## Diag: A Drawing Preprocessor for LaTeX

Benjamin R. Seyfarth

**Abstract**

Diag is a preprocessor for drawing diagrams for LaTeX documents. The user prepares a text file containing commands in the `diag` language which are processed by `diag` producing `fig` commands which are then processed by `transfig` producing commands in a variety of formats acceptable to LaTeX. The `diag` preprocessor interprets a language with graphics commands using infix expressions with user-defined variables. In addition it provides a macro facility for simplifying repetitive operations. The combination of `diag` and `transfig` provides a simple, portable method for producing diagrams within LaTeX.

### 1 Introduction

The TeX typesetting system by Donald Knuth [3] provides a method for producing high-quality typesetting on a wide variety of computer systems. TeX has been augmented by Leslie Lamport's LaTeX [5] to provide an easier interface for TeX users. TeX was designed for typesetting text and mathematical formulas and does a splendid job for both. However, TeX provides nearly no support for graphics. LaTeX provides a variety of macro packages which do allow the user to incorporate drawings in a LaTeX document, but none of these is particularly easy to use. Diag provides a convenient alternative for LaTeX drawings.

LaTeX incorporates a simple picture drawing environment which can be used to produce lines, boxes, circles and arrows. Unfortunately, this package is designed around a set of line drawing characters which can only be used to draw lines with a limited number of preset slopes. In addition the package is written as TeX macros which means that specifications of $x$ and $y$ values can become onerous if the user must use TeX macros to compute locations. Lamport suggests that LaTeX drawings be completely designed using an initial drawing on a piece of graph paper. This is feasible, but it does not provide easily-modifiable diagrams.

There are several macro packages which have been written for LaTeX to provide better graphics. EPIC [7] is an extension of the LaTeX picture environment which uses the LaTeX drawing commands as primitives to produces lines, grids and arcs. EEPIC [4] is an extension of EPIC which uses `tpic` specials to overcome the limitations inherent in the LaTeX picture drawing primitives. The PiCTeX [8] package overcomes most of the limitations of the other macro packages and can produce high-quality graphics. Unfortunately it and the other macro packages suffer from the inconvenience of doing arithmetic using TeX macros.

In contrast to these LaTeX macro packages is the PIC preprocessor [2] for the `troff` typesetting system [6]. PIC provides a separate language supporting variables, infix expressions, looping and macros. This language allows a user to describe a diagram very simply using variables to define the $x$ and $y$ coordinates for graphics objects. This makes it easy to position objects relative to other objects which makes diagrams easier to modify.

There is a version of the PIC preprocessor called `tpic`, which has been altered to output TeX `\special` commands. These specials are then interpreted by DVI drivers to do the actual drawing. Unfortunately `tpic` is a modification of PIC and can only be distributed to licensed PIC users.

A completely different alternative for producing drawings in LaTeX is to use an interactive drawing program such as `fig` or `xfig`. `fig` is a graphics editor originally written by Supoj Sutanthavibul at the University of Texas. `xfig` is a version of `fig` written for the X Windowing System by Brian Smith of the Lawrence Berkeley Laboratory and others. Both these programs output `fig` commands which can be translated using `transfig` into EPIC, EEPIC, PiCTeX, `tpic` and several other formats usable in LaTeX. This is a convenient proposition for people with graphics terminals, but graphics terminals are not always available. Another drawback to using interactive drawing programs is that they do not generally support a convenient language interface. A language interface would allow users to write special programs to output graphic commands when there are many drawings to create.

The `diag` preprocessor provides a language similar to the PIC language, although it is considerably simplified. It does support variables, infix expressions, relative positioning and macros similarly to PIC. It does not presently support loops or `if` statements, nor does it support as many relative positioning options as PIC. It was decided that loops and conditional statements would be most useful for graphing mathematical functions and the author suggests using the `GNUPLOT` program for plotting functions. The relative positioning options in `diag` are fewer than those in PIC, but sufficient for most uses. The `diag` language is designed to be easy to learn and is capable of producing high-quality graphics for LaTeX documents.

