Drawing with dpic

Dwight Aplevich[*]

May 3, 2013

Contents

---

1 Introduction

This document is meant for persons using dpic to produce diagrams for LaTeX 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]. A concise dpic language reference is included here in Appendix A, and the context-free dpic input grammar is given in Appendix B.

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 LaTeX documents.

The pic language is particularly suited for easily creating line diagrams such as electric circuits, and many persons use a package of macros called Circuit_macros along with the m4 macro processor. The pic language itself allows macro definitions, and both pic and m4 macros will be described.

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, of course: 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 some 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 LaTeX, PSTricks, TikZ-PGF, mfpic processing):

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

To produce other formats:
   dpic -[e|f|g|h|m|p|r|s|t|v|x] [-z] *file*.pic > *file*.[tex|eps|mp|fig|svg]

2.1 Options

Dpic accepts the following options:

   (none) Latex drawing commands
   -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 (sh, copy, and print to file disabled)

The `-p` option produces output for postprocessing by the 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 is 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 `-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. The 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 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.

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

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 LaTeX drawing commands by default. LaTeX 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 LaTeX, 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 Lexical error messages

Lexical error messages are generated automatically, and state the lexical value that was found and what was expected. Some non-obvious lexical values are:

`corner`	a compass corner or `.c`, `.start`, `.end`
`envvar`	environmental variable
`primitiv`	`box`, `circle`, `ellipse`, `arc`, `line`, `arrow`, `move`, `spline`

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.

Pic is very suitable for line diagrams, and recent language extensions allow for 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 features. 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 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 composite object, is a group of elements enclosed by square brackets, such as
  `Q: [ B: [A: arc]; circle ].`

3.1.1 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.

3.1.2 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.

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 3.2 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.

3.2 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 its 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 for drawing electric circuits and other diagrams using dpic and m4 [1], 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. Then any separate appearance of *name* in the following lines is replaced by the characters between the defining braces, including newlines. If the name is given arguments of the form
```
name(x, y, z, ...)
```
then, in the macro body, $n expands to the nth argument, which may be nul, if at least one argument has been defined and n is a positive integer. Otherwise $n is not evaluated.

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 enclosed within strings or ( ) parentheses. 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.

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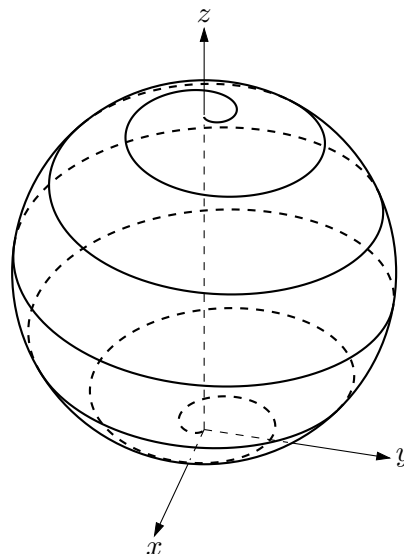
3.2.2 Composing statements

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 }
```
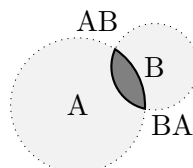


```
shade(0.5,lozenge(A,B))
```

3.2.3 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 i=1 to $+ do { c[i] = $i }

Instead, use the dpic statement

    exec *string*

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

    sprintf("<i>format</i>**",**<i>expression</i>**,...)**

behaves like a string, the following works:

    for i=1 to $+ do { exec sprintf("c[i] = $%g",i) }

3.2.4 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 Hiding variables

As mentioned in Section 3.1.2, 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.

3.3.1 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 i=1 to 2 do {
   if x > $2 then { i = 2 } else { print x; x = x*2; i=1 } } }
```

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( x=1, x<10, x*=2,
    loop( y=1, y<=4, y+=1,
     print (x,y) ) )
```

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;}
```

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 library.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 LaTeX, enclose them in single quote pairs.

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

3.4.1   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 the name of 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,svg,`SVG code',
 `default code')
```

3.4.2   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)')dnl
  define('m4pdy','('$1'.end.y-'$1'.start.y)')dnl
  -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) and point_(radians),
```
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) \
```

12

```
    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 capitalized labels to have subscripts; thus `x` and `x[4]` are distinct
variable names, and can be employed in expressions as usual. Similarly, `P` and `P[3]` are distinct
labels.

