

# Drawing with dpic

Dwight Aplevich<sup>1</sup>  
2023.02.01

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>	4.1	SVG text . . . . .	20
<b>2</b>	<b>Dpic usage</b>	<b>2</b>	4.2	Bounding boxes . . . . .	21
2.1	Options . . . . .	3	<b>5</b>	<b>Pic processor differences</b>	<b>23</b>
2.2	Error messages . . . . .	4	5.1	Command-line options . . . . .	23
<b>3</b>	<b>Dpic programming</b>	<b>5</b>	5.2	Output formats . . . . .	23
3.1	Character set . . . . .	5	5.3	“.” and “\” lines . . . . .	23
3.2	Blocks . . . . .	5	5.4	for-loop and if bodies . . . . .	23
3.2.1	Positioning blocks . . . . .	5	5.5	Closing braces and nul elements	24
3.2.2	Defining scope . . . . .	6	5.6	End of line . . . . .	24
3.3	Dpic macros . . . . .	7	5.7	Logic . . . . .	25
3.3.1	Finding roots . . . . .	8	5.8	Defaults for <code>line</code> , <code>spline</code> , <code>arrow</code> , <code>move</code> . . . . .	25
3.3.2	Composing statements . . . . .	9	5.9	Arc defaults . . . . .	25
3.3.3	Branching . . . . .	9	5.10	Strings . . . . .	26
3.3.4	Looping . . . . .	9	5.11	<code>print arg</code> , . . . . .	27
3.3.5	Evaluating arguments . . . . .	10	5.12	<code>command arg</code> , . . . . .	27
3.3.6	Hiding variables . . . . .	11	5.13	Operating system commands . . . . .	27
3.3.7	Complex numbers . . . . .	12	5.14	<code>copy</code> . . . . .	27
3.4	M4 macros . . . . .	12	5.15	<code>plot</code> . . . . .	28
3.4.1	M4 branching . . . . .	12	5.16	<code>fill</code> . . . . .	28
3.4.2	Extending pic: perpen- diculars . . . . .	13	5.17	Scaling . . . . .	28
3.4.3	Setting directions . . . . .	13	5.18	Arrowheads . . . . .	29
3.5	Subscripts . . . . .	14	5.19	Compass corners and justified text . . . . .	30
3.5.1	Assigning an array of numbers . . . . .	15	5.20	<code>continue</code> . . . . .	30
3.6	Splines . . . . .	15	5.21	Subscripted variables and labels	30
3.6.1	Curve fitting . . . . .	16	5.22	Splines . . . . .	31
3.7	Text . . . . .	16	5.23	Arithmetic . . . . .	31
3.8	Postprocessor commands and color . . . . .	17	5.24	Vector arithmetic . . . . .	31
3.8.1	Color . . . . .	18	5.25	<code>exec</code> . . . . .	31
3.8.2	Filling with color . . . . .	18	5.26	Functions . . . . .	31
3.8.3	External files . . . . .	19	<b>6</b>	<b>References</b>	<b>32</b>
<b>4</b>	<b>SVG, PDF, and Postscript output</b>	<b>20</b>	<b>7</b>	<b>Appendix A: dpic man page</b>	<b>33</b>
<b>8</b>	<b>Appendix B: dpic grammar</b>	<b>46</b>			

---

<sup>1</sup>Copyright © 2023 J. D. Aplevich, all rights reserved. This document version is made available under the Creative Commons attribution licence version 3.0 (<http://creativecommons.org/licenses/by/3.0/>); you are free to copy and distribute this document provided proper attribution is given by identifying the author.

# 1 Introduction

This document is meant for persons using dpic to produce diagrams for L<sup>A</sup>T<sub>E</sub>X documents or web files. You are assumed to have basic knowledge of the pic language as described for the original Documenter’s Workbench (ATT) pic [3] or the current GNU pic (gpic) processor [4]. However, a concise dpic language reference is included here in Appendix A (Section 7) and the context-free dpic input grammar is given in Appendix B (Section 8).

The pic “little language” is particularly suited for easily creating line diagrams such as electric circuits, and many persons use a set of macros called Circuit\_macros, which are processed using the m4 macro processor and dpic. The pic language itself allows macro definitions, and both pic and m4 macros will be described.

Many other software tools can produce line diagrams and the choice of tool is often subjective or a matter of prior familiarity. The advantages of dpic are that its basic use is easy to learn and that arbitrary formatted diagram text can be included using L<sup>A</sup>T<sub>E</sub>X. Dpic will also produce SVG output for web files but the advantage of employing a word processor for text is not then directly available. A route to sophisticated text in SVG is to use L<sup>A</sup>T<sub>E</sub>X to produce a pdf file, followed by a pdf-to-svg converter.

Dpic accepts gpic input for the most part but there are minor differences. The outputs of gpic and dpic are quite different but both can serve as preprocessors that create diagrams for inclusion in L<sup>A</sup>T<sub>E</sub>X documents.

Dpic usage will be reviewed, some programming examples that illustrate dpic extensions of the pic language will be given, and then the differences among pic translators will be itemized, particularly the differences between gpic and dpic.

There had better be a disclaimer: the temptation to change the pic language has been resisted most of the time so that valid diagrams can be processed with minimal changes using the original Documenter’s Workbench (ATT) pic, with gpic, or dpic. There are exceptions: embedded word-processor commands principally, but also minor differences in defaults for valid pic input, a few gpic constructs that dpic does not implement directly, and a few dpic extensions that gpic does not implement. For details, see Section 5.

## 2 Dpic usage

In the following, items in square brackets [ ] are optional and items separated by a vertical line | are alternatives. To produce .tex output (for L<sup>A</sup>T<sub>E</sub>X, PSTricks, TikZ-PGF, mfpic processing):

```
dpic [options] file.pic > file.tex
or
cat file.pic | dpic [options] > file.tex
```

To produce other formats:

```
dpic [-d|e|f|g|h|m|p|r|s|t|v|x] [-z] file.pic > file[.tex|eps|pdf|mp|fig|svg]
```

## 2.1 Options

Dpic accepts the following options:

- (none) Latex picture-environment output (very limited font-based drawing commands)
- d PDF output
- e pict2e output
- f Postscript output with psfrag strings
- g TikZ-PGF output
- h write out these options and quit
- m mfpic output
- p PSTricks output
- r raw Postscript output
- s MetaPost output
- t eepicemu output
- v SVG output
- x xfig 3.2 output
- z safe mode (disabled `sh`, `copy`, and print to file)

The `-p` option produces output for postprocessing by the  $\text{\LaTeX}$  package PSTricks. Similarly, the `-g` option produces output for the TikZ-PGF packages and makes pdf production via `pdflatex` a one-step process.

Mfpic and MetaPost output are provided for compatibility.

The `-r` option produces Postscript `.eps` files, in which font changes or typesetting must be done explicitly. The `-f` option writes Postscript strings in psfrag format for tex or latex typesetting.

The `-d` option produces PDF files. This format is most useful for diagrams containing graphics only, since there is no simple way to change or manipulate fonts for labels. The Courier fixed-width font is employed by default. To produce PDF files containing significant text content, use an option such as TikZ with `pdflatex`.

The `-v` option produces SVG for inserting figures into web documents or for further processing by the Inkscape graphics editor, for which SVG is the native format. When the SVG output is used directly in a web document, then any required text formatting generally must be included explicitly. One other possibility for SVG output is that an SVG library of elements can be drawn with dpic, and Inkscape used to place and connect copies of the elements. Then Inkscape can export the graphics as eps for processing by  $\text{\LaTeX}$  or as pdf for processing by `pdflatex`. Inkscape will also export a tex file from which labels can be formatted and overlaid on the imported eps or pdf. A third possibility is to produce the graphic without text and then overlay the text elements.

The `-z` option disables the commands that access external files. These commands can be permanently disabled by the use of a compile-time option.

In all cases, arbitrary postprocessor commands (that is, arbitrary PSTricks, SVG, Postscript, or other code) can be inserted into the dpic output directly from the source. This possibility adds considerable power for manipulating diagram elements.

The use of output formats meant to be processed by L<sup>A</sup>T<sub>E</sub>X is quite different from the other formats. If a word processor is not part of the process, the selection and formatting of diagram text requires the insertion of postprocessor commands into the output. Dpic knows nothing of text formatting except for SVG output, for which basic sizing and placement are provided.

The file Examples.txt contains a minimal example of each of these options except `-z`. Consult the appropriate manual for processing mfpic, PSTricks, MetaPost, pgf, or psfrag output.

Invoking dpic without options produces basic L<sup>A</sup>T<sub>E</sub>X drawing commands by default. L<sup>A</sup>T<sub>E</sub>X line slopes, for example, are very limited and you must ensure that lines and arrows are drawn only at acceptable slopes. Dpic sets the maximum slope integer to be 6 for L<sup>A</sup>T<sub>E</sub>X, 453 for eepic, and 1000 for pict2e. To see the effect of the slope limitations, process the following:

```
.PS
dtor = atan2(0,-1)/180
for d = 0 to 360 by 2 do {
  line from (0,0) to (cos(d*dtor),sin(d*dtor))
}
line from (-1.2,0) to (1.2,0)
line from (0,-1.2) to (0,1.2)
.PE
```

## 2.2 Error messages

The following is an example of an error message:

```
*** dpic ERROR: in macro "A", file "B.pic" line 16:
x = y
Variable not found
Search failure for "y"
```

Dpic displays the input file name with the number of the line to which scanning has progressed. The name of the current macro is also given if one is being executed. Most often, scanning stops on or near the offending line, which is displayed up to where reading ceased. However, the source of the problem might be a considerable distance from the indicated line number if the error is in a for loop or a macro. In the above example, line 16 of file B.pic happens to be

```
A(5)
```

which invokes macro A, so one has to find the definition of A and look for the line

```
x = y
within it.
```

Syntax error messages show the offending line and state the lexical object that was found at or immediately before the error, together with expected (permissible) objects. Other errors are signalled according to the circumstances.

## 3 Dpic programming

Pic is a simple language with a good ratio of power to complexity, so surprisingly sophisticated diagrams can be produced in several areas of application. Arbitrary postprocessor commands can be included in the output and several of the postprocessors are powerful drawing languages in their own right, so it can be argued that dpic has all the power of these languages. However, if you find yourself writing extensive postprocessor code then you might ask why you are not programming exclusively in that language. Similarly, pic floating-point operations and macro facilities enable complex numerical computations; the price is the inefficiency of number-crunching in an interpreted language. However, it can be convenient to calculate plotted values in the same source as graphing them and, although pic was developed for simplicity, its implementations are Turing-complete so arbitrarily complex computation is possible in principle.

The pic language is very suitable for line diagrams with basic color. Elaborate fills and cropping are the domain of the postprocessors, but can be included easily with the use of macros.

The following sections are intended to help you become familiar with dpic language syntax, which is simple but somewhat unusual. Some of these details are exclusive to dpic and are not described in the GNU manual but are discussed in [Section 5](#) of this document.

### 3.1 Character set

Dpic input employs ordinary ASCII characters. The characters LF (`\n`) and CR (`\r`) have identical effect, and ETX (`\003`) is reserved for internal use. Characters outside the range of 0 to 127 are allowed in strings but receive no special treatment.

### 3.2 Blocks

The basic planar objects in pic are `box`, `circle`, and `ellipse`, the placing of which is done according to the current drawing direction or by explicit placement such as

```
box at position
```

which places the object so that its center is at *position*.

A block (or a composite object) is a group of elements enclosed by square brackets, such as

```
Q: [ B: [A: arc]; circle ].
```

#### 3.2.1 Positioning blocks

A block is placed by default as if it were a box, after which the compass points (`Q.n`, `Q.sw`, ... in the previous example) are automatically defined as for a box of the same size and position.

A block can also be positioned by specifying the location of one of its defined points. A defined point is one of the following:

1. A compass corner `.center`, `.n`, `.ne`, ... of the block, e.g.,  
`Q: [ B: [A: arc]; circle ] with .ne at position`
2. A defined point of a labeled object or position within the block, preceded by a dot, e.g.,  
`Q: [ B: [A: arc]; circle ] with .B.A.ne at position`
3. A defined point of an enumerated object in the block, preceded by a dot (but make sure there is a space after the dot if it is followed by a number), e.g.,  
`Q: [ B: [A: arc]; circle ] with . 1st circle at position`  
 Even better, put braces around the ordinal value, which can now be any expression, e.g.,  
`... with .{10-9}th circle at ...`
4. A displacement `(x,y)` from the lower left corner of the block, e.g.,  
`Q: [ B: [A: arc]; circle ] with (0.5,0.2) at position`

Reference to a defined point may correspond to drilling down through several block layers, as the second example above shows.