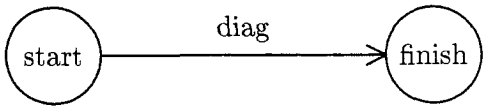


**Figure 1**: Getting There

## 2 Using diag

The input to diag is a text file containing diag commands. Let's suppose that we have a text file named "ex1.d" which contains the following:

```
Start: circle "start" at (1,1);
Finish: circle "finish" at (3,1);
arrow "diag" Start.e to Finish.w;
```

This file contains three diag commands. The first two draw circles and the third command draws an arrow from the start circle to the finish circle, placing the word "diag" above the arrow. The words before the colons of the first two commands are the names of the objects. The arrow is drawn from the easternmost point of the start circle to the westernmost point of the finish circle. We could modify the location of either circle and still connect the two circles using the same arrow command.

To convert the diag commands into PICTEX commands, we use the following command

```
diag < ex1.d | fig2dev -L pictex > ex1.tex
```

This will produce a file, "ex1.tex" containing commands which can be input into LATEX using

```
\begin{figure}
    \begin{center}
        \input{ex1}
    \end{center}
    \caption{Getting There}
\end{figure}
```

The resulting diagram is shown in Figure 1.

## 3 The Diag Coordinate System

The default coordinate system for diag uses measurements in inches. The origin of the coordinate system, $(0, 0)$, is defined to be the lower left corner of the diagram. From there increasing $x$ values refer to points to the right and increasing $y$ values refer to points up the page from the origin.

The default scaling can be altered by assigning a new value to the scale variable which is initially 1. Making scale larger will shrink your diagram, while making it larger will expand your diagram. It is possible to change scale in the middle of a diagram, but this is likely to cause confusion.

As graphics commands are executed, diag maintains a current drawing position. This position can be used as a default for most commands and it can be explicitly altered. The variables x and y refer to the current position and are available to the user.

## 4 The Diag Language

Parsing in diag is performed by an interpreter generated using the yacc parser generator. The interpreter consists of a lexical analyzer feeding the LALR(1) parser from yacc. The two work together to translate commands in the diag language into equivalent fig commands.

### 4.1 Diag Lexical Conventions

The lexical analyzer expands macros, ignores comments and groups input characters into lexical items. The macro expansion facility will be defined later. The diag lexical items are *identifiers*, *numbers*, *strings* and the following special characters:

```
+ - / * ( ) > .  ; :  ? and =
```

An identifier is a letter followed by any number of letters or digits. Upper and lower case letters are permitted and denote different identifiers. Several unexpected identifiers are keywords in diag and will most likely cause syntax errors if they are used as variable names. These include:

```
e n s w ne nw sw se
```

A number in diag must start with a digit and may have any number of digits afterwards with at most one decimal point. Any fraction less than 1.0 must have a leading zero as in "0.5". A number can be preceded by a minus sign.

A string is defined as in the C programming language to be anything between a pair of quote symbols as in "string". It is not possible to place a quote symbol in a diag string.

The special characters are used to form arithmetic expressions and for a handful of special purposes detailed below.

Comments in diag are identified by either a # or % and extend from that character to the end of the line. This allows comments to either stand alone or to be placed on the end of a command.

An identifier in diag is either a keyword or a variable name. A variable becomes defined either by an assignment statement or by a graphics command preceded by an identifier naming a graphics object. In either case the variable name is the first element of the command. Most commands start with one of the command keywords defined below and every diag command is terminated with a semicolon.

### 4.2 Diag Statements

A diagram is defined to be one or more statements in the `diag` language. Using Backus-Naur Form (BNF), we have:

*diagram* → *diagram statement ;*
→ *statement ;*

There are several types of statements in `diag`:

*statement* → *assignment*
→ *drawbox*
→ *drawcircle*
→ *drawellipse*
→ *drawline*
→ *drawarrow*
→ *drawtext*
→ *drawarc*
→ *drawcurve*
→ *gotostatement*

#### 4.2.1 Assignment Statement

An assignment statement is defined to be a variable name followed by an equals sign and an arithmetic expression. The variable will be created if it does not already exist. The BNF for assignment statements and expressions is

*assignment* → *IDENTIFIER = expr*
*expr* → *IDENTIFIER*
→ *NUMBER*
→ *expr + expr*
→ *expr * expr*
→ *expr - expr*
→ *expr / expr*
→ *- expr*
→ *( expr )*
→ *IDENTIFIER . xory*
→ *IDENTIFIER . pos . xory*
*xory* → *x* | *y*

Precedence for arithmetic expressions follows the normal pattern with multiplication and division having higher precedence than addition and subtraction. Parenthetical expressions are evaluated first.