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 define 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
```
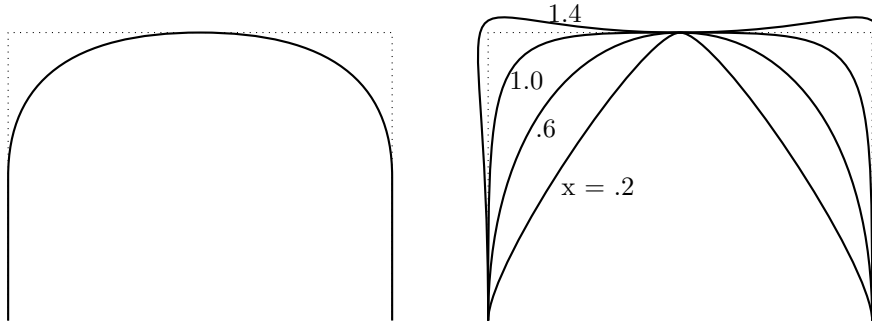

A straight line is drawn along the first half of the first segment and the last half of the last segment. The curve is tangent to the centres of the segments. The dpic result of including an expression after `spline`, as in

```
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]`, ... `X[n]` has been given, and a spline is to be found to pass through each point. The control points `P[0]`, `P[1]`, ... `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] for i = 1 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 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 dpic output.

Dpic lines beginning with the backslash \ are passed through to the output without modification. This method works well for TeX statements, LaTeX statements, and commands to

postprocessors that rely on the 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.

3.7.1   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 does 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.

3.7.2   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. For example, the definitions

```
define('gold_','1,0.84,0')
ifelse(m4postprocessor,pstricks,
 'command "\definecolor{'gold'}{rgb}{gold_}"',
m4postprocessor,pgf,
 'command "\definecolor{'gold'}{rgb}{gold_}"',
m4postprocessor,mpost,
 'define('gold','(gold_)')',
m4postprocessor,postscript,
 'define('gold','1 0.84 0')')
```

allow the command
```
   circle shaded "gold"
```
to work for the three different postprocessors specified plus direct postscript output.

More elaborate options can also be invoked. For example, 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. The language was never explicitly designed to draw filled objects of arbitrary shape. 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 { [
  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 gpic 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 gpic example. The above macro can be modified easily to produce arbitrary polygons, for example. The color `Dandelion` is automatically defined for PSTricks by the

LaTeX line
```
\usepackage[dvipsnames]{pstricks}
```
For TikZ-PGF, try
```
\usepackage[usenames,dvipsnames]{xcolor}
\usepackage{tikz}
```

4   SVG and Postscript output

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

4.1   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, over-painting, arbitrary text, arbitrary Postscript or SVG, and other complications are allowed; consequently dpic can only provide an estimate of the exact bounding box.

There is apparently no reliable way to know the exact bounding box of arbitrary SVG text, so dpic uses an approximation and text placement on diagrams may have to be adjusted by hand.

Truncated text: The dpic `textht` environmental variable 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 often 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. `"string" wid 0.75` often serves as a quick cure in specific cases, but cannot be done automatically. Otherwise, strategic `move` commands can be used to enlarge the bounding box as illustrated at the end of the discussion below.

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.

Manual bounding box adjustment: It may be necessary to adjust the bounding box manually. To zero the automatic adjustment for Postscript or SVG output, 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 gpic particularly, are summarized below. Normally, the only changes required to process correct pic or gpic 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. The remaining differences documented below are small syntactical differences or relate to the use of PSTricks or the other dpic output formats. Sometimes, differences in default syntax (such as for arcs) must be considered.

Gpic is being actively maintained so some of the items below apply only to older versions.

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 . lines and program-generated pic

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 lines beginning with .
within pictures. Some programs (e.g., pstoedit) add Troff comment lines beginning with .\"
outside the `.PS`, `.PE` delimiters. These lines must be dealt with separately.