### 3.2.2 Defining scope

Variables defined within a block are accessible only within the block itself or its sub-blocks. Thus, the statement `x = 5` creates the variable `x` and assigns it a value. If the statement is `x := 5` then `x` must already have been defined either in the block or in a scope containing the block. Limiting the scope to a block avoids name conflicts but occasionally results in subtle behavior. The code

```
x = 5
[ x = x + 2; print x ]
print x
```

prints 7 on the first line and 5 on the second because the assignment within brackets creates variable `x` with scope within the brackets. However, Dpic allows a second line of the form

```
[ x += 2; print x ]
```

which changes the previously defined variable `x` and the value 7 is printed twice.

Locations inside a block are accessible from outside as shown in the previous section, but the values of variables are not; thus, an error results from

```
Q: [ v = 5 ]; y = Q.v
```

The `:=` assignment operator in

```
v = 0; Q: [ v := 5 ]; y = v
```

works around this problem, but this method requires the internal name `v` to be known and defined in advance of the block.

Variable values can be exported by the use of macro arguments as shown later in [Section 3.3](#) or, if you must, by using the following trick:

```
Q: [ v = 5; w = 6
    Origin: (0,0); Export: (v,w) ]
v = Q.Export.x - Q.Origin.x
w = Q.Export.y - Q.Origin.y
```

This method works because all locations (`Origin` and `Export`) inside the block will be translated by the same amount no matter where the block is positioned.

### 3.3 Dpic macros

Macros can be used to turn the basic dpic language into a powerful tool for producing line drawings. A macro serves to

- specialize the pic language in order to draw components from an application area
- abbreviate long sequences of repetitive commands
- substitute particular values in commands by the use of macro arguments
- provide conditional text replacement depending on the value of macro arguments
- provide recursive looping

Macro definitions are not local to blocks so care must be taken to avoid conflicts with macro names.

The pic language includes basic macro facilities, but the m4 macro processor [2, 5] makes a good companion to the language, so both will be mentioned. General-purpose macro definitions can be stored in files external to the diagram source and read in automatically. In particular, the author has written a package called Circuit\_macros [1] for drawing electric circuits and other diagrams using dpic and m4, from which examples will be taken.

A dpic macro is defined by the statement

```
define name { contents }
```

where *name* may begin with a lowercase or uppercase letter. The form

```
define name X contents X
```

is also allowed, where X is any character except {. Then any separate appearance of *name* in the following lines is replaced by the *contents* characters, including newlines. If the name is given arguments of the form

```
name(x, y, z, ...)
```

then, in the macro body, *\$n* expands to the *n*th argument, which may be nul.

The line

```
undefine name
```

deletes the macro definition.

Dpic skips white space immediately preceding macro arguments so that, for example,

```
name( x,  
      y, z )
```

is equivalent to

```
name(x,y,z )
```

In a macro invocation, the arguments are separated by commas. An argument may contain commas if they are between ( ) parentheses, [ ] brackets, or in strings. A double quote character within a string must be preceded by a backslash. Thus, for example, the macro invocation

```
name(ABc\"t\", \"(,DE,F))
```

has one argument.

In a dpic macro, the value of `$+` is the number of arguments given to the macro on invocation. Thus if `x` is a macro name, the values of `$+` when the macro is invoked as `x`, `x()`, `x(8)`, `x(8,9)`, and `x(,,)` are respectively 0, 1, 1, 2, and 3.

To check whether an argument is null, put it in a string; for example,

```
if "$3" == "" then { ... }
```

Values internal to a scope can be passed back through macro arguments with the `:=` operator; thus, executing the following

```
define m {[ v = 5; $1 := v ]}; y = 0; m(y)
```

gives `y` the internal value of `v`. Notice that `y` must be defined prior to macro expansion.

The use of macros will be illustrated in the following examples, for which source is included with this distribution and in the `Circuit_macros` distribution [1].

### 3.3.1 Finding roots

A root finder is a powerful tool for determining where lines or curves intersect in diagrams, and can be implemented using a macro. Consider the trivial example in which we wish to find the root of  $x^2 - 1$  between 0 and 2. First, define a macro called `bisect` by reading in a library file containing definitions using a command such as

```
copy "filename"
```

or by writing a definition such as given below, which employs the method of bisection, a suitably robust (but not particularly fast) algorithm. Define the two-argument macro corresponding to the function of which we want to calculate the root:

```
define parabola { $2 = ($1)^2 - 1 }
```

In general, many statements might be required to calculate the function, but the essential statement is to assign the function value to the name given by the second argument. Then call the `bisect` macro using a command such as

```
bisect( parabola, 0, 2, 1e-8, x )
```

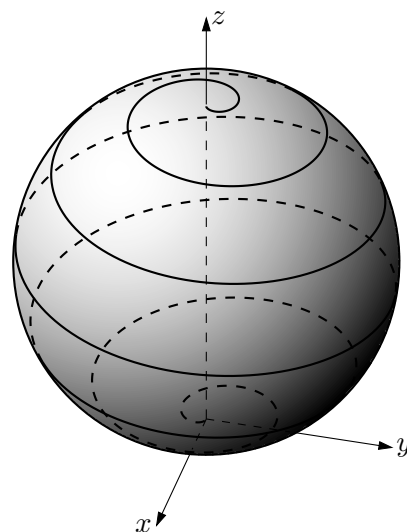
where the second and third arguments define the search interval, the fourth argument specifies the solution precision, and the fifth argument is the name of the variable to be set to the root. A basic version of `bisect` is given by

```
define bisect {
  x_m = $2; x_M = $3
  x_c = (x_m+x_M)/2
  if (abs(x_m-x_M) <= $4) then { $5 = x_c } else {
    $1(x_m,f_m)
    $1(x_c,f_c)
    if (sign(f_c)==sign(f_m)) then { bisect($1,x_c,x_M,$4,$5) } \
    else { bisect($1,x_m,x_c,$4,$5) } }
}
```

This macro repeatedly calls `parabola` and then itself, halving the search interval until it is smaller than the prescribed precision. The `Circuit_macros` library version operates essentially as above but avoids name clashes by appending the value of the first argument to the internal names.



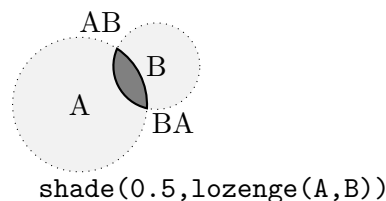
A somewhat more sophisticated use of a root finder is shown at the right. Drawing the spiral on the surface of the sphere requires knowing the points of transition of the curve from visible to hidden and back. A root finder provides a method that is both simple and adaptable to other shapes. The source file `Spiral.m4` makes use of both `m4` macros and `pic` macros.



### 3.3.2 Composing statements

Dpic macro arguments can be expanded almost anywhere. Suppose that circles A and B have been defined, with intersections at positions AB and BA found using the macro `cintersect` from `Circuit_macros`, for example. Then the boundary of the region within both circles might be drawn using the macro shown, invoked as `lozenge(A,B)`:

```
define lozenge {
  arc from $1$2 to $2$1 with .c at $2
  arc from $2$1 to $1$2 with .c at $1 }
```



### 3.3.3 Branching

Pic has a basic if-statement of the form

```
if expression then { if-true } else { if-false }
```

but lacks a case statement. Multiple branches can be defined by nested `if` statements but there is another way. The macro

```
define case { exec sprintf("%g",floor($1+0.5)+1); }
```

adds 1 to its rounded first argument to determine which alternative among the remaining arguments should be executed. The semicolon (or a newline) forces dpic to perform the `exec` statement before leaving the macro. For example,

```
case(2,
  print "A",
  print "B")
```

executes the second alternative (the third argument) and prints B.

### 3.3.4 Looping

The pic language includes a basic for-loop, such as the following:

```
for x = 1 to 10 by 2 do { print x }
```

but there is no explicit language element (except macro recursion) for executing a block of code an indefinite number of times. However, the `for` variable can be reset within the executable code, as in the following example where the first macro argument is printed and doubled repeatedly until it becomes greater than the second argument:

```
define series { x = $1; for done = 0 to 1 do { print x; x *= 2
  done = (x > $2) } }
```

If this trick seems like an abuse of language, it can be disguised somewhat by the definition of a C-like loop. For example, suppose we wish to write

```
loop(i=0, i<3, i+=1, drawing commands ...)
```

Then the following defines a suitable mechanism with a loop depth index to allow nesting:

```
ld__ = 0
define loop { ld__+= 1; $1
  for lx__[ld__]=0 to 1 do {
    if $2 then { lx__[ld__]=0; $4; $3 } else {lx__[ld__]=1}}
  ld__-= 1; }
```

However, `loop()` is a macro, so references to arguments in the body will refer to `loop()` arguments, which may not be desired. In that case, use a `for` loop. The following prototypical example shows how to implement the previous example using a *repeat-until* structure that executes the drawing commands 1 or more times:

```
i = 0; define finished {($1 >= 3)}
for done = 0 to 1 do {
  drawing commands ...
  i += 1
  done = finished(i) }
```

To execute the drawing code 0 or more times as in a *while* loop, change the `for` line to

```
for done = 1 to !finished(i) do {
```

If such loops are nested, the loop variables and stopping criteria have to be defined so they do not clash.

### 3.3.5 Evaluating arguments

A macro argument is referenced as  $\$n$ , where  $n$  must be an integer and may not be an integer expression. Consequently, the following does not work in a dpic macro:

```
for j=1 to $+ do { c[j] = $j }
```

Instead, use the dpic statement

```
exec string
```

which executes the contents of *string* as if it were the current input line. Since

```
sprintf("format", expression,...)
```

behaves like a string, the following works:

```
for j=1 to $+ do { exec sprintf("c[j] = $g",j); }
```

Macro arguments are passed verbatim, as for example, `function(x,y+z)`, inside which  $\$1$  is replaced by `x` and  $\$2$  is replaced by `y+z`. To pass arguments by value, use `exec sprintf` ..., for example, `exec sprintf("function(%g,%g)",x,y+z)`. The recursive calls near the bottom of the sorting algorithm shown are another example:

```

#                               dpquicksort(a,lo,hi,ix)
#                               Given array a[lo:hi] and index
#                               array ix[lo:hi] = lo,lo+1,lo+2,...hi, sort
#                               a[lo:hi] and do identical exchanges on ix
define dpquicksort { [ if $3 > $2 then {
    pivot = $1[(($2+($3))/2]
    loop(lo = $2; hi = $3, lo <= hi,
        loop($1[lo] < pivot, lo += 1)
        loop($1[hi] > pivot, hi -= 1)
        if lo < hi then {
            tmp = $1[lo]; $1[lo] := $1[hi]; $1[hi] := tmp
            tmp = $4[lo]; $4[lo] := $4[hi]; $4[hi] := tmp }
        if lo <= hi then { lo += 1; hi -= 1 } )
    if hi > $2 then { exec sprintf("dpquicksort($1,%g,%g,$4)", $2, hi) }
    if lo < $3 then { exec sprintf("dpquicksort($1,%g,%g,$4)", lo, $3) }
} ] }

```

### 3.3.6 Hiding variables

As mentioned in [Section 3.2.2](#), locations inside a block are accessible from outside, but the values of variables are not. Therefore, a block can be used to hide variables internal to a macro, as in the following example:

```

define rgbtohsv { $4=0; $5=0; $6=0; [
    r = $1; g = $2; b = $3
    maxc = max(max(r,g),b)
    minc = min(min(r,g),b)
    if maxc==minc then { $4 := 0 } \
    else {if maxc == r then { $4 := pmod(60*((g-b)/(maxc-minc)),360) } \
    else {if maxc == g then { $4 := 60*((b-r)/(maxc-minc)) + 120 } \
    else { $4 := 60*((r-g)/(maxc-minc)) + 240 }}}
    if maxc == 0 then { $5 := 0 } else { $5 := 1 - (minc/maxc) }
    $6 := maxc
] }

```

The three assignments in the first line of the example ensure that the variables exist when the `:=` assignments are performed.