Boxes, circles and ellipses may be named within `diag`. This is done by preceding the command to draw an object by a variable name and a colon. Afterwards the object's name can be used to specify a position. The $x$ and $y$ coordinates of an object can be used in an expression by adding either "`.x`" or "`.y`" after the variable name.

There are eight compass point positions defined for every named object. These can be used to specify positions in a diagram. This can be quite convenient compared to computing a position like the northeast point of an ellipse.

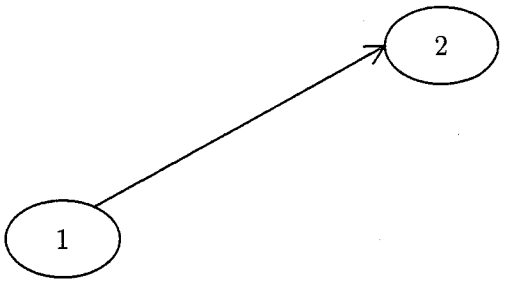


**Figure 2**: Ellipses and arrow

Object names and corner points are special cases of point expressions in `diag`. A point expression can also be specified as two arithmetic expressions in parentheses. Here is the syntax for point expressions:

*ptexpr* → *( expr , expr )*
→ *ptexpr + ptexpr*
→ *ptexpr - ptexpr*
→ *IDENTIFIER*
→ *IDENTIFIER . pos*
→ *IDENTIFIER . ?*
*pos* → *n* | *s* | *e* | *w* | *ne* | *nw* | *se* | *sw*

The following code draws two ellipses and connects them with an arrow:

```
x1 = 1;
x2 = x1 + 2;
e1: ellipse "1" at (x1,1);
e2: ellipse "2" at (x2,e1.y+1);
arrow from e1.ne to e2.w;
```

The diagram is in Figure 2. The second ellipse is placed two inches to the right and one inch higher than the first ellipse. The arrow is drawn from the northeast point of the first ellipse to the west compass point of the second ellipse.

Sometimes the eight compass points are not exactly the right points. Suppose we wish to draw an ellipse with four circles beneath and draw arrows to each circle. This is indicated with an object name followed by "`.?`" to indicate that `diag` should calculate a boundary point of the object for the connect line or arrow. This is shown in Figure 3. Here is the code required:

```
scale = 1.5;
e1: ellipse "Start" at (2.5,2);
c1: circle "1" at (1,1);
c2: circle "2" at (2,1);
c3: circle "3" at (3,1);
c4: circle "4" at (4,1);
arrow from e1.? to c1.?;
arrow from e1.? to c2.?;
arrow from e1.? to c3.?;
```

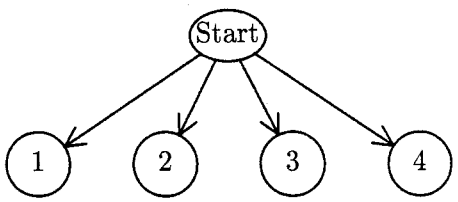


**Figure 3**: Ellipse and arrows to circles

```
arrow from e1.? to c4.?;
```

### 4.2.2 Drawing Boxes

A `diag` box is a rectangle which has sides parallel with the $x$ and $y$ axes. A box may include a text string placed at its center. A box may be described by giving two corner points or by specifying its height, width and center point. A box specified by corner points must specify two corners which must be opposite corners for the box. The syntax allows the keywords `from` and `to` to be optional.

A `box` command without two corner points specifies a box by height, width and center point. The predefined variables `boxht` and `boxwid` provide convenient defaults and the current point is used for the center if it is omitted.

| | | |
|---|---|---|
| *drawbox* | → | *objectname box boxopts* |
| *boxopts* | → | $\epsilon$ |
| | → | *boxopts from ptexpr to ptexpr* |
| | → | *boxopts label* |
| | → | *boxopts height expr* |
| | → | *boxopts width expr* |
| | → | *boxopts invisible* |
| | → | *boxopts at ptexpr* |
| *objectname* | → | $\epsilon$ \| *IDENTIFIER :* |
| *label* | → | *STRING* |
| *from* | → | $\epsilon$ \| *from* |
| *to* | → | $\epsilon$ \| *to* |
| *at* | → | $\epsilon$ \| *at* |