5.4 \ lines

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

5.5 `for`-loop and `if` bodies

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

5.6 End of line

The line end is significant in the pic grammar. 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 }
```

where \ is the last character of the line or is followed by the # character beginning a comment
which ends at the end of the line. 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.

19

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 to guarantee identical behavior. In addition, put string comparisons in parentheses, e.g.
  ("*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 then

Versions of gpic up to 1.19 ignore the then keyword, so that
  line -> then up 0.5
draws one line segment and is the same as
  line -> up 0.5
whereas dpic does not ignore then, and draws two line segments. Newer versions of gpic also draw two segments.

5.9 line, spline, arrow, move

Dpic treats the defaults for linear objects consistently with planar objects with respect to the at modifier. Versions of gpic up to 1.19 treated 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.

The compass corners of multisegmented linear objects are not precisely defined and they should be used with care.

5.10 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, with centres on opposite sides of the line joining A and B, that satisfy this specification, and the interpreter will choose one. Instead,

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

5.11   Strings

Strings are arbitrary character sequences between double quotes, with double quotes in strings preceded by the backslash character. Equivalently, a string is produced by the C-like sprintf function
```
sprintf("format" [, expression, ... ])
```
The C sprintf 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 the formatted string into the output. The numerical `sprintf` arguments can be arbitrary expressions rather than the constants shown.

Dpic allows strings to be concatenated by the `+` operator; thus,
```
"abc" + sprintf(" def%g",2)
```
is equivalent to `"abc def2"`.

Both dpic and gpic treat \ as an escape character prior to the quote character in a string, so `"\""` is a length-one string containing the double quote. Otherwise, the backslash is an ordinary string character. 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. Some previous versions of dpic allowed `$` to escape the dollar sign in strings but this created other difficulties.

Both dpic and gpic allow logical comparison of strings. Put the comparison in parentheses.

String height and width are unscaled on final output from dpic since these depend on later formatting by LaTeX.

5.12   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.

Dpic sends the print output to a file 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. 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 drawn pattern is required in order to draw a background object that will be overlaid by the pattern. 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
```

5.13 `command` *arg*, ...

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

5.14 Operating system commands

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

    sh "*text*"

or

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

5.15 `copy`

Dpic supports the command

    copy "*filename*"

but does not directly support the 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 comma, 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 by customization of the copythru macro to suit.

5.16   plot

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

5.17   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, but fill is supported by dpic using the shaded directive. 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.

5.18   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 in place of setting the scale environment variable when, for example, different parts of a diagram require different scaling. As always, line thicknesses are not scaled.

5.19   Arrowheads