This is not the full story, however. Macro arguments are called by name rather than by value; should the fourth argument, for example, be literally `x[minc]`, then the interior variable `minc` prevails. A block is only a partial solution to the problem of hiding variables, and care must be exercised in choosing the names of arguments. A more robust solution is to call by value using `exec`; thus,

```
exec sprintf("rgbtohsv(%g, %g, %g, h, s, v)",expr1,expr2,expr3)
```

In this case however, the last three arguments remain vulnerable to name clashes and should be named with care.

When, as in the example, no drawing commands appear in a [ ] block, then the block has zero size but has position **Here** so the block can affect the diagram bounding box if **Here** happens to be outside the intended bounding box of drawn elements. An alternative to this complication is to omit the [ ] brackets and rename the local variables to avoid name clashes. For example, `r` above could be `r_rgbtotshv` and so on for other variables.

### 3.3.7 Complex numbers

A complex number is an ordered pair of real numbers like a position, so it is natural to do complex algebra using macros such as the following example, which evaluates the product of two complex-number arguments:

```
define Zprod { ( $1.x*$2.x - $1.y*$2.y, $1.y*$2.x + $1.x*$2.y ) }
```

## 3.4 M4 macros

M4 is a simple but powerful macro language originally distributed with Unix systems [2], but free versions are available for other operating systems. The use of this language requires an extra processing step, but the power and flexibility of the macros easily make up for it. The macro definitions are read before the text to be processed, typically by a system command such as

```
m4 configurationfile.m4 diagram.m4 | dpic -g > diagram.tex
```

An m4 macro is defined as follows:

```
define('name','contents')
```

so that distinct occurrences of *name* will be replaced by *contents* in the following text. This basic description is a vast simplification of the power that results from conditional substitution, recursion, file inclusion, integer arithmetic, shell commands, and multiple input and output streams. The online manual [5] is a good source of details.

A general rule might be that floating point computation is in the domain of dpic macros but text substitution is often better done in m4 macros.

When m4 reads text, it strips off pairs of single quotes: thus, `'text'` becomes `text`. If `text` is read again, as when it is a macro argument, for example, then the process is repeated. The single quotes serve to delay the evaluation of macros within `text`, as in macro definitions described above. Therefore, to avoid m4 changing dpic macro definitions or L<sup>A</sup>T<sub>E</sub>X, enclose them in single-quote pairs.

Some simple applications of m4 macros are illustrated in the subsections that follow.

### 3.4.1 M4 branching

As an illustration of m4 macros, suppose that commands that are specific to the postprocessor must be generated. Then the macro

```
ifpgf('pgf-specific commands','other commands')
```

for example, should expand to its first argument if pgf is to be the postprocessor, otherwise it

should expand to the second argument. To implement this, `ifpgf` is defined in the statement

```
define('ifpgf','ifelse(m4postprocessor,pgf,'$1','$2'))
```

which tests for equality of the character sequences `m4postprocessor` and `pgf`. However, if `m4postprocessor` is a macro, it is replaced by the macro text before the test is performed, and if the macro text is `pgf`, then the first argument of `ifpgf` is evaluated. In the `Circuit_macros` package, `m4` is required to read a postprocessor-specific file before anything else, and that file contains the required definition of `m4postprocessor`.

The built-in macro `ifelse` can have multiple branches, as illustrated below:

```
ifelse(m4postprocessor,pstricks,'PSTricks code',
      m4postprocessor,pgf,'TikZ PGF code',
      m4postprocessor,mfpic,'Mfpic code',
      m4postprocessor,mpost,'MetaPost code',
      m4postprocessor,xfig,'Xfig code',
      m4postprocessor,postscript,'Postscript code',
      m4postprocessor,pdf,'PDF code',
      m4postprocessor,svg,'SVG code',
      'default code')
```

### 3.4.2 Extending pic: perpendiculars

The `Circuit_macros` `vperp` macro illustrates how `m4` macros can extend the `pic` language. The purpose is to generate a pair of values representing the unit vector perpendicular to a given line, say.

```
define('vperp',
  'define('m4pdx','('$1'.end.x-'$1'.start.x'))dn1
  define('m4pdy','('$1'.end.y-'$1'.start.y'))dn1
  -m4pdy/vlength(m4pdx,m4pdy),m4pdx/vlength(m4pdx,m4pdy)')
```

The macro can be invoked as `vperp(A)` where `A` is the name of a line. Another invocation might be `vperp(last line)`. First, two macros (beginning with `m4` to avoid name clashes) are defined as the  $x$ -distance  $dx$  and  $y$ -distance  $dy$  of the end of the line from the start. The macro evaluates to the pair  $-dy/\sqrt{(dx)^2 + (dy)^2}$ ,  $dx/\sqrt{(dx)^2 + (dy)^2}$ , where the denominators are calculated by the macro `vlength`.

### 3.4.3 Setting directions

The `pic` language defines the concept of the current drawing direction, which is limited to `up`, `down`, `left`, and `right`. Two-terminal circuit elements, for example, might have to be drawn in any direction, which calls for the ability to define diagrams without knowing their final orientation and to rotate the result at will. This capability can be added to the basic `pic` language by judicious use of macros.

First, instead of specifying positions in the usual way, such as in

```
line from (x1,y1) to (x2,y2)
```

for example, let us agree to write

```
line from vec_(x1,y1) to vec_(x2,y2)
```

where `vec_(x1,y1)` evaluates to

```
(a*x1 + b*y1, c*x1 + d*y1)
```

Then if `a` and `d` are `cos(theta)`, `b` is `-sin(theta)`, and `c` is `sin(theta)`, this transformation corresponds to rotating the argument vector by an angle `theta`. To produce relative coordinates, the macro `rvec_(x,y)` evaluates to

```
Here + vec_(x,y),
```

so writing

```
line to rvec_(x1,y1)
```

draws a line from the current position `Here` to a point `(x1,y1)` defined with respect to rotated coordinates relative to `Here`.

The `Circuit_macros` package makes extensive use of versions of the above two macros. The angle and transformation constants are set using macros

```
Point_(degrees), point_(radians), or setdir_([U|D|L|R|degrees],default)
```

which have angles as arguments.

This usage is illustrated by the macro `lbox`, which draws a pic-like box oriented in the current direction. It can be defined as

```
define('lbox',
  'define('m4bwd',ifelse('$1',,boxwid,('$1'))dnl
  define('m4bht',ifelse('$2',,boxht,('$2'))dnl
  line from rvec_(m4bwd,0) to rvec_(m4bwd,m4bht/2) \
    then to rvec_(0,m4bht/2) \
    then to rvec_(0,-m4bht/2) \
    then to rvec_(m4bwd,-m4bht/2) \
    then to rvec_(m4bwd,0) '$3'')
```

The macro is invoked as `lbox(width, height, type)`; for example,

```
Point_(20); lbox(, ,fill 0.9)
```

draws a light gray-filled box of default size at an angle of 20 degrees from the horizontal. In the macro, the width and height of the box are first defined, using default values if the first and second arguments are not given. Then a line is drawn to outline the box, and the `fill 0.9` argument is appended to the line command to fill the box. A slightly more elaborate version that encloses the box in `[, ]` brackets is given by the `Circuit_macros` `rotbox` macro.

### 3.5 Subscripts

Dpic allows variables and labels to have a single or double subscript; thus `x`, `x[4]`, and `x[4,2]` are distinct variable names, and can be employed in expressions as usual. Similarly, `P` and `P[3]` are distinct labels. Writing the subscript as `[position]` for any defined position is the same as `[position.x, position.y]`.

Subscripted variables can simplify the storage and manipulation of sequential data, and subscripted labels aid in drawing sequences of objects.

### 3.5.1 Assigning an array of numbers

We can assign an array of numbers to subscripted variables using statements such as

```
x[1] = 47
x[2] = 63
:
```

and so on, but generating the subscripts is inconvenient, particularly when these statements are obtained by editing a data file. One way of entering the data is to employ the m4 macro definition

```
define('inx','define('m4x',ifdef('m4x','incr(m4x)',1))m4x')
```

Then, writing

```
x[inx] = 47
x[inx] = 63
:
```

and processing with m4 automatically generates the required subscripts. The macro sets `m4x` to 1 if it is not yet defined, otherwise it increments `m4x`, and then it evaluates to `m4x`. On completion of the assignments, `m4x` has the value of the last subscript.

Another way of assigning variables to a subscripted variable is by the definition

```
define array {
  for i=2 to $+ do { exec sprintf("$1[%g] = %g",i-1,i); } }
```

which equates the subscripted first argument to the values in argument 2, 3, ... so that, for example,

```
array(x,9,-4,7,4.2,0,0,7.9,0,0,10,11,12,13,14,10)
```

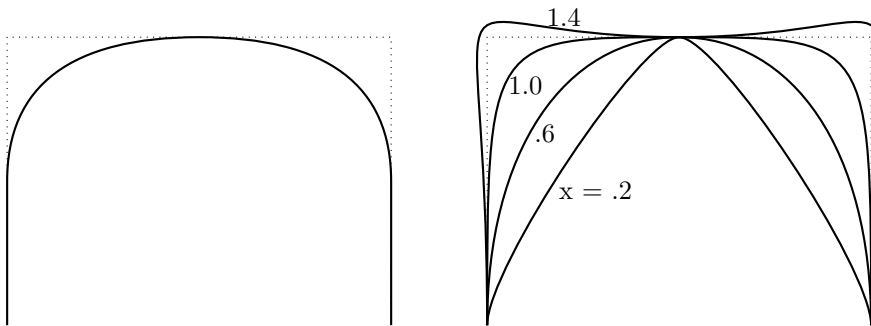
assigns the second to sixteenth arguments to `x[1]` to `x[15]` respectively.

Dpic does not define vector operations, but it is easy to write macros for them. For example, macros can be used to define 3-dimensional vectors, transform them, and to project them onto a drawing plane.

## 3.6 Splines

Dpic implements standard pic splines by default, as on the left in the following figure, which shows the result of the command

```
spline up 1.5 then right 2 then down 1.5
```



A straight line is drawn along the first half of the first segment and the last half of the last segment. The curve is tangent to the centres of the segments. The dpic result of including an expression after `spline`, as in

```
spline x up 1.5 then right 2 then down 1.5
```

is shown on the right of the figure, as the tension parameter `x` varies from 0.2 to 1.4. The curve begins at the start of the first segment and terminates at the end of the last segment. The tension parameter can be varied to assist in fitting a multisegment spline to a curve. It turns out, for example, that the optimum tension for approximating a circle using four splines is the value 0.551784.

### 3.6.1 Curve fitting

Splines are drawn with respect to control points, but only pass through the first and last point. Suppose that a sequence of points  $X[0], X[1], \dots, X[n]$  has been given, and a spline is to be found to pass through each point. The control points  $P[0], P[1], \dots, P[n]$  have to be calculated. These points satisfy the following equations:

$$P[0] = X[0]$$

$$P[i-1]/8 + P[i]*3/4 + P[i+1]/8 = X[i] \text{ for } i = 1 \text{ to } n-1$$

$$P[n] = X[n]$$

The Circuit\_macros `fitcurve` macro performs the required calculations and draws the spline to pass through the given points.

## 3.7 Text

Pic interpreters are not wordprocessors, but generate output compatible with L<sup>A</sup>T<sub>E</sub>X in the case of dpic and groff or L<sup>A</sup>T<sub>E</sub>X in the case of gpic. Dpic also generates output formats in which font choice and formatting, if any, must be done according to the context, as for example, postscript output containing psfrag strings, or SVG output formatted by macros or by hand. The result is that the size and position of text may need to be adjusted by hand for the particular workflow and postprocessor to be employed.

The dpic `textht` environment variable (see [Section 7](#)) often gives dpic a good estimate of the actual height of embedded text, but the width of the text is more difficult to estimate. Consequently, text is sometimes truncated by the figure bounding box at the left or right



edge of the figure. Setting the width of strategic strings by hand, e.g., `"text" wid 0.65` can serve as a quick cure in specific cases, but is better done automatically as described in the `Circuit_macros` documentation [1]. Briefly summarized, L<sup>A</sup>T<sub>E</sub>X can write the exact height, width, and depth of formatted text (or anything that creates a box) to a file. The file is read by dpic during a second processing of the diagram, so that the width  $x$  and height  $y$  are known numerical values in the phrase

```
"text" wid  $x$  ht  $y$ 
```

Text in PDF and SVG output is mentioned further in Section 4.1.