Notice that a box can be `invisible`. This can be useful for drawing lines between words in a parse tree. Consider the following code and its diagram in Figure 4:

```
scale = 1.5;
boxht = 0.3;
b1: box invisible width 1.5 "sentence"
    at (2,2);
b2: box invisible width 1 "subject"
    at (1,1);
b3: box invisible width 1 "verb"
    at (2,1);
b4: box invisible width 1 "object"
    at (3,1);
line from b1.? to b2.?;
line from b1.? to b3.?;
line from b1.? to b4.?;
```

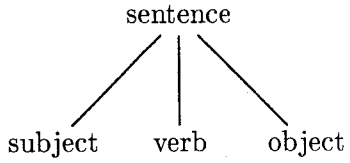


**Figure 4**: Simple sentence

### 4.2.3 Drawing Circles and Ellipses

A `diag` circle is defined by its radius and center point. The default for the radius is provided by the variable "`circlerad`", while the center point defaults to the current drawing position. An ellipse is defined similarly except that an ellipse has a major axis and a minor axis rather than a radius. In `diag` the major axis always refers to the $x$ axis of an ellipse and the minor axis refers to the $y$ axis. The defaults for the ellipse axes are the variables "`majoraxis`" and "`minoraxis`".

| | | |
|---|---|---|
| *drawcircle* | → | *objectname circle circleopts* |
| *circleopts* | → | $\epsilon$ |
| | → | *circleopts label* |
| | → | *circleopts radius expr* |
| | → | *circleopts at ptexpr* |
| *drawellipse* | → | *objectname ellipse ellipseopts* |
| *ellipseopts* | → | $\epsilon$ |
| | → | *ellipseopts label* |
| | → | *ellipseopts major expr* |
| | → | *ellipseopts minor expr* |
| | → | *ellipseopts at ptexpr* |

### 4.2.4 Drawing Lines and Arrows

There are two basic ways to draw lines and arrows. First you can specify the start and end points for the line. The keyword `from` is optional, but the keyword `to` is required if the end point is specified. The second way to specify a line is to specify the end point, a direction and a line length. The only possible directions are up, down, left and right. The start point defaults to the current point and the direction defaults to right.

Text may be drawn at the midpoint of the line or arrow. Following a text string you may optionally specify where to place the text relative to the midpoint of the line. The default is to place the text slightly above the midpoint.

Here is the full syntax for line and arrow drawing:

*drawline* → *line lineopts*
*lineopts* → $\epsilon$
→ *lineopts label where*
→ *lineopts from ptexpr*
→ *lineopts to ptexpr*
→ *lineopts direction expr*
*direction* → *up* | *down* | *left* | *right*
*drawarrow* → *arrow lineopts*
*where* → $\epsilon$ | *above* | *below*
→ *left* | *right*

### 4.2.5 Placing Text at Arbitrary Positions

It is possible to place a text string at any arbitrary position of a diagram by entering the string followed by the position for the string. The keyword `at` is optional and the position defaults to the current position.

*drawtext* → *STRING at ptexpr*
→ *STRING*

### 4.2.6 Drawing Circular Arcs and Curves

There are two types of curves supported by `diag`. They are both circular arcs, but they are given separate commands for simplicity. The first type of circular arc is drawn with the `arc` command. It is always a 90 degree arc in one of the four quadrants. Such an arc is specified by starting point, quadrant and direction. The starting point defaults to the current point. The quadrant is specified as `ur`, `ul`, `ll` or `lr` to indicate upper-right, upper-left, lower-left or lower-right. The direction is either `cw` or `ccw` to indicate clockwise or counter-clockwise.

The second type of arc is specified by start point, end point and curvature. The curvature is specified by the keyword `bend` followed by a number or an expression. The curvature defaults to value of the variable `curvature`. In general, the curvature should be between 0 and 1, but not very close to either.