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

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 (change only the currently drawn object):
  arrow thick 2 ht 0.18 wid 0.09 -> 0
```

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.

5.20   Compass corners

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
```

5.21 `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))
```

Input parsing would be possible if `continue` were to be replaced by `then`:
```
move to (0,0); line right 1; then to (Here,(2,1))
```
so that the keyword `continue` would not have been necessary, but it has been retained for historical compatibility.

5.22 Subscripted variables and locations

Dpic allows subscripted variables and positions, as an aid in geometric calculations such as fitting splines to a set of points. Thus `Pos[`*expression*`]` is a valid name for a position. The rounded integer value of the expression is used. Similarly, `x[`*expression*`]` is a valid variable name. There are no array operations, but it is easy to write macros for them.

5.23 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. However when x is present, dpic draws the spline from the start of the first segment to the end of the last segment.

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

5.25   Positions

If `X` is a position, then `(X)` is a valid position for dpic but not older versions of gpic, which give an error for
```
(a between A and B) + (x,y)
```
Dropping the parentheses to avoid the error gives
```
a between A and B + (x,y)
```
which is not the same resulting position. Use dpic or the latest gpic if you need this construction.

5.26   `int()`

Gpic `int()` up to version 1.19 computed the floor of its argument rather than the integer part as specified by ATT pic. Dpic provides both the `floor()` and `int()` functions but `int()` does not compute the same value as these versions of gpic `int()` for non-integer negative arguments.

5.27   `exec`

In dpic the contents of a string can be executed using
```
exec string
```
or
```
exec sprintf(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)
```
This effect can also be accomplished with a simple macro.

5.28   Functions

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

5.29   PSTricks anomaly

Some time in the mid 2000 to 2010 interval, the behaviour of the low-level PSTricks command `\psbezier` was changed to match the existing context-dependent behavior of the `\pscurve`

command within the `\pscustom` environment[1]. Dpic does not generate the `\pscurve` command, but the new behaviour affects dpic-defined filled splines. The "normal" PSTricks spline syntax is

`\psbezier(x1,y1)(x2,y2)(x3,y3)(x4,y4) ...`

However, if a correctly defined path containing this spline as its second or later subpath is now enclosed in a `\pscustom` environment, the path coordinates are incorrect; `(x1,y1)` must be omitted since, under these conditions, `\psbezier` takes its first coordinate pair to be the current position. This anomaly would not be a significant problem if it were always known at the time of generating the `\psbezier` coordinates whether the curve will be enclosed within `\pscustom`, but such is not the case.

Dpic always generates four (or more) coordinate-pair arguments as in the normal syntax, sometimes resulting in the addition of an extraneous path segment produced within `\pscustom`.

For some versions of PSTricks, a workaround for the problem has been to insert a PSTricks patch in your LaTeX code if you are enclosing splines within `\pscustom` commands. More recent versions of the PSTricks package have changed again but, starting from the August 27, 2010 `pstricks.tex` distribution file, a parameter `noCurrentPoint` has been introduced, which prevents `\psbezier` (and other commands) from taking the current point as the first coordinate pair. The following line inserted into a latex document avoids an error message if the PSTricks version is August 2010 or later:

` \psset{noCurrentPoint}`

Older versions of PSTricks are handled using the following workaround:

```
command "\makeatletter\@ifundefined{ifPst@noCurrentPoint}%
{\@ifundefined{MPS@Patch}{\gdef\MPS@Patch{}%
 \typeout{ Dpic -p: patching psbezier in pstricks.tex (some versions) }%
 \def\psbezier@ii{\addto@pscode{%
  \ifshowpoints true \else false \fi\tx@OpenBezier%
  \ifshowpoints\tx@BezierShowPoints\fi}\end@OpenObj}}{}}%
{\@ifundefined{MPS@PatchMsg}{\gdef\MPS@PatchMsg{}%
 \typeout{ Dpic -p: Setting noCurrentPoint to use all coord pairs }}{}%
 \psset{noCurrentPoint}}\makeatother%"
```

This patch need only be executed once per LaTeX run rather than with every diagram.

If the above fix does not work then, for now, do not insert `spline` commands into paths within the `\pscustom` environment, *unless* the spline is the first path segment. It is probably safe to say that this anomaly does not occur very often, but it can cause considerable difficulty when it does; the cure is to update your PSTricks files.

Legacy diagrams can often be treated in the following way, if pstricks97.tex is available on your machine as in MiKTeX distributions, for example. Replace the line `\usepackage{pstricks}`

---

[1]The context-dependency of `\pscurve` was taken to be a feature rather than a bug. An alternative that preserves command orthogonality would have been to define two curve (and spline) commands for the in-pscustom and out of pscustom contexts. Such are the difficulties of software library maintenance.

27

with `\usepackage[97]{pstricks}` or `\input{pstricks97.tex}`. However, this method precludes using the newer PSTricks facilities.

References

[1] J. D. Aplevich. M4 macros for electric circuit diagrams in latex documents, 2009. Available with the CTAN Circuit_macros distribution: `http://www.ctan.org/tex-archive/graphics/circuit_macros/doc/CMman.pdf`.

[2] B. W. Kernighan and D. M. Richie. The M4 macro processor. Technical report, Bell Laboratories, 1977.

[3] B. W. Kernighan and D. M. Richie. PIC—A graphics language for typesetting, user manual. Technical Report 116, AT&T Bell Laboratories, 1991.

[4] E. S. Raymond. Making pictures with GNU PIC, 1995. 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`.

6   Appendix A: dpic man page

NAME

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

SYNOPSIS

`dpic [ -efghmprstvxz ]` [*infile*] [> *outfile*]

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

OPTIONS

`(none)` LaTeX drawing output (very limited font-based drawing capability)
`-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 drawing 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 or SVG. Arbitrary text can be inserted into the output for later formatting, and arbitrary commands for 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 language extensions unique to dpic are implemented for specific purposes.

LANGUAGE SUMMARY

Input consists of a sequence of lines. 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, which 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 alphameric 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 but it is 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. Using 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 corner 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.

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 the 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

> **define** *name* { *anything* }**,**

so that, when *name* appears alone or with arguments as *name* (*arg, ...*) then it is replaced by the contents between the braces in the definition. A comma in an argument list is ignored within a string or parentheses. In this substitution, occurrences of $1 are replaced by the first argument, 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

The size and other parameters control the appearance of objects. An *attribute* is a keyword or keywords with expressions as appropriate.

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 environmental 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 shape parameter *expr* may be omitted, in which case the value of the environment variable `arrowhead` is used. The default for `arrow` is `->`.

The drawing direction of an arc is changed by the attribute

> `ccw|cw`

with `ccw` the default.

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

> `fill` [*expr*]

where a value of 1 means white, and 0 means black. Paths can be filled where the postprocessor allows.

Line color can be set using

> `outline|outlined` *string*

where the contents of the string depend on the postprocessor. The predefined colours of LaTeX or Tikz-pdf packages can be specified. The pic language knows no details about color; the string contents must be compatible with the postprocessor. Custom colors have to 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 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 `maxpswid` or the height exceeds `maxpsht` 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 postion; 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.

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 circle radius. 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*

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 one-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 environment variables

A set of predefined variables establishes the default values of drawing parameters. 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

arcrad	0.25 arc radius
arrowht	0.1 length of arrowhead
arrowwid	0.05 width of arrowhead
boxht	0.5 box height
boxrad	0 radius of rounded box corners
boxwid	0.75 box width
circlerad	0.25 circle radius
dashwid	0.05 dash length for dashed lines

ellipseht	0.5 ellipse height
ellipsewid	0.75 ellipse width
lineht	0.5 height of vertical lines
linewid	0.5 length of horizontal lines
movewid	0.5 length of horizontal moves
movewid	0.5 length of horizontal moves
textht	0 assumed height of text
textoffset	2.5/72 text justification gap
textwid	0 assumed width of text

When a value is assigned to the variable `scale,` all of the above values are multiplied by the new value of `scale`. The drawing units are thereby changed but the default physical sizes of drawn objects remain unchanged since, on final output, dimensions are divided by the `scale` value. In addition, the following are unchanged by `scale`:

arrowhead	1 arrowhead shape
fillval	0.5 fill density
linethick	0.8 line thickness in points
maxpsht	11.5 maximum allowed diagram height
maxpswid	8.5 maximum allowed diagram width
scale	1 drawing unit scale factor

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 enumerated object within the current block.

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 expressed 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. 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,

       "Crunchy 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,

    "Crunchy 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, E. S., *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,* 2011, `http://ece.uwaterloo.ca/~aplevich/dpic/dpicdoc.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 Typesetting, 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,* 2011. File CMman.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>

7   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 value.