Justified text deserves special mention. The second line in

```
A: circle rad 0.01 at (0,0)
S: "Hello" at A
   box wid S.wid ht S.ht at S
```

produces a greeting centered at A as expected. The bounding box of the text is also centered at A but has default height and width `textht` and `textwid` (that is, zero in most cases) respectively. The revised version

```
S: "Hello" wid 24.7/72 ht 7.6/72 at A
```

(where the formatted size in points has been obtained automatically or by measurement) produces a correct bounding box. There are minor differences in the way gp<sub>ic</sub> and dp<sub>ic</sub> handle justification, as discussed in Section 5.19.

### 3.8 Postprocessor commands and color

Arbitrary postprocessor commands can be interspersed with pic statements to achieve effects such as gradient fills, clipping, and transformations. There are two ways of passing commands into dp<sub>ic</sub> output.

Dpic lines beginning with the backslash `\` are passed through to the output without modification. This method works well for L<sup>A</sup>T<sub>E</sub>X statements and commands to postprocessors that rely on TeX macro processing.

The second method is the form

```
command "text"
```

or

```
command sprintf("format",expression,...)
```

both of which pass the contents of the string into the output stream without the enclosing quotes. The string need not start with a backslash.

Both of the above methods add considerable power to the pic language, but there are two issues. The first is that if a postprocessor transformation changes the size of a drawn element, the pic processor will not know the new size parameters unless they are explicitly calculated. The second is the challenge of designing a single macro that produces appropriate postprocessor code to have identical effect with different postprocessors.

The `Circuit_macros` library contains several routines that produce equivalent or nearly equivalent results for several postprocessors. It is probably a good rule to stick with one or two postprocessors such as Tikz-pgf or PSTricks.

Dpic defines a number of internal variables for controlling actions that depend on output

format. Internal variables `optTeX`, `opttTeX`, `optpict2e`, `optPSTricks`, `optPDF`, `optPGF`, `optMFPic`, `optPS`, `optPSfrag`, `optMpost`, `optSVG`, and `optxfig` have numerical values 1 to 12 respectively, and variable `dpicopt` is given the value corresponding to the output format specified on the command line. Therefore, conditional drawing with a `case` statement as described on [page 9](#), or with `if` as in the following can be performed:

```
if (dpicopt==optPS) || (dpicopt==optPDF) then {
  drawing commands for postscript or pdf }\
else { drawing commands for latex-related formats }
```

The variable `optsafe` is set to true if option `-z` has been specified or if `dpic` has been compiled to allow only safe mode.

### 3.8.1 Color

From version 18, `gpic` allows coloured lines and filled objects as follows, and `dpic` allows them where the postprocessor supports them:

```
object outlined string
object shaded string
object colored string
```

where *string* specifies a colour compatible with the postprocessor. For planar objects, the third case is equivalent to

```
object outlined string shaded string
```

For the linear objects `line`, `arrow`, `spline`, `dpic` treats

```
colored string
```

to be the same as

```
outlined string
```

but fill can be added by explicitly writing

```
outlined string shaded string
```

The original `pic` language did not include the `outlined` or `shaded` attributes. Current processors recognize these but know nothing about color except as strings attached to drawing elements. What the string should contain depends on the postprocessor.

### 3.8.2 Filling with color

Basic `pic` shapes such as boxes, circles, and ellipses can be colored and filled using, for example,

```
ellipse shaded "color" outlined "color"
```

and, if the two colors are the same, this can be abbreviated as

```
ellipse colored "color"
```

where the color strings are compatible with the postprocessor. The `m4` macro

```
rgbstring(red fraction,green fraction,blue fraction)
```

evaluates to a string compatible with the postprocessor (for postprocessors that allow it; that is, `Tikz PGF`, `PSTricks`, `SVG`, `MetaPost`, `PDF`, and raw `Postscript`) so that the following produces a circle filled in gold, for example:

```
circle shaded rgbstring(1,0.84,0).
```

More elaborate options can also be invoked. For example, with Tikz PGF output, the command

```
box shaded "orange, opacity=0.5"
```

sets the opacity of the fill, and with PSTricks output, the sequence

```
box shaded "lightgray,fillstyle=hlines*,linecolor=blue,
hatchwidth=0.5pt,hatchsep=5pt,hatchcolor=red,hatchangle=45"
```

produces a hatched multicoloured fill and is equivalent to

```
command "\pscustom[fillcolor=lightgray,fillstyle=hlines*,linecolor=blue,"
command "hatchwidth=0.5pt,hatchsep=5pt,hatchcolor=red,hatchangle=45]{%"
box
command "%"
```

One limitation of the pic language is that it lacks the concept of a path composed of different basic curves such as lines, splines, and arcs. Dpic, however, extends the `shaded` and `outlined` directives to linear objects such as lines or splines where the output postprocessor allows. Consider, for example, the following macro:

```
define slantbox { [ # wid, ht, xslant, yslant, attributes
w = $1 ; h = $2 ; xs = $3 ; ys = $4
NE: (w+xs,h+ys)/2 ; SE: (w-xs,-h+ys)/2
SW: (-w-xs,-h-ys)/2 ; NW: (-w+xs,h-ys)/2
N: 0.5 between NW and NE ; E: 0.5 between NE and SE
S: 0.5 between SE and SW ; W: 0.5 between SW and NW
C: 0.5 between SW and NE
line $5 from N to NE then to SE then to SW then to NW then to N
] }
```

This macro implements a version of the `xslanted` and `yslanted` attributes recently introduced for gpics boxes, for example

```
box wid 0.1 ht 0.5 xslanted 0 yslanted 0.1 \
shaded "Dandelion" outlined "black"
```

The `slantbox` macro defines the implied compass corners N, S, NE, .... Its fifth argument can be used to fill or otherwise specify the line. For example, the command

```
slantbox(0.1,0.5,0,0.1,shaded "Dandelion" outlined "black")
```

is equivalent to the gpics example. The above macro can be modified easily to produce arbitrary polygons, for example. The color `Dandelion` is automatically defined for PSTricks by the L<sup>A</sup>T<sub>E</sub>X line

```
\usepackage[dvipsnames]{pstricks}
```

For TikZ-PGF, try

```
\usepackage[usenames,dvipsnames]{xcolor}
\usepackage{tikz}
```

### 3.8.3 External files

Dpic can send print output to a file (operating system permitting) using the command

```
print arg > "filename"
```

which creates the named file, or

```
print arg >> "filename"
```

which appends output to the named file if it exists and creates the file if not. If the `-z` option has been invoked or `dpic` was compiled in safe mode, both of these give warning messages rather than writing to the file.

External files can be used to implement forward referencing. For example, sometimes the final size of a diagram component is required in order to draw a background object that will be overlaid by the component. One solution is to process the diagram twice. In the following code fragment, an assignment to `x` is written to an external file so that the required value of `x` will be known after the file is read on the second pass.

```
x = default value
print "" > "datafile"                                # Make sure the file exists
copy "datafile"                                       # On second pass, read x = required value
... draw objects and calculate required value y ...
print sprintf("x = %g",y) > "datafile"                # Write out the assignment
```

## 4 SVG, PDF, and Postscript output

The `-r` (raw Postscript), `-v` (SVG), `-d` (PDF), and `-x` (xfig) options of `dpic` produce output that is not intended to be processed directly by  $\text{\LaTeX}$  or, in some cases, printed on paper.

As a “little language,” Pic and its interpreters leave text sizing and formatting to word processors, which are flavours of  $\text{\LaTeX}$  in the case of `dpic`, and `groff` or  $\text{\LaTeX}$  in the case of `gpics`. Text formatting is specified by the text content itself or by the string contents of `command` lines. The exception to this is that text is also a graphical element; its size and location is sometimes required, and the interpreter must also be able to stack multiple text strings vertically.

In most cases, placing text using

```
"Text" wid x ht y at location
```

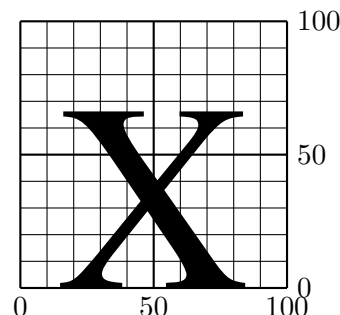
is sufficient.  $\text{\LaTeX}$  can write the height and width of the formatted text (which can be anything that produces a TeX box) to an external file, which is passed to the `pic` processor for use on a second pass over a diagram, as described in detail in reference [1], so the values of `x` and `y` can be obtained automatically and macros can generate the above line without intervention by hand.

Other output formats, however, require different treatment. This is specially true of SVG output for the web.

### 4.1 SVG text

In contrast with the various `dpic`  $\text{\LaTeX}$  output formats for which text size is established by postprocessor commands, the size of SVG text has to be set by the `pic` interpreter (at least until an automatic way of returning formatted sizes from renderer back to `pic` interpreter is established).

One issue is that the formatted height of text is generally a fraction of the nominal size. For example, the Times letter X at 100 bp placed on a 100 bp by 100 bp grid is shown at the right. The height of the X is approximately 66 bp or 0.66 of the nominal height. This ratio depends on the font. Consequently, a language issue arises: if the attribute `ht y` is to be used to set text size, should the value of  $y$  be the nominal height (the point size) or the formatted height of the text? Dpic treats  $y$  as the true graphical height in drawing units. Thus for example, `"text" ht 10*(scale/72)` produces output 10bp high.



To assist with the above and similar issues when the `-v` option is invoked, dpic performs the equivalent of

```
dpPPI = 96
dptextratio = 0.66
```

in the outermost scope of the diagram. These are ordinary variables that can be referenced or changed as usual. The variable `dpPPI` is given the initial value 96 which is pixels per inch on a 20-inch wide computer monitor of 1920 pixels width. On such a monitor, dpic output should be close to true size. This parameter can be changed using the assignment

```
dpPPI := value
```

for different screen resolution, and `dptextratio` can be similarly changed. (For xfig output, the following are defined under the `-x` option: `xfigdispres = 80`; `xfigres = 1200`.)

When dpic sees the `ht y` attribute attached to SVG text, it sets the nominal text size to  $y/\text{dptextratio}$  expressed in points. Otherwise, dpic uses the current value of the environment variable `textht`, which has default value  $11/72 \times 0.66$ , to set the default size to 11 bp.

The Circuit\_macros configuration file `svg.m4` contains a macro `svg_font` that can simplify font specification, and a macro `svgfontratio` that can be used to set `dptextratio` and `textht` automatically for common HTML fonts.

Dpic sets the width of text to the current value of `textwid` by default but, as always, it is best not to rely on default behaviour. Specify the object (text in this case) completely using `wid expression ht expression`. In the absence of an automatic means of determining it, text width can be measured approximately when displayed full-size on a monitor. Fortunately on many diagrams, knowing the width is unnecessary or is required for only a few key strings. In principle, this process might be automated using suitable macros and scripting (JavaScript, probably).

## 4.2 Bounding boxes

The bounding box of a diagram is not always known exactly or even defined exactly, since it can depend on the context in which the diagram is to be used. Within a diagram, different line widths, mitred joints, splines, colored output, overpainting, arbitrary text, arbitrary Postscript or SVG, and other complications are allowed; consequently dpic can only provide an estimate of the exact bounding box.

Strategic `move` commands can be used to enlarge the bounding box as illustrated at the end of the discussion below.

Dpic uses the constant-width Courier font in pdf files, thus allowing the width of strings to be calculated from their height, which is specified by `textht` with default 11 pt, or by the “`height expression`” attribute. The string width is calculated from the height and character count, but can be set explicitly by using `width expression` as the rightmost string attribute.

*Psfrag output* is another special case. If `textht` is set to a nonzero value, then its value relative to 11 bp is given as a scale factor to the psfrag `\tex` command. An alternative way of changing the diagram text height is to set it in the main document; for example, `{\small \includegraphics{file.eps}}`.

*Postscript bounding boxes:* For a while, the dpic `%%BoundingBox` output line simply gave the nominal bounding box determined by line ends and other control points. The use of dpic in server mode has induced a change that correctly defines the bounding box for very basic diagrams. More explicitly, consider

```
.PS
box with .sw at 0,0
.PE
```

which draws a box with southwest corner line centres intersecting at Postscript coordinates 0,0 and northeast intersection at 54,36. Dpic `-r` augments this nominal bounding box by half of the last `linethick` value (default `linethick` is 0.8 bp) in the outermost diagram scope to produce the Postscript bounding-box definitions

```
%%BoundingBox: -1 -1 55 37
%%HiResBoundingBox: -0.4 -0.4 54.4 36.4
```

The `%%BoundingBox` line contains integer values that enclose the high-resolution coordinates.