The `bend` keyword is illustrated in Figure 5. In this figure the `bend` was set to 0.2. This means that the height of the curve, $h$, is 0.2 times the length of the chord $c$. The code to produce Figure 5 is:

```
scale = 2.0;
curve cw bend 0.2 (1,1) to (5,1);
linestyle = dashed;
line "c" below from (1,1) to (5,1);
line "h" left from (3,1) to (3,1.8);
```

It is also possible to place an arrow head on the end point of a curve. This is done by placing a greater than symbol in the command. The full syntax for drawing arcs and curves is:

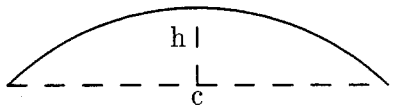


**Figure 5**: Curvature definition

*drawarc* → *arc arcopts*
*arcopts* → $\epsilon$
→ *arcopts quadrant*
→ *arcopts arcdir*
→ *arcopts radius expr*
→ *arcopts ptexpr*
*drawcurve* → *curve curveopts*
*curveopts* → $\epsilon$
→ *curveopts label where*
→ *curveopts* >
→ *curveopts arcdir*
→ *curveopts bend expr*
→ *curveopts from ptexpr*
→ *curveopts to ptexpr*
*arcdir* → *cw* | *ccw*
*quadrant* → *ul* | *ur* | *ll* | *lr*

### 4.2.7 Changing the Current Drawing Position

There is a `goto` command to explicitly change the current drawing position of `diag`. It consists of the keyword `goto` followed by an arbitrary point expression.

*gotostatement* → *goto ptexpr*

## 4.3 Predefined Variables

There are a number of variables created by `diag` which can be changed to control things like line thickness and arrow head length. These variables are different from user-created variables only in the sense that they exist when `diag` starts and `diag` uses their values for various purposes.

### 4.3.1 arcrad and circlerad

The `arc` command will draw a 90 degree arc of a certain radius. If the radius is not specified, it will default to the value of `arcrad`. Similarly the circle command defaults to a radius of `circlerad`. Both these variables are measurements in inches unless `scale` has been changed. The initial values for `arcrad` and `circlerad` are each 0.25 inches.

### 4.3.2 boxht and boxwid

The `box` command can be used to draw a box with a center at a certain position. In that usage the user can specify the height and width of the box, or allow `diag` to use `boxht` and `boxwid` as default values.

These are measurements in inches by default. The initial value for `boxht` is 0.5 inches and the initial value for `boxwid` is 0.75 inches.

#### 4.3.3 curvature

The `curve` command allows the user to specify the curvature using the `bend` keyword. If `bend` is not specified, the value of `curvature` will be used instead. The initial value for `curvature` is 0.2.

#### 4.3.4 dashlength

If the linestyle has been selected as dashed or dotted, then the variable `dashlength` can be set to control the length of dashes or the spacing of dots. This is a measurement in inches by default. The initial value of `dashlength` is 0.1 inches.

#### 4.3.5 head

The length of the head of an arrow can be controlled by the `head` variable. This is a measurement in inches by default. The initial value of `head` is 0.1 inches.

#### 4.3.6 linestyle

The variable `linestyle` can be used to change the line style from `solid` to `dashed` or `dotted`. The initial value of `linestyle` is `solid`. The variables `solid`, `dashed`, and `dotted` have the values 0, 1 and 2 matching their `fig` values.

#### 4.3.7 linethickness

This variable controls the thickness of lines in pixels. Its initial value is 5 pixels.

#### 4.3.8 majoraxis and minoraxis

These variables are used as defaults by the `ellipse` command. The $x$ axis is always considered the major axis and the the $y$ axis the minor axis. These variables represent inches by default. The initial value for `majoraxis` is 0.3 inches and the initial value for `minoraxis` is 0.2 inches.

#### 4.3.9 pi

This variable has the value 3.14159 and should not be changed.

#### 4.3.10 scale

The default scaling of coordinates in `diag` is in inches. This can be overridden by assigning a new value to `scale`. Coordinates in `diag` are divided by `scale` before translating into dot positions on the page. This means that making `scale` greater than 1.0 will shrink the diagram.

## 5 Defining and Using Macros

The macro facility of `diag` is implemented as a text replacement algorithm by the lexical analyzer. A macro is defined by the keyword `define`, the name of the macro, and its replacement text. The replacement text is identified by starting and ending it with a special symbol such as `%`.

A macro invocation is either the macro name followed by a semicolon or the name followed by parameters in parentheses. These parameters are positional parameters separated by commas and are referred to within the macro's replacement text as `$1`, `$2`, ..., `$9`. There can be up to nine positional parameters.

