line with slope 1.

```
define foo { next at $1, $2 }
for i from 1 to 5 { foo(i,i) }
```

Macros persist across graphs. The file `/usr/local/share/grap/grap.defines` contains simple macros for plotting common characters. The **undefine** command deletes a macro.

See the file `/usr/local/share/doc/grap/examples` for more examples of macros.

Number Lists

A whitespace-separated list of numbers is treated specially. The list is taken to be points to be plotted using the default line style on the default coordinate system. If more than two numbers are given, the extra numbers are taken to be additional y values to plot at the first x value. Number lists in DWB **grap** can be comma-separated, and this **grap** supports that as well. More precisely, numbers in number lists can be separated by either whitespace, commas, or both.

```
1 2 3
4 5 6
```

Will plot points using the default line style at (1,2), (1,3),(4,5) and (4,6). A simple way to plot a set of numbers in a file named `./data` is:

```
.G1
copy "./data"
.G2
```

Pic Macros

grap defines pic macros that can be used in embedded pic code to place elements in the graph. The macros are **x_gg**, **y_gg**, and **xy_gg**. These macros define pic distances that correspond to the given argument. They can be used to size boxes or to plot pic constructs on the graph. To place a given construct on the graph, you should add Frame.Origin to it. Other coordinate spaces can be used by replacing **gg** with the name of the coordinate space. A coordinate space named **gg** cannot be reliably accessed by these macros.

The macros are emitted immediately before the frame is drawn.

DWB **grap** may use these as part of its implementation. This **grap** provides them only for compatibility. Note that these are very simple macros, and may not do what you expect under complex conditions.

ENVIRONMENT VARIABLES

If the environment variable `GRAP_DEFINES` is defined, **grap** will look for its defines file there. If that value is a relative path name the path specified in the **-M** option will be searched for it. `GRAP_DEFINES` overrides the compiled in location of the defines file, but may be overridden by the **-d** or **-D** flags.

If `GRAP_SAFER` is set, **`sprintf`** is disabled to prevent forcing **`grap`** to core dump or smash the stack.

FILES

`/usr/local/share/grap/grap.defines`

SEE ALSO

`atan2`(3), `groff`(1), `pic`(1), `printf`(3), `sh`(1), `sprintf`(3), `troff`(1)

BUGS

There are several small incompatibilities with K&R **`grap`**. They include the **`sh`** command not expanding variables and macros, and a more strict adherence to parameter order in the internal commands.

Although much improved, the error reporting code can still be confused. Notably, an error in a macro is not detected until the macro is used, and it produces unusual output in the error message.

Iterating many times over a macro with no newlines can run **`grap`** out of memory.

AUTHOR

This implementation was done by Ted Faber ⟨faber@lunabase.org⟩. Bruce Lilly ⟨blilly@erols.com⟩ contributed many bug fixes, including a considerable revamp of the error reporting code. If you can actually find an error in your **`grap`** code, you can probably thank him. **`grap`** was designed and specified by Brian Kernighan and Jon Bentley.