*PDF bounding boxes:* PDF includes a `MediaBox` element which serves approximately the same purpose as the postscript `BoundingBox`, and is calculated the same way. By default, dpic puts the lower left corner of PDF output at coordinates (0,0).

*Manual bounding box adjustment:* It may be necessary to adjust the bounding box manually. To zero the automatic adjustment, put `linethick=0` at the end of the outermost scope. Then arbitrary margins can be added to the nominal box as shown below, where 2, 1, 1, and 0 points are added to the left, bottom, right, and top margins respectively:

```
.PS
Diagram: [
    drawing commands
]
linethick = 0
move from Diagram.sw-(2,1)/72*scale to Diagram.ne+(1,0)/72*scale
.PE
```

## 5 Pic processor differences

Differences among processors, and between dpic and gpik particularly, are summarized below. These differences result from incomplete language definition and from different implementation contexts. Normally, the only changes required to process correct pic or gpik input with dpic are changes to `{...}` instead of `X...X` syntax as explained below, together with text formatting if the original code was written for groff. Sometimes, differences in default behavior (such as for arcs or object placement) must be considered. The remaining items are reasonably small syntactical differences or relate to the use of L<sup>A</sup>T<sub>E</sub>X or the other dpic output formats.

### 5.1 Command-line options

They are completely different, of course. Type `dpic -h` to see a list of dpic options.

### 5.2 Output formats

Gpic `-t` output consists of a sequence of `\special` statements contained in the TeX box `\box\graph`. The `\special` statements are automatically copied into the `.dvi` file for interpretation by a printer driver such as dvips.

Dpic does not generate tpic specials. See the option list in [Section 2](#) for output formats.

### 5.3 “.” and “\” lines

Gpic passes lines beginning with `.` through to the output, thereby allowing arbitrary Troff macros to be interspersed with pic drawing commands. Some programs that generate pic output automatically insert the Troff line

```
.ps 11
```

on the assumption that the text point size should be 11. Dpic ignores non-continuation lines beginning with `.` within pictures. Some programs (e.g., pstoeedit) add Troff comment lines beginning with `.\` outside the `.PS`, `.PE` delimiters. These lines must be dealt with separately.

Both gpik and dpic pass lines beginning with `\` to the output but dpic does not automatically append a `%` at the end as gpik does.

### 5.4 for-loop and if bodies

In gpik, a for loop has the form

```
for variable = expr1 to expr2 [by [*]expr3] do X body X
```

where `X` is any character not occurring in `body`, but `{ body }` is also allowed. In dpic only the latter is allowed. Similarly, the required form of an if statement for dpic is

```
if expr then { if-true } [else { if-false }]
```

The test for termination of the multiplicative form of the for loop is not identical for dpic and gpic but the effects are identical for positive parameters.

Both gpic and dpic allow the loop index variable to be changed within the loop, so infinite repetition or control of termination by a test are possible.

## 5.5 Closing braces and nul elements

In some cases where the contents of {} are scanned, a semicolon or newline must be inserted prior to the closing brace to ensure that the contents of the braces are completed before exit. Such a case occurred in the definition of the `case` macro on [page 9](#). Similarly, a semicolon or newline is required in the following `if` test with an `else` clause:

```
if (sh "test -f fileA.pic")==0 then { copy "fileA.pic"; }\
else { copy "fileB.pic" }
```

The semicolon ensures that the `copy` command completes before exiting the first pair of braces.

In the following lines, gpic ignores the {} braces enclosing the trivial for-loop in the first line and the if-test in the second line. Consequently, the "x" labels are misplaced. Dpic places the labels correctly. The cure for gpic is to put a semicolon or newline between the rightmost braces.

```
{for i=1 to 1 do {line right}}; "x"
move; i=1; {if i==1 {line right}}; "x"
```

Gpic gives error messages if there is nothing or nothing more than newlines or semicolons between "{ }" braces or "[ ]" brackets. These constructs do not occur often in hand-generated code (although [] is sometimes convenient) but m4 or other macros containing conditional expansion can generate them unless care is taken to avoid them. Dpic allows both items, with [ ] generating a block of zero size.

## 5.6 End of line

The line end is significant in the pic grammar. However, dpic ignores line ends following `then`, `{`, `:`, `else`, or end of line. Both the CR (octal 015) and NL (octal 012) characters are treated as line ends.

If the last character of a line is \ (but not within a string), then reading is continued on the next line. Dpic allows this to occur within keywords or constants. Strings can contain multiple lines.

The # character begins a comment which ends at the end of the line.

The construction

```
if condition then { if-true }
else { if-false }
```

produces an error with all pic interpreters. To avoid this error, write

```
if condition then { if-true } \
else { if-false }
```



## 5.7 Logic

Dpic and gpic give slightly different default precedences to the logical operators `!`, `&&`, `||`, `==`, `!=`, `>=`, `<=`, `<`, and `>`, so judicious use of parentheses may sometimes be in order. Dpic also requires comparisons and logical operations to be put in parentheses in numerical expressions; e.g.,

```
x = ("text1" == "text2")
```

The construct `x<A,B>` is intended to have the same meaning as `(x between A and B)` but, in some obscure circumstances, all pic interpreters have difficulty determining whether the `<` character is part of such an expression or is a logical comparison operator. Dpic treats `<` as a comparison in the expression following `if` so the form `(x between A and B)` should be used in such expressions, e.g.

```
if (0.5 between A and B).y < 2 then { ... }
```

## 5.8 Defaults for line, spline, arrow, move

Dpic treats the defaults for linear objects consistently with planar objects with respect to the `at` modifier. Gpic treats them differently:

In dpic, `line at Here` means `line with .center at Here`.

In gpic, `line at Here` means `line with .start at Here`.

In dpic, the location corresponding to `last line` is `last line.c`.

In gpic, the location corresponding to `last line` is `last line.start`.

With dpic, `last line.wid` returns the arrowhead width of the line or the default arrowhead width if the line has not been given an arrowhead; similarly for `.ht` and for the other linear objects including `move` and, of course, `arrow`.

The compass corners of multisegmented linear objects are not defined in descriptions of the pic grammar; reasonable positions are returned but they should be used with care.

## 5.9 Arc defaults

Gpic and dpic have different algorithms for picking a default radius. The best practice is to specify arcs completely. There is also ambiguity in the pic language. The statement

```
arc cw rad x from A to B
```

does not define a unique arc. There are two arcs that satisfy this specification, with centres on opposite sides of the line joining A and B. Dpic will choose one by attempting to minimize the angle between the current direction and the initial tangent of the arc. Instead, use

```
arc cw from A to B with .c at C
```

## 5.10 Strings

Strings are arbitrary character sequences between double quotes. Equivalently, a string is produced by the C-like `sprintf` function

```
sprintf("format" [, expression, ... ])
```

The C `snprintf` function is used for implementing this; therefore, the precision of default formats such as `%g` may vary by machine and compiler. To produce transportable results, specify the precision completely, e.g., `%8.5f`. As in C, the pair `%%` in the format string prints the percent character. Only the `f`, `e`, `g` formatting parameters are valid, since expressions are stored as floating-point numbers, e.g.

```
line sprintf("%g%g0", 2, 0 ) above
```

is equivalent to

```
line "200" above
```

Similarly,

```
command sprintf("\pscircle(%g,%g){%g}",0,0,0.5)
```

places a line containing the formatted string without the quotes into the output. The numerical `sprintf` arguments can be arbitrary expressions in place of the constants shown.

Dpic allows strings to be concatenated by the `+` operator; thus,

```
"abc" + sprintf(" def%g",2)
```

is equivalent to `"abc def2"`.

In a macro, a dollar sign followed by an integer in a string will expand to the corresponding macro argument if it is defined. Separate the dollar sign from the integer to avoid expansion, as in the TeX strings `"{\$}1"`, `"\$ 1\$"`, or `"${0}$"`, for example.

Both dpic and gpik treat `\` as an escape character prior to the double-quote character in a string, so `"\"` is a length-one string containing the double quote. Otherwise, the backslash is an ordinary string character. For example, the characters `\"u` in a string are output as `\u` to produce `ü` when processed by L<sup>A</sup>T<sub>E</sub>X. A string with a backslash as last character has to be generated using a macro; for example, if we write `define charstr {"$1"}` then `charstr(abc\)` evaluates to a string containing the required final backslash.

Gpic implements the attribute `aligned` of an object to rotate attached strings by sending commands to postprocessor `grops` or `gropdf`, which translate troff output to postscript or pdf respectively. The bounding box of the rotated string is unknown. Dpic does not define this attribute but allows the user to rotate text directly when permitted by the postprocessor. For example, the Circuit\_macros macro `rs_box` rotates text for PSTricks or TikZ-PGF output and calculates the bounding box. For SVG output, `svg_rot()` has a similar function. The following simple pic macro accomplishes the same goal for PSTricks and TikZ PGF but the resulting text bounding box is not calculated.

```
#                                dprtext(degrees,text)
define dprtext {[
  if dpicopt==optPSTricks then {
    sprintf("\rput[c]{%g}(0,0)", $1)+"{$2}" } else {
  if dpicopt==optPGF then {
    sprintf("\pgftext[rotate=%g]", $1)+"{$2}" }}
  ]}
```

Then, for example, `dprtext(45>Hello)` at *position* places the text “Hello” rotated 45° at the specified position.

A string cannot be assigned to a variable name; the nearest equivalent is to define a macro containing the string as the body.

Both `dpic` and `gpik` allow logical comparison of strings. Put the comparison in parentheses if it is to be used in an expression.

String height and width are unscaled on final output from `dpic` since these may depend on later formatting by L<sup>A</sup>T<sub>E</sub>X.

`Gpic` does not allow newlines in strings; `dpic` does, with an upper limit of 4095 characters.

### 5.11 `print arg, ...`

`Dpic` allows only one argument, which may be an expression, position, or string. To print several quantities at once, use

```
print sprintf(...)
```

to generate a string and, if the string is complicated, remember that *string1* + *string2* + ... evaluates to one string.

### 5.12 `command arg, ...`

Arbitrary commands are sent to the standard output stream. `Dpic` allows only one argument, which is a string or `sprintf(...)`.

### 5.13 Operating system commands

With `dpic`, the required form for a shell (operating system) command is

```
sh "shell command"
```

or

```
sh sprintf("format", expression,...).
```

The value returned by the operating system can be captured by putting the command in parentheses; for example,

```
if (sh "shell command") == 0 then {...}
```

### 5.14 `copy`

`Dpic` supports the command

```
copy "filename"
```

but does not directly support the `gpik` commands

```
copy [filename] thru X body X [ until word ]
```

```
copy [filename] thru macro [ until word ]
```

These functions and many more are readily implementable with dpic in any Unix-like environment. For example, a basic implementation of `copy filename thru macro` is given by the following macro:

```
#                               copythru(macro_name,"filename")
#                               Implements copy filename thru macro_name
#                               for data separated by commas, spaces, or tabs
define copythru { sh "sed -e 's/^[ \t]*$1(/' -e 's/[ \t]*$)/' \
    -e 's/[ , \t][ \t]*/,/g' $2 > copy_tmp_"
copy "copy_tmp_"
sh "rm -f copy_tmp_" }
```

The lines of the *filename* file are changed to calls of the `macro_name` macro and written into a temporary file, which is then read by dpic. Such usage is not as simple as a built-in function but allows greater flexibility of data because the `copythru` macro can be replaced by one customized to suit. For example, the `sh` command might invoke another program to generate the data.

## 5.15 plot

The `plot` command is deprecated in gpic and not allowed in dpic.

## 5.16 fill

In gpic, a fill value of 0 means white, 1 means black. Dpic uses 0 as black and 1 as white as do Postscript and the original ATT pic.

The pic language specifies fill only for box, ellipse, and circle. With dpic, the `shaded` directive fills a path defined by a linear object, where allowed by the postprocessor. The concept of a path containing several arbitrary linear objects does not exist in the pic language but can be implemented using postprocessor commands inserted into `command string` statements. Arbitrary paths can also be constructed using a single spline and the `continue` statement.