```
emptyterm: "<EMPTY>"

terminals:
    "<ERROR>"
    "<" "cw" "ccw"
    "(" ")" "*" "+" "-" "/" "%" ";" = "<NL>"
    "^" "!" "&&" "||"
    "," ":" "[" "]" "{" "}" "." "[]" "'" "'"
    "=" ":=" "+=" "-=" "*=" "/=" "%=" (* the order matters *)
    "&"
    "<float>" "<name>" "<Label>" "<LaTeX>"
    '"' = "<string>"
    "#"
    "$" = "<arg>"
    "<START>" "<END>"
    "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"
 "<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" "path"
 "<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:

```
METAGOAL = input "<EOF>"

input = "<EMPTY>"
      | input picture NL

picture = start NL elementlist "<END>"
        | start NL elementlist NL "<END>"

NL = "<NL>"
   | "<ERROR>"

start = "<START>"
      | "<START>" term
      | "<START>" term term
```

```
elementlist = "<EMPTY>"
            | element
            | elementlist NL element


term = factor
     | term "*" factor
     | term "/" factor
     | term "%" factor


element = namedobj
        | "<Label>" suffix ":" position
        | assignlist
        | "<directon>"
        | "<LaTeX>"
        | command
        | lbrace "{" elementlist optnl "}"
        | ifpart
        | elsehead "{" elementlist optnl "}"
        | for "}"
        | "command" stringexpr
        | "exec" stringexpr


lbrace = "<EMPTY>"


namedobj = object
         | "<Label>" suffix ":" object


suffix = "<EMPTY>"
       | "[" expression "]"


position = pair
         | expression "between" position "and" position
         | expression "of" "the" "way" "between" position "and" position
         | expression "<" position "," position "<compare>" shift


assignlist = assignment
           | assignlist "," assignment


command = "print" expression redirect
        | "print" position redirect
        | "print" stringexpr redirect
        | "reset"
        | "reset" resetlist
        | "sh" stringexpr
        | "copy" stringexpr
        | defhead optnl "{" "}"
        | "undefine" "<name>"
        | "undefine" "<Label>"
```

```
optnl = "<EMPTY>"
        | NL

ifpart = ifhead "{" elementlist optnl "}"

elsehead = ifpart "else"

for = forhead "{" elementlist optnl
    | for forincr "<endfor>" elementlist optnl

stringexpr = string
           | stringexpr "+" string

string = "<string>"
        | sprintf ")"

assignment = "<name>" suffix "=" expression
           | "<name>" suffix "=" assignment
           | "<envvar>" "=" expression
           | "<envvar>" "=" assignment

expression = term
           | "+" term
           | "-" term
           | expression "+" term
           | expression "-" term

ifhead = setlogic "if" logexpr "then"

setlogic = "<EMPTY>"

logexpr = logprod
        | logexpr "||" logprod

forhead = "for" assignlist "to" expression do

forincr = "<EMPTY>"

do = "do"
   | by expression "do"

by = "by"
   | "by" "*"

redirect = "<EMPTY>"
         | "<compare>" stringexpr
         | "<compare>" "<compare>" stringexpr
```

```
resetlist = "<envvar>"
          | resetlist "," "<envvar>"

defhead = "define" "<name>"
        | "define" "<Label>"

sprintf = "sprintf" "(" stringexpr
        | "sprintf" "(" stringexpr "," exprlist

exprlist = expression
         | expression "," exprlist

object = block
       | object "height" expression
       | object "width" expression
       | object "radius" expression
       | object "diameter" expression
       | object "thickness" expression
       | object "scaled" expression
       | object "<directon>" optexp
       | object "<linetype>" optexp
       | object "chop" optexp
       | object "filled" optexp
       | object "<arrowhd>" optexp
       | object "then"
       | object "cw"
       | object "ccw"
       | object "same"
       | object stringexpr
       | object "by" position
       | object "from" position
       | object "to" position
       | object "at" position
       | object "<textpos>"
       | object "<colrspec>" stringexpr
       | objectwith "at" position
       | objectwith "<corner>" "at" position
       | objectwith pair "at" position
       | "continue"

openblock = "<EMPTY>"

block = "<primitiv>" optexp
      | stringexpr
      | openblock "[" closeblock "]"
      | openblock "[]"
```

```
optexp = "<EMPTY>"
        | expression

closeblock = elementlist optnl

objectwith = object "with"
           | objectwith "." "<Label>" suffix
           | objectwith "." nth primobj

pair = expression "," expression
     | location shift

nth = ncount "th"
    | ncount "th" "last"
    | "last"

primobj = "<primitiv>"
        | "[]"
        | "<string>"
        | "[" "]"

shift = "<EMPTY>"
      | shift "+" location
      | shift "-" location

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

place = placename
      | placename "<corner>"
      | "<corner>" placename
      | "<corner>" "of" placename
      | "Here"

factor = primary
       | "!" primary
       | primary "^" factor

placename = "<Label>" suffix
          | nth primobj
          | placename "." "<Label>" suffix
          | placename "." nth primobj

ncount = "<float>"
       | "`" expression "'"
```

```
            | "{" expression "}"

    logprod = logval
            | logprod "&&" logval

    logval = lcompare
           | stringexpr "<" stringexpr
           | expression "<" expression

    lcompare = expression
             | stringexpr
             | lcompare "<compare>" expression
             | lcompare "<compare>" stringexpr

    primary = "<envvar>"
            | "<name>" suffix
            | "<float>"
            | "(" logexpr ")"
            | location ".x"
            | location ".y"
            | placename "<param>"
            | "rand" "(" ")"
            | "rand" "(" expression ")"
            | "<func1>" "(" expression ")"
            | "<func2>" "(" expression "," expression ")"
            | "(" assignlist ")"
```