A sample macro definition to define a macro to draw three boxes centered at a given position would be

```
define ThreeBoxes #
   box at ($1-boxwid,$2);
   box at ($1,$2);
   box at ($1+boxwid,$2);
#
```

This macro be used to create a three by three arrangement of boxes using

```
p = 2;
ThreeBoxes ( 3, p );
ThreeBoxes ( 3, p-boxheight );
ThreeBoxes ( 3, p+boxheight );
```

## 6 A Larger Diagram

In this section a diagram is shown of an array of pointers to structures containing pointers and names. This is a reasonable example for illustrating macros. The code for this example is split up into several sections along with some explanation. The diagram is shown in Figure 6.

First there is a macro to draw a `NULL` pointer to the right of some of the array elements. This macro uses the variable `bw` to determine how long the constituent lines should be.

```
define Null %
   line right bw * 1.0;
   line down bw * 0.15;
   X = x;
   Y = y;
   line (X-0.2*bw,Y) to (X+0.2*bw,Y);
   line (X-0.125*bw,Y-0.05*bw)
     to (X+0.125*bw,Y-0.05*bw);
   line (X-0.05*bw,Y-0.1*bw)
     to (X+0.05*bw,Y-0.1*bw);
%
```

Next there is a macro to draw a box and then move down `bw` inches. This macro draws the box

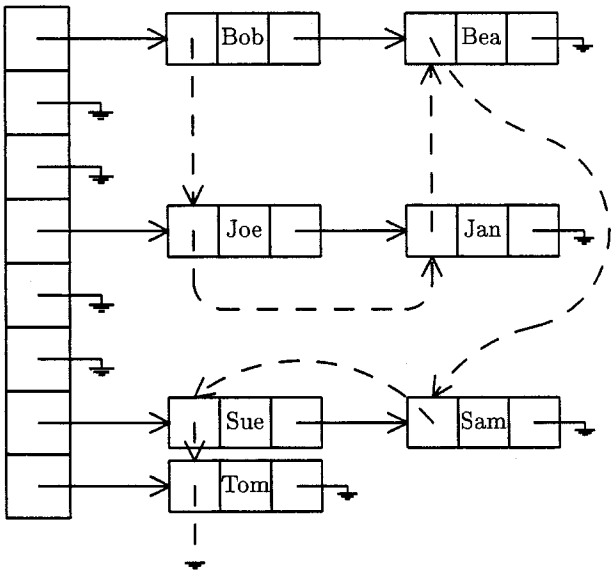


**Figure 6**: Symbol Table with an Auxiliary Linked List

at (x1,y1) and then modifies y1 to prepare for the next vertical box.

```
define Vbox %
   box (x1,y1) to (x1+bw,y1-bw);
   y1 = y1 - bw;
   goto (x1+bw/2,y1+bw/2);
%
```

Similarly there is a macro to draw a horizontal box at (x1,y1) and move to the right bw inches. This macro is used to draw structures containing three boxes. In the struct macro each Hbox is given a name to make it easy to connect the components later. The first parameter of struct is the prefix for the names of the boxes. The middle box is given that name and the others are given that name with an added l or r.

```
define Hbox %
   box $1 (x1,y1) to (x1+bw,y1-bw);
   x1 = x1 + bw;
   goto (x1+bw/2,y1+bw/2);
%

define struct %
   x1 = $2 - bw * 1.5;
   y1 = $3 + bw * 0.5;
   $1l: Hbox;
   $1: Hbox ( $4 );
   $1r: Hbox;
%
```

Now begins the first non-macro code. First the variables x1, y1 and bw are initialized and then the array of pointers to records is drawn along with several null pointers.

```
scale = 1.8;
x1 = 1;
y1 = 4;
bw = 0.6;
#
# Place the array of record pointers
# on the left.
#
A: Vbox;
B: Vbox;
Null;
C: Vbox;
Null;
D: Vbox;
E: Vbox;
Null;
F: Vbox;
Null;
G: Vbox;
H: Vbox;
```