## 5.17 Scaling

Dpic implements a `scaled` attribute, so that

```
box scaled 1.2
```

produces a box with dimensions scaled by 1.2, and

```
[box; line scaled 3; circle] scaled 0.5
```




scales all objects within the block by 0.5. The latter can be used when different parts of a diagram require different scaling. For example, if file `component.pic` contains a component scaled by 25.4, then the following allow it to be used:

```
[ copy "component.pic" ] scaled 1/25.4
```

As always, line thicknesses are not scaled.

## 5.18 Arrowheads

Pic processors provide a limited variety of built-in arrowhead shapes. Dpic draws arrowheads according to the environment variable `arrowhead` as shown below.

```
arrowhead = 0   
arrowhead = 3   
default 
```

Any other value of `arrowhead` produces the default filled head shown but also results in an a head shape native to the postprocessor in some cases. The default value of `arrowhead` is 1 in conformance with other pic processors. Postprocessor parameters can be changed using lines of the form

```
command "postprocessor commands"
```

Changing the line thickness does not affect arrowhead size parameters, which have to be changed explicitly by either of the following methods. The line thickness is specified in points but the arrowhead size parameters are in drawing units:

```
# Method 1 (global change within the current block):
```

```
linethick = 2    # default 0.8 (bp)  
arrowht = 0.18   # default 0.1 (in)  
arrowwid = 0.09  # default 0.05 (in)  
arrowhead = 0    # default 1  
arrow
```

```
# Method 2 (put a type 0 arrowhead on the current object):
```

```
arrow -> 0 thick 2 ht 0.18 wid 0.09
```

There is a subtle language problem concerning arrowheads. Let us agree that the following examples should all produce an arrow of length 1 inch and arrowhead width 1 millimetre:

```
.PS  
  arrow right 1 width 1/25.4  
.PE  
.PS  
  scale = 25.4  
  arrow right 25.4 width 1  
.PE  
.PS 1  
  arrow right 1/4 width 1/25.4  
.PE
```

The original (ATT) pic fails on the second example, insisting that arrowhead dimensions be given in inches. Gpic fails on the third by scaling the arrowhead on final output. Although it might be argued that this is a feature, it causes serious awkwardness when diagrams are to be scaled to exact final dimensions using the `.PS x` construction, since the effective scale factor is unknown until the `.PE` line is processed. Dpic generates the same arrow in all three cases, treating arrowhead parameters like line thickness (unscaled) parameters on final output.

## 5.19 Compass corners and justified text

Dpic consistently requires a dot before compass corners, so the gpic line

```
x at center of last box
```

should be written for dpic as

```
x at .center of last box
```

Justification is performed by the postprocessor, and combining justified text with drawing elements requires special consideration. The lines

```
S: "Hello" wid 24.7/72 ht 7.6/72 at A ljust
```

```
box wid S.wid ht S.ht at S
```

produce a box of the correct size but centered at A because the location (A) and a justification command are given to the postprocessor. This behavior could be easily changed but would affect much legacy code and has been kept for dpic, which will produce a justified box using

```
box wid S.wid ht S.ht with .w at S.w
```

Gpic and dpic place justified text slightly differently. In the previous example, gpic places the text with its left edge exactly at A, with similar effect for right justification. Dpic uses the environment variable `textoffset`, which is 2pt by default, to place the left edge of the text at `A + (textoffset,0)` with a similar gap for other relative positioning. Assigning `textoffset` to 0 sets the gap added by dpic to zero.

## 5.20 continue

In dpic, the `continue` command appends a linear drawn object to the previous drawn object as if `then` had been used in the original command, but calculations can be performed to determine size or placement of the appended part as in, for example,

```
line up right
```

```
  calculations
```

```
continue down
```

```
  more calculations
```

```
continue up left
```

The keyword `continue` can also be used slightly differently. The line drawn by

```
move to (0,0); line right 1 then to (Here,(2,1))
```

terminates at (0,1) since `Here` is the position (0,0), whereas the following terminates at (1,1) since `Here` is (1,0):

```
move to (0,0); line right 1; continue to (Here,(2,1))
```

## 5.21 Subscripted variables and labels

Dpic allows subscripted variables and capitalized labels, as described in [Section 3.5](#).

## 5.22 Splines

Gpic extends the ATT pic grammar to make `line 0.5` legal and mean “a line of length 0.5 in the current direction.” All linear objects are treated similarly. Dpic does the same except for splines. In the statement

```
spline x from A to B then to C ...
```

the parameter *x* is a tension parameter, normally between 0 and 1, to control the spline curvature. If *x* is not present as in the normal pic grammar, the curve starts with a straight line halfway along the first segment and ends with a straight line along the second half of the last segment. When *x* is present however, dpic draws the spline from the start of the first segment to the end of the last segment.

## 5.23 Arithmetic

In an expression, gpic allows terms of the form `x*-y`, whereas dpic produces an error message. Use `x*(-y)` or the equivalent.

Dpic allows the assignment operators `+=`, `-=`, `*=`, `/=`, `%=`, which do not create a new variable but update the value of the variable already defined. Thus, the assignment `x += 1` is equivalent to `x := x + 1`.

## 5.24 Vector arithmetic

The dpic grammar permits the following:

```
X: Y + Z
```

where Y and Z are defined positions. Gpic requires

```
X: Y + (Z.x,Z.y)
```

Dpic also allows scalar postmultiplication:

```
X: Y*2/3 (but not X: 2/3*Y)
```

## 5.25 exec

In dpic the contents of a string can be executed using

```
exec string
```

or

```
exec sprintf(format string, args)
```

as if the string were the next line of input. This enables the programmed generation of names and labels, for example:

```
for i=1 to 10 do exec sprintf("A%g: x%g,y%g",i,2*i,3*i)
```

## 5.26 Functions

A few additional mathematical functions are defined in dpic: `abs`, `acos`, `asin`, `expe`, `floor`, `loge`, `sign`, `tan`, and `pmod`.

## 6 References

- [1] J. D. Aplevich. M4 macros for electric circuit diagrams in latex documents, 2015. Available with the CTAN Circuit\_macros distribution: [http://www.ctan.org/tex-archive/graphics/circuit\\_macros/doc/Circuit\\_macros.pdf](http://www.ctan.org/tex-archive/graphics/circuit_macros/doc/Circuit_macros.pdf).
- [2] B. W. Kernighan and D. M. Richie. The M4 macro processor. Technical report, Bell Laboratories, 1977. <https://wolfram.schneider.org/bsd/7thEdManVol2/m4/m4.pdf>.
- [3] B. W. Kernighan and D. M. Richie. PIC—A graphics language for typesetting, user manual. Technical Report 116, AT&T Bell Laboratories, 1991. <http://doc.cat-v.org/unix/v10/10thEdMan/pic.pdf>.
- [4] E. S. Raymond. Making pictures with GNU PIC. In GNU groff source distribution; also <http://www.kohala.com/start/troff/gpic.raymond.ps>.
- [5] R. Seindal *et al.* GNU m4, 1994. <http://www.gnu.org/software/m4/manual/m4.html>.



## 7 Appendix A: dpic man page

### NAME

dpic - convert pic-language input to LaTeX-compatible and other formats

### SYNOPSIS

```
dpic [ -defghmprstvx ] [ -z ] [ infile ] [> outfile ]
```

Typically, *infile* is of the form *name* [.pic] and *outfile* is of the form *name* .[tex|eps|pdf|fig|mp|svg]

### OPTIONS

(none) LaTeX picture-environment output (very limited font-based drawing commands)  
 -d PDF output, basic strings  
 -e pict2e output  
 -f Postscript output, strings in psfrag format  
 -g TikZ-pgf output  
 -h write help message and quit  
 -m mfpic output (see mfpic documentation)  
 -p PSTricks output  
 -r raw Postscript output, no automatic string formatting  
 -s MetaPost output  
 -t eepicemu output (slightly less limited than LaTeX picture output)  
 -v svg output  
 -x xfig 3.2 output  
 -z safe mode (access to external files disabled)

### DESCRIPTION

Dpic accepts a tight subset of the pic drawing language accepted by GNU pic (sometimes named gpic) or AT&T pic, and emits lower-level drawing commands for insertion into LaTeX documents, for processing by the xfig or Inkscape drawing tools, or for direct display as encapsulated Postscript, PDF, or SVG. LaTeX-compatible output can contain arbitrary text for formatting. Commands to be passed through to the postprocessor (PSTricks, Tikz-pgf, etc.) can be included. Dpic returns EXIT\_SUCCESS (normally 0) if messages no more severe than warnings were generated, otherwise EXIT\_FAILURE (normally 1).

A few pic-language extensions unique to dpic are implemented for specific purposes.

## LANGUAGE SUMMARY

Input consists of a sequence of ASCII text lines. The characters **LF** (`\n`) and **CR** (`\r`) have identical effect, and **ETX** (`\003`) is reserved for internal use. Characters outside the range of 0 to 127 are allowed in strings but receive no special treatment.

The first line of a picture is **.PS** and the last is **.PE**, with lines between these two converted into low-level drawing commands for the postprocessor chosen by the option. Lines outside of **.PS ... .PE** pass through dpic unchanged.

### Coordinates

Coordinate axes in the pic language point to the right and up. The drawing units are inches by default but the statement

```
scale = 25.4
```

at the beginning of the diagram has the effect of changing the units to millimetres.

### Drawn objects

The primitive drawn objects consist of the planar objects **box**, **circle**, **ellipse**; the linear objects **line**, **arrow**, **move**, **spline**; and **arc**, which has characteristics of both planar and linear objects. A *block* is a pair of square brackets enclosing other objects: `[ anything ]` and is a planar object. The complete diagram is contained implicitly in a block.

A *string* is a planar object similar to a box, but the pic language also allows strings to be attached to other objects as overlays, in which case they are part of the object.

The current drawing position **Here** is always defined. Initially and at the beginning of a block, **Here** is 0,0. Similarly, the current direction, which can be any of **up**, **down**, **left**, **right**, is set as **right** initially.

Each drawn object has an entry point and exit point that depend on the current direction. The entry point is placed by default at the current position. Objects can also be placed explicitly with respect to absolute coordinates or relative to other objects. The exit point becomes the new current position.

### Labels

A *label* in pic is an alphanumeric sequence that starts with an uppercase letter. Dpic allows variables and labels to be subscripted; thus **T** and **T[5]** are distinct labels. The value in brackets can be any expression, comma-separated expression pair, or a defined position, but expressions are rounded to the nearest integer value. A label gives a symbolic name to a position or drawn object; for example,

```
Post: Here + (1,2)
Bus[23]: line right 4
```

### Defined points

Once drawn, a linear object has defined points **.start**, **.center**, and **.end**, which can be referenced as known positions, for example,

```
L: line; line up 0.5 from L.c
```

where `.center` has been abbreviated as `.c`

The defined points for a planar object are the compass points on its periphery given by `.n`, `.s`, `.e`, `.w`, `.nw`, `.ne`, `.se`, `.sw`, together with `.center`, `.top`, `.bottom`, `.right`, `.left`. For an arc, these points correspond to the circle of which the arc is a part, with the addition of `.start` and `.end`.

A *block* has defined points similar to a box, but can also have others in its interior. For the example

```
A: [ circle; Q: [ line; circle ]; T: Q.n ]
```

the defined points are as follows:

The points of the outer block as if it were a box, for example, `A.ne`

A position defined in the block, for example, `A.T`

The defined points of labeled objects inside the block, preceded by a dot, for example, `A.Q` (the centre of block `Q`), or `A.Q.ne` (the northeast point of `Q`).

The defined points of enumerated objects inside the block, preceded by a dot (but make sure there is a space after the dot if it is followed by a number rather than an expression in braces), for example, `A.Q. 1st circle.n` or (better) `A.Q.{1}st circle.n`

### Language elements

The lines defining a picture are separated by newline characters or semicolons. Newlines are significant except after `then`, `;`, `:`, `{`, `else`, or newline.

A line is continued to the next if the rightmost character is a backslash.

Non-continuation lines beginning with a period are ignored, except for `.PS` and `.PE` lines.

The pic source may be commented by placing each comment to the right of a `#` character (unless the `#` is in a string).

The language elements include the following:

A drawing command with optional label, for example, `box` or `A: box`

A position-label definition, for example `A: position`

An assignment to a variable, for example `x = 5`

A *direction* (to change the default), for example, `up`

Branching is performed by

```
if expr then { dotrue } else { dofalse }.
```

The looping facility is

```
for variable = expr to expr [by [*] incr ] do { anything }.
```

The loop variable is incremented by 1 by default, otherwise by *incr* (which may be a negative expression) unless it is preceded by the asterisk, in which case the loop variable is multiplied by *incr*. The loop variable may be changed by statements in the loop, thereby controlling the number of loop repetitions.

Braces occur in several contexts. When used independently of other language elements, as

```
{ anything }
```

the left brace saves the current position and direction, and the right brace restores them to the saved values after the enclosed lines have been processed.

Arbitrary postprocessor commands can be generated using

```
command string,
```

which inserts the contents of *string* into the output. The *string* contents must be compatible with the chosen postprocessor. Similarly, any line that begins with a backslash is copied literally to the output.

The line

```
exec string
```

executes the contents of *string* as if it were a normal input line.

To execute operating-system shell commands, use

```
sh string
```

and to read lines from an external file, use

```
copy string
```

These commands are disabled by the dpic option **-z** or by a compile-time switch.

The command

```
print expr|position|string [ > string | >> string ]
```

prints or appends its argument to the file named in the string on the right, or by default to the standard error. Printing to a file is disabled by the **-z** option.

## Macros

The pic language includes macro definition and expansion, using either

```
define name { anything },
```

or

```
define name X anything X,
```

where X is any character except {. When *name* appears alone or with arguments in parentheses as *name(arg, ...)* then it is replaced by the contents between the delimiters in the macro definition. Arguments are separated by commas except that a comma in an argument is ignored within a string, parentheses, or brackets. In this substitution, occurrences of \$1 are replaced literally by the first argument (not by its value, if any), and so on for other arguments.

The value \$+ in a macro is the number of arguments given to the macro. dpic ignores white space (spaces, new lines, and tab characters) that directly precede an argument in a macro invocation. A macro definition can be deleted by

```
undefine name
```

Macro definitions are global, which may require judicious undefinition of macros if there is a risk of name clashes.

### Drawing commands

An object is drawn using the following general format:

```
[ Label :] object [ attributes ] [ placement ] [ strings ]
```

The items following *object* can occur in any order, but the order can make a difference to the drawn result, since lines are read and interpreted from left to right. Defaults are assumed for all drawing parameters as necessary. For example, the sequence

```
circle "Chew"; arrow; box "Swallow"
line; arc cw ->; ellipse "Digest"
```

draws a simple flow diagram using default sizes for all objects, with centered words in the circle, box, and ellipse.

### Attributes

An *attribute* is a keyword or keywords followed by expressions as appropriate. Attributes are used to set parameters that control the placement, size, and appearance of objects.

The dimension attributes are the following, showing valid abbreviations:

```
height|ht|width|wid|radius|rad|diameter|diam|scaled expr
```

When appended to linear objects, **height** and **width** apply to arrowhead dimensions. The **scaled** attribute scales the object by *expr*.

The postprocessed size of a string is unknown in advance to the pic processor, but once known, the bounding box dimensions can be specified explicitly as for other drawn objects, as shown:

```
string wid expr ht expr
```

The thickness of lines defining an object are modified using the environment variable **linethick** or the attribute

`thickness|thick expr`

expressed in points. Line thickness is independent of any scaling.

Solid lines are drawn by default; this can be modified with

`solid|invisible|invis`

or with

`dotted|dashed [ expr ]`

the optional expression in the latter setting the length and distance between dashes or dots.

The following attributes are for putting arrowheads at the start, end, or both ends of a linear object:

`<-|->|<-> [ expr ]`

The default for **arrow** objects is `->`. The shape parameter *expr* may be omitted, in which case the value of the environment variable **arrowhead** is used. The accepted values of *expr* are currently 0, 1, and 3, with 1 the default.

The drawing direction of an arc is changed by the attribute

`ccw|cw`

with `ccw` the default.

To fill an object with a shade of gray, use the attribute

`fill [ expr ]`

where the value of *expr* can vary from 1, meaning white, to 0, meaning black. A linear object defining a path can be filled where the postprocessor allows, currently for MFpic, MPost, PDF, PGF, PS, PSfrag, PSTricks, and SVG output.

Line color can be set using

`outline|outlined string`

The pic language knows no details about color; the string contents must be compatible with the postprocessor. For example, the predefined colours of LaTeX or Tikz-pdf packages can be specified, or custom colors can be defined using the

`command string`

facility so that the postprocessor will know about them.

Filling by color is similar, using the attribute

`shaded string`

and, when the object is planar and both the fill and outline colors are the same, the

two attributes can be combined as

`colour|color|colored|coloured string`

in which all four spellings are equivalent.

Finally, the attribute

`same`

duplicates the properties of the previously drawn object of the same type, but with the current default placement.

In addition to scale changes effected by the `scale` variable, the size of the complete picture can be set by appending one or two terms to the `.PS` line as shown:

`.PS [x [y]]`

where `x` and `y` evaluate to constant values. On encountering the `.PE` line, the picture width `w` and height `h` are calculated. If `x > 0` then the picture is scaled so that `w = x`. If `h > y > 0` or if `x = 0` and `y > 0` then the picture is scaled so that `h = y`. Horizontal and vertical scaling are not independent. Text size, line thickness, and arrowheads are not scaled. The units are inches, so for example,

`.PS 100/25.4`

sets the final picture width to 100 mm. Printed string text may extend beyond the rectangular boundaries defined by `w` and `h` unless the text dimensions have been explicitly set.

If the final diagram width exceeds the environment variable `maxpswid` or the height exceeds `maxpsht`, both of which can be changed by assignment, then the diagram is scaled as for `x` and `y` above.

### Placement of drawn objects

An object is placed by default so that its entry is at the current point.

Explicit placement is obtained with

`object at position`

which centers the object at *position*, or

`object with defined point at position`

for example,

`arc cw from position to position with .c at position`

A block can also be positioned by reference to a displacement from its lower left corner, for example,

`A: [ contents ] with (0.5,0.2) at position.`

Linear objects are placed by default with the `.start` point placed at the current drawing position; otherwise linear objects are defined using a *linespec*, which is of the form

```
linespec = from position | to position | direction [ expr ]
           | linespec linespec
           | linespec then linespec
```

where the second line means that *linespecs* can be concatenated, and the third that multisegment linear objects are drawn using multiple *linespecs* separated by `then`.

As an example, the following draws a triangle with the leftmost vertex at the current point:

```
line up 2 right 1.5 then down 3 then to Here
```

Exceptionally, the *linespec*

```
to position to position to ...
```

is multiple and does not require the `then` keyword, but this also means that the *linespec*

```
up expression to position
```

is multiple and creates a two-segment line.

A single *expr* is also an acceptable *linespec* immediately after a linear object and means that the *object* is drawn to length *expr* in the current direction. The exception to this is

```
spline [ expr ] linespec
```

for which the *expr* is a spline tension parameter. If *expr* is omitted, a straight line is drawn to the midpoint of the first two spline control points and from the midpoint of the last two to the last point; the spline is tangent to all midpoints between control points. If *expr* is present, the spline is tangent at the first and last control points and at all other midpoints, and the spline tension can be adjusted. Tension values between 0 and 1 are typical.

In cases where all of the points of a multisegment linear object are not known in advance or inconvenient to calculate, the drawing command

```
continue linespec
```

will append a segment to the previously drawn linear object as if `continue` were `then`, with two differences. Arbitrary calculations may be done between the previous object and the `continue` statement, and the current point is the exit point of the previous object.

The construction

```
line from position to position chop [ expr ]
```



truncates the line at each end by the value of *expr* or, if *expr* is omitted, by the current value of environment variable `circclerad`

Otherwise

`line from position to position chop expr1 chop expr2`

truncates the line by the two specified distances at the ends. Truncation values can be negative.

The attribute

`by position`

is for positioning, for example,

`move by (5,6)`

### Variables and expressions

Variable names are alphameric sequences beginning with a lower-case letter, optionally subscripted as for labels, and are defined by assignment. For example, the following line defines the variable `x` if it does not already exist in the current scope:

`x = expr`

The scope of pic variables is the current block in which they are defined, including blocks defined later within the current block. The assignment

`x := expr`

or any assignment using an operator in the set

`:= += -= *= /= %=`

requires `x` to have been defined previously in the current block or an enclosing block.

Expressions consist of floating-point values combined using the unary operator `!` for logical negation and the usual parentheses and binary operators in decreasing order of precedence:

```

^
* / %
+ -
== != >= <= < >
&&
||

```

In logical tests, the value 0 is equivalent to false and a nonzero value to true, with resulting true value of 1.

A floating-point value is obtained as an integer, a number with `e` syntax, a function value, a size value of a drawn object, for example,

`last box.ht,`

or the horizontal or vertical coordinate of a **position**, obtained respectively as

*position* .x|.y

The single-argument functions are **abs**, **acos**, **asin**, **cos**, **exp**, **expe**, **int**, **log**, **loge**, **sign**, **sin**, **sqrt**, **tan**, **floor**. The functions **exp** and **log** are base 10. The function **rand()** delivers a random number between 0 and 1, and **rand** (*expr*) initializes the random number generator.

The two-argument functions are **atan2**, **max**, **min**, **pmod** where **pmod** is the modulo function delivering a positive value.

### Predefined variables

The following variables are predefined on invocation of **dpic**: **optTeX**, **opttTeX**, **optPict2e**, **optPSTricks**, **optPDF**, **optPGF**, **optMFpic**, **optPS**, **optPSfrag**, **optMpost**, and **optSVG**. Variable **dpicopt** is set according to the output option chosen, so that if one of options **-p** or **-g** has been invoked for example, then the test

```
if dpicopt == optPSTricks || dpicopt == optPGF then { ... }
```

will selectively execute its body statements. In addition, variable **optsafe** has value **true** if the **-z** option has been selected or **dpic** has been compiled in safe mode.

If the specified option is one of **-v** (SVG), **-d** (PDF), or **-r** (raw postscript), then the variable **dpextratio** is defined, with default value 0.66, together with the variable **dpPPI** with default value 0.96. The latter is the assumed pixel density per inch and the former is the ratio of text height to nominal point size. These variables are used by the output routines and can be changed by assignment.

If the **-x** (xfig) option has been specified, then the following two variables are predefined: **xfigres** (default value 1200), and **xdispres** (default 80).

### Predefined environment variables

A set of environment variables establishes the default values of drawing parameters within the scope of the current block. Their values are inherited from the superior block, but can be changed by assignment. They can be used in expressions like other variables. The variables, their default values, and default uses are given below

<b>arcrad</b>	0.25 arc radius
<b>arrowht</b>	0.1 length of arrowhead
<b>arrowwid</b>	0.05 width of arrowhead
<b>boxht</b>	0.5 box height
<b>boxrad</b>	0 radius of rounded box corners
<b>boxwid</b>	0.75 box width
<b>circlerad</b>	0.25 circle radius
<b>dashwid</b>	0.05 dash length for dashed lines
<b>ellipseht</b>	0.5 ellipse height
<b>ellipsewid</b>	0.75 ellipse width
<b>lineht</b>	0.5 height of vertical lines
<b>linewid</b>	0.5 length of horizontal lines
<b>moveht</b>	0.5 length of vertical moves

<b>movewid</b>	0.5	length of horizontal moves
<b>textht</b>	0	assumed height of text (11pt for postscript, PDF, and SVG)
<b>textoffset</b>	2.5/72	text justification gap
<b>textwid</b>	0	assumed width of text

When a value is assigned to the environment variable **scale**, all of the above values are multiplied by the new value of **scale**. This is normally done once at the top of the outermost scope of a diagram. Drawing units are thereby changed but the default physical sizes of drawn objects remain unchanged since dimensions are divided by the outermost **scale** value on output. The following are unaffected by **scale** changes:

<b>arrowhead</b>	1	arrowhead shape
<b>fillval</b>	0.5	fill density
<b>linethick</b>	0.8	line thickness in points
<b>maxpsht</b>	11.5	maximum allowed diagram height
<b>maxpswid</b>	8.5	maximum allowed diagram width

The variables **maxpswid** and **maxpsht** may have to be redefined for large diagrams or landscape figures, for example.