Next the variable bw is shrunk to make slightly smaller boxes and then the structures are drawn. After the structures on a row are drawn, connecting arrows and null pointers are drawn.

```
bw = bw * 0.8;
#
# Add the 'B' records
#
struct(s1,3.25,A.y,"{\small Bob}");
struct(s2,5.5,A.y,"{\small Bea}");
arrow from A to s1l.w;
arrow from s1r to s2l.w;
goto s2r;
Null;
#
# Add the 'J' records
#
struct(s3,3.25,D.y,"{\small Joe}");
struct(s4,5.5,D.y,"{\small Jan}");
arrow from D to s3l.w;
arrow from s3r to s4l.w;
goto s4r;
Null;
#
# Add the 'S' records
#
struct(s5,3.25,G.y,"{\small Sue}");
struct(s6,5.5,G.y,"{\small Sam}");
arrow from G to s5l.w;
arrow from s5r to s6l.w;
goto s6r;
Null;
```

```
#
# Add the 'T' record
#
struct(s7,3.25,H.y,"{\small Tom}");
arrow from H to s71.w;
goto s7r;
Null;
```

Finally we add a collection of dashed arrows and curves to indicate another linked list comprising the same set of records.

```
#
# Add auxiliary linked list pointers
#
linestyle = dashed;
arrow from s11 to s31.n;
goto s31;
line down bw;
arc ll ccw;
line right s4.x-s3.x-2*arcrad;
arc lr ccw;
arrow to s41.s;
arrow s41 to s21.s;
ya = (s2.y+s4.y) / 2;
yb = (s4.y+s6.y) / 2;
curve ccw bend 0.1 from s21
   to (s4r.x,ya);
curve cw bend 0.4 to (s4r.x,yb);
curve > ccw bend 0.1 to s61.n;
curve > ccw bend 0.2 from s61 to s51.n;
arrow from s51 to s71.n;
goto s71;
line down bw*1.5;
X = x;
Y = y;
linestyle = solid;
line (X-0.2*bw,Y) to (X+0.2*bw,Y);
line (X-0.125*bw,Y-0.05*bw)
  to (X+0.125*bw,Y-0.05*bw);
line (X-0.05*bw,Y-0.1*bw)
  to (X+0.05*bw,Y-0.1*bw);
```

## 7 Possible Additions to the Language

The most obvious features missing from the language are conditional statements and loops. These would clearly be useful, but are not critical for the anticipated uses of `diag`.

If `diag` needs conditional statements and loops at some future date, it is likely that procedures would also be added. The current implementation uses macros which look like procedure calls, but they use global variables. There is no such thing as a local variable and using macros which manipulate variables is a hazard. Procedures would eliminate this problem.

Another possible improvement to `diag` would be to use the ".?" positioning operator with the `curve` command. This would be relatively easy to implement and could be useful for some diagrams.

It would be useful to draw and fill polygons. This is supported by `transfig` and would be easy to add to `diag`. Another feature supported by `transfig` which could be added is a spline drawing command.

There are many more features which could be added to the `diag` language. The author selected the most basic commands for the first version of `diag`. It is anticipated that the language will grow as needs arise.

## References


[1] Micah Beck, *TransFig: Portable Figures for TEX*, Cornell University Dept. of Computer Science Technical Report #89-967, February 1989.

[2] Brian W. Kernighan, *PIC - A Graphics Language for Typesetting*, Bell Laboratories Computing Science Technical Report 85, March 1982.

[3] Donald E. Knuth, *The TEXbook*, Addison-Wesley, 1986.

[4] Conrad Kwok, *EEPIC: Extensions to* EPIC *and LATEX Picture Environment*, Software documentation, University of California, Davis, Dept. of Computer Science, July 1988.

[5] Leslie Lamport, *LATEX: A Document Preparation System*, Reading, Mass.: Addison-Wesley, 1986.

[6] Joseph F. Osanna, *NROFF/TROFF User's Manual*, Bell Laboratories Computing Science Technical Report 54, October 11, 1976.

[7] Sunil Podar, *Enhancements to the Picture Environment of LATEX*, State University of New York at Stony Brook, Dept. of Computer Science, Technical Report #86-17, July 1986.

[8] Michael Wichura, *The PICTEX Manual*, University of Chicago, November 1986.



◇ Benjamin R. Seyfarth
Department of Computer Science and Statistics
The University of Southern Mississippi
Southern Station
Box 5106
Hattiesburg, Mississippi 39406-5106