## Positions

A *position* is equivalent to a coordinate pair defined in current drawing units, and can be expressed in the following forms:

**Here** The current drawing position.

*expr,expr* A pair of expressions separated by a comma.

( *position* ) A position in parentheses for grouping.

( *position* , *position* ) Takes the horizontal value from the first position and the vertical value from the second.

*position* +|- *position* Vector addition.

*position* \*// *expr* Scalar postmultiplication.

*Label* The label of a defined position or object. The position is the center of the object.

*expr* [of the way] **between** *position* and *position*

The example **x between A and B** is equal to  $A*(1-x) + B*x$ . Any value of *expr* is allowed.

*expr* < *position*, *position* > An abbreviated equivalent of the previous form.

*number* **st|rd|nd|th** [*last*] *object* An object within the current block, enumerated in order of definition.

The *object* is one of

**line, move, arrow, arc, box, ellipse, circle, spline, [], ""** .

The number can be replaced by `{ expr }`. For example, `last ""` means the last string, and `{2^2}nd []` means the fourth block in the current scope. The position is the center of the object.

Parentheses may be required when composite positions or expressions are used in the above forms.

Finally, a position can be given as

*object . defined point*

## Strings

A *string* is a sequence of characters enclosed in double quotes. To include a double quote in a string, precede it with a backslash. Strings can be concatenated using the `+` operator. The C-like function

```
sprintf( format string, expr, ... )
```

is equivalent to a string. (Its implementation passes arguments singly to the C `sprintf` function.) Expressions are floating-point values, so the only applicable number formats are `e`, `f`, and `g`.

Multiple strings such as `"text1" "text2"` are stacked and centered vertically.

A string attached to an object overlays the object at the center, and any `height` or `width` attributes apply to the object, not the string. However, the justification attributes `ljust` and `rjust` can be applied to the individual strings of a stack overlaying an object.

An independent string is placed with its center at the current point by default, or by specifying the position of one of its defined points as for any *object*, for example,

```
"Animal crackers" wid 82.3/72 ht 9.7/72 with .sw at Q
```

The placement qualifiers `above`, `below`, `ljust`, `rjust` place the string above, below, or justified with respect to the placement point. For example,

```
"Animal crackers" at Q ljust above
```

places the string above and `textoffset` units to the right of `Q`.

## EXAMPLES

Source file `example.pic`:

```
\documentclass{article}
\usepackage{tikz}
\begin{document}
.PS
box dashed "Hello" "World"
.PE
```

```
\end{document}
```

The command

```
dpic -g example.pic > example.tex; pdflatex example
```

produces example.pdf containing a dashed box with Hello and World stacked inside.

To produce a .tex file containing PSTricks drawing commands for insertion into a LaTeX document using the \input command, delete the first three and last lines in the above source and process using the -p option of dpic.

Similarly, the picture source

```
.PS
\definecolor{puce}{rgb}{0.8,0.53,0.6}%
box shaded "puce"
.PE
```

produces a box filled with a flea-like color when processed with `dpic -g` or `dpic -p` and the resulting file is inserted into a latex source file invoking, respectively, the `tikz` or `pstricks` package.

## SEE ALSO

E. S. Raymond, *Making Pictures with GNU PIC*, 1995. In GNU groff source distribution; <http://www.kohala.com/start/troff/gpic.raymond.ps> (A good introduction to the pic language, with elementary illustrations.)

J. D. Aplevich, *Drawing with dpic*, 2022, <http://ece.uwaterloo.ca/~aplevich/dpic/dpic-doc.pdf> (Specific discussion of dpic facilities and extensions, with differences between dpic and GNU pic.)

B. W. Kernighan, B. W. and D. M. Richie, *PIC - A Graphics Language for Type-setting, User Manual*, 1991. AT&T Bell Laboratories, Computing Science Technical Report 116. (The original Unix pic.)

J. D. Aplevich, *M4 Macros for Electric Circuit Diagrams in LaTeX Documents*, 2022. File Circuit\_macros.pdf in the graphics/Circuit\_macros section of CTAN repositories. (Extension of the pic language using the m4 macro processor for drawing electric circuits and other diagrams.)

## AUTHOR

Dwight Aplevich <aplevich at uwaterloo dot ca>

## 8 Appendix B: dpic grammar

The tokens recognized by the dpic parser are as shown. The tokens in `< >` pairs are generated internally. Equated tokens (`" ; " = "<NL>"`) have identical effect.

```

terminals:
  "<" "cw" "ccw"
  "(" ")" "*" "+" "-" "/" "%" ";" = "<NL>"
  "^" "!" "&&" "|"
  "," ":" "[" "]" "{" "}" "." "[]" "‘" "’"
  "=" ":=" "+=" "-=" "*=" "/=" "%=" (* the order matters *)
  "&"
  "<float>" "<name>" "<Label>" "<LaTeX>"
  ‘ ’ = "<string>"
  "#"
  "$" = "<arg>"
  ".PS" ".PE"
  "ht" = "height" "wid" = "width"
  "rad" = "radius" "diam" = "diameter"
  "thick" = "thickness"
  "scaled"
  "from" "to" "at" "with" "by" "then" "continue"
  "chop" "same"
  "of" "the" "way" "between" "and" "Here"
  "st" = "rd" = "nd" = "th" "last"
  "fill" = "filled"
  ".x" ".y"
  "print" "copy" "reset" "exec" "sh" "command"
  "define" "undef" = "undefine"
  "rand"
  "if" "else" "for" "do" "<endfor>"
  "sprintf"
  "<corner>"
  ".ne" ".se" ".nw" ".sw"
  ".t" = ".top" = "top" = ".north" = ".n"
  ".b" = ".bot" = ".bottom" = "bottom" = ".south" = ".s"
  ".right" = ".r" = ".east" = ".e"
  ".left" = ".l" = ".west" = ".w"
  ".start" = "start"
  ".end" = "end"
  ".center" = ".centre" = ".c"
  "<compare>"
  "==" "!=" ">=" "<=" ">"
  "<param>"
  ".height" = ".ht"
  ".width" = ".wid"
  ".radius" = ".rad"

```

```

        ".diameter" = ".diam"
        ".thickness" = ".thick"
        ".length"
"<func1>"
        "abs" "acos" "asin" "cos" "exp" "expe" "int" "log" "loge"
        "sign" "sin" "sqrt" "tan" "floor"
"<func2>"
        "atan2" "max" "min" "pmod"
"<linetype>"
        "solid" "dotted" "dashed" "invis" = "invisible"
"<colrspec>"
        "color" = "colour" = "colored" = "coloured"
        "outline" = "outlined"
        "shade" = "shaded"
"<textpos>"
        "centre" = "center" "ljust" "rjust" "above" "below"
"<arrowhd>"
        "<->" "<->" "<->"
"<directon>"
        "up" "down" "right" "left"
"<primitiv>"
        "box" "circle" "ellipse" "arc" "line" "arrow" "move" "spline"
"<envvar>"
        "arcrad" "arrowht" "arrowwid" "boxht" "boxrad" "boxwid"
        "circlerad" "dashwid" "ellipseht" "ellipsewid" "lineht"
        "linewid" "moveht" "movewid" "textht" "textoffset" "textwid"
= "<lastsc>"          (* marker for last scaled env var *)
        "arrowhead" "fillval" "linethick" "maxpsht" "maxpswid" "scale"
= "<lastenv>"          (* marker for last env var *)

```

The expanded dpic grammar follows:

```

input: %empty
      | input picture NL

picture: prestart psline NL elementlist optnl ".PE"

NL: ";"
   | error

prestart: %empty

psline: ".PS"
        | ".PS" term
        | ".PS" term term

elementlist: %empty
            | element

```

```

        | elementlist NL element

term: factor
    | term "*" factor
    | term "/" factor
    | term "%" factor

element: namedobj
    | "label" suffix ":" position
    | assignlist
    | "direction: up down right left"
    | "LaTeX"
    | command
    | lbrace elementlist optnl "}"
    | ifpart
    | elsehead elementlist optnl "}"
    | for "}"
    | "command" stringexpr
    | "exec" stringexpr

lbrace: "{"

namedobj: object
    | "label" suffix ":" object

suffix: %empty
    | "[" expression "]"
    | "[" position "]"

position: pair
    | expression "between" position "and" position
    | expression "of" "the" "way" "between" position "and" position
    | expression "<" position "," position "logical operator" shift

assignlist: assignment
    | assignlist "," assignment

command: "print" expression redirect
    | "print" position redirect
    | "print" stringexpr redirect
    | "reset"
    | "reset" resetlist
    | systemcmd
    | "copy" stringexpr
    | "define" "identifier"
    | "define" "label"
    | "undef" "identifier"
    | "undef" "label"

```



```

optnl: %empty
      | NL

ifpart: ifhead elementlist optnl "}"

elsehead: ifpart "else" "{"

for: forhead elementlist optnl
    | for forincr elementlist optnl

stringexpr: string
           | stringexpr "+" string

string: "string"
       | sprintf ")"

assignment: "identifier" suffix "=" assignrhs
           | "environment variable" "=" assignrhs

assignrhs: expression
          | assignment

expression: term
           | "+" term
           | "-" term
           | expression "+" term
           | expression "-" term

ifhead: setlogic logexpr "then" "{"

setlogic: "if"

logexpr: logprod
        | logexpr "||" logprod

forhead: "for" assignlist "to" expression do "{"

forincr: "end of for {...} contents"

do: "do"
   | by expression "do"

by: "by"
   | "by" "*"

redirect: %empty
        | "logical operator" stringexpr

```

DPIC(1)

DPIC(1)

```
| "logical operator" "logical operator" stringexpr

resetlist: "environment variable"
| resetlist "," "environment variable"

systemcmd: "sh" stringexpr

sprintf: "sprintf" "(" stringexpr
| "sprintf" "(" stringexpr "," exprlist

exprlist: expression
| expression "," exprlist

object: block
| object "ht" expression
| object "wid" expression
| object "rad" expression
| object "diam" expression
| object "thick" expression
| object "scaled" expression
| object "direction: up down right left" optexp
| object "line type: dotted etc" optexp
| object "chop" optexp
| object "fill" optexp
| object "arrowhead parameter: <- -> <->" optexp
| object "then"
| object "cw"
| object "ccw"
| object "same"
| object stringexpr
| object "by" position
| object "from" position
| object "to" position
| object "at" position
| object "text position: ljust rjust above below center"
| object "color spec: colored outlined shaded" stringexpr
| objectwith "at" position
| objectwith "compass corner: .n .ne .center .end etc" "at" position
| objectwith pair "at" position
| "continue"

openblock: "["

block: "drawn object: box circle line etc" optexp
| stringexpr
| openblock closeblock "]"
| "[]"
```

DPIC(1)

```
optexp: %empty
      | expression

closeblock: elementlist optnl

objectwith: object "with"
          | objectwith "." "label" suffix
          | objectwith "." nth primobj

pair: expression "," expression
     | location shift

nth: ncount "ordinal: st or rd or nd or th"
    | ncount "ordinal: st or rd or nd or th" "last"
    | "last"

primobj: "drawn object: box circle line etc"
        | "["
        | "string"
        | "[" "]"

shift: %empty
      | shift "+" location
      | shift "-" location

location: "(" position ")"
         | "(" position "," position ")"
         | place
         | location "*" factor
         | location "/" factor

place: placename
      | placename "compass corner: .n .ne .center .end etc"
      | "compass corner: .n .ne .center .end etc" placename
      | "compass corner: .n .ne .center .end etc" "of" placename
      | "Here"

factor: primary
       | "!" primary
       | primary "^" factor

placename: "label" suffix
          | nth primobj
          | placename "." "label" suffix
          | placename "." nth primobj

ncount: "number"
        | "' expression '"
```

DPIC(1)

DPIC(1)

```
| "{" expression "}"

logprod: logval
| logprod "&&" logval

logval: lcompare
| stringexpr "<" stringexpr
| expression "<" expression

lcompare: expression
| stringexpr
| lcompare "logical operator" expression
| lcompare "logical operator" stringexpr

primary: "environment variable"
| "identifier" suffix
| "number"
| "(" logexpr ")"
| location ".x"
| location ".y"
| placename "attribute .ht .wid etc"
| "rand" "(" ")"
| "rand" "(" expression ")"
| "function (1 arg)" "(" expression ")"
| "function (2 args)" "(" expression "," expression ")"
| "(" assignlist ")"
| "(" systemcmd ")"
```

DPIC(1)