The Expressg MetaPost package for drawing box-line-annotation diagrams*

Peter Wilson
Catholic University of America
Now at `peter.r.wilson@boeing.com`

(First published 1996/05/09)
2004/03/17

Abstract

The expressg MetaPost package provides facilities to assist in drawing diagrams that consist of boxes, lines, and annotations. Particular support is provided for creating express-g diagrams

Contents

---

[*] This file (`expressg.dtx`) has version number v1.61, last revised 2004/03/17.

List of Figures

1 Introduction

EXPRESS-G is an ISO international standard graphical information modeling language [ISO94, SW94], and is a subset of the EXPRESS lexical object-flavoured information modeling language. Like most graphical modeling languages, EXPRESS-G diagrams consist of boxes, lines, and annotations associated with the boxes and the lines. As a colleague of mine once put it, these kinds of graphical languages are BLA (boxes, lines, annotations) with respect to lexical languages.

MetaPost [Hob92] is a powerful lexical language for producing Encapsulated PostScript[1] diagrams, and is based on the METAFONT [Knu86] language for creating fonts.

---

[1] PostScript is a registered trademark of Adobe Systems Incorporated.

The expressg package provides extensions to the base METAPOST language to facilitate the creation of EXPRESS-G diagrams. The package may also be used as is, or certainly by extension, for the creation of other BLA diagrams.

METAPOST is typically used to generate pictures for inclusion in LaTeX documents [Lam94]. By default it uses TeX for typesetting any text; this behaviour can be modified so that LaTeX is used instead. If standard PostScript fonts (like Times) are used, then LaTeX typesetting may be avoided altogether. Running METAPOST and methods of including METAPOST diagrams in (pdf)LaTeX documents are described in [Wil99].

The expressg package has been some years in the making. For occasional diagrams I got by with using either the simple LaTeX picture commands, or general GUI drawing packages. As time went on and the number and complexity of the diagrams increased it became, to me at least, more important to use a robust and powerful drawing language than fancy GUI packages. This was especially important if a diagram had to be changed at a later date, as the language denoted exactly how the drawing had been produced. With the GUI tools all that you had was the picture and whatever graphics file format the tool used — there was no user-sensible record of how any result was achieved. Also, generally speaking I find it quicker to use METAPOST than a GUI tool as too much time has to be spent using a mouse or a cursor to reposition and rescale the drawing elements.

Section 2 describes the facilities provided by the package. Several worked example diagrams are presented in Section 3, together with information on how to run METAPOST. Commented code for the package macros[2] is in Section 4.

It is assumed that the reader has some understanding of METAPOST, although the example in Section 3 does provide an *aide-memoire* for most of the METAPOST constructs that are likely to be used in creating BLA diagrams. The definitive METAPOST manual is [Hob92] but both [Hoe98] and [GRM97] give some interesting examples of how to use it; in fact the front cover of Hoenig's book is a METAPOST picture. Similarly, some familiarity with BLA diagramming would be of assistance.

I suggest that on first reading it may be sensible to skim Section 2 first and then peruse the example in Section 3 more closely. After this go back and re-read Section 2 as it then should make more sense.

This manual is typeset according to the conventions of the LaTeX DOCSTRIP utility which enables the automatic extraction of the LaTeX macro source files [GMS94].

2 The expressg package

2.1 Internal variables

The package includes several internal variables that are initialised to commonly appropriate values. This section describes the principal internal variables. Short

---

[2] If requested, I may be prepared to consider providing additional macros, but no promises.

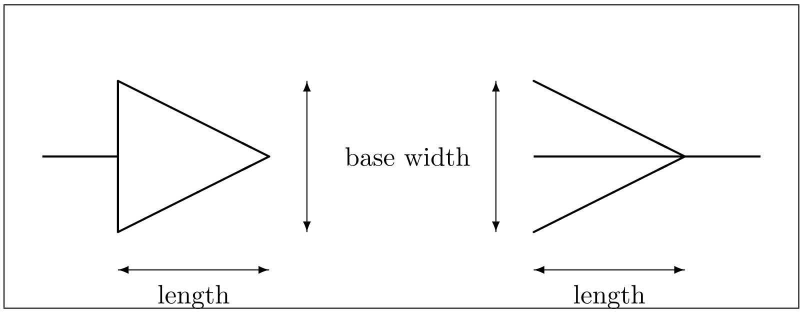

Figure 1: Length parameters for arrowhead and fanin line end styles

descriptions of the others, which are likely to be used in diagrams only very occasionally, can be found in Section 4.

The values of internal variables in MetaPost can only be changed via assignment. For example:

```
internal_variable := expression;
```

`u` `maxx` `maxy` `u` is the value of the unit of length for a diagram; it is initialised to 1mm. All lengths, except for line thicknesses, are defined in terms of `u`. `maxx` and `maxy` are the maximum x and y coordinate values for a diagram. These are initialised to the maximum size for a diagram within an ISO standard document (i.e., `maxx:=159.5u` and `maxy:=210u`).

The next set of variables control the thickness of lines and the dimensions of line end styles.

`normalpen` Lines with normal linethickness are drawn with `normalpen`.

`thickpen` Thick lines are drawn with `thickpen`.

`thinpenpen` Thin lines are drawn with `thinpen`.

`dotspen` Dotted lines are drawn with `dotspen`.

`dotpen` `dotpen` is for drawing the black dot line end style.

`dotdiam` `dotdiam` is the diameter of an open circle or dot line end style.

`smoothrad` `smoothrad` is the radius of the circular arc used to round the join between two straight lines.

`drumlid` `drumlid` is the ratio of the minor to major diameter of the ellipse forming the top of the drum box; the larger the value the more the apparent viewpoint shifts to above the drum. The default value is 0.2.

`gal` `gab` An arrowhead is defined by the height of the triangle and the base of the triangle forming the arrowhead, as shown in Figure 1. `gal` and `gab` are the length and base width of arrows for arrowed line end styles.

`gfl` `gfb` Like an arrowhead, a fanin is defined by the height and the base of the triangular fan shape, as shown in Figure 1. `gfl` and `gfb` are the length and base width of fans for fanin line end styles. They are initialised to the default values.

The next set of variables control certain geometric aspects of the various express-g boxes.

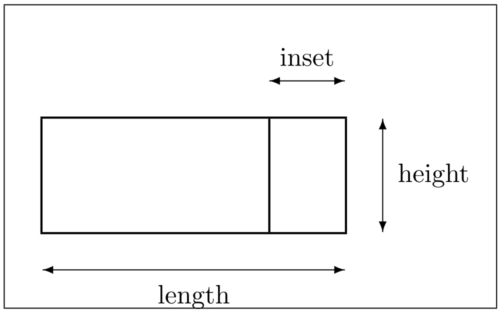

Figure 2: Parameters for (inset) boxes

`onelineh`

`onelineh` is the minimum height of a box that encloses a single line of text. This is initialised to `5u`.

`sdtbl` `sdtbh` `sdtbs`

The length (`sdtbl`), height (`sdtbh`) and line inset (`sdtbs`) for simple data type boxes (e.g., a BOOLEAN data type box), as illustrated in Figure 2. These are initialised as `22u`, `onelineh` and `2u` respectively.

`sdtebl` `sdtebh` `sdtebs`

The length (`sdtebl`), height (`sdtebh`) and line inset (`sdtebs`) for a simple data type EXPRESSION box. The value of `sdtebl` is `28u`, and the other values are the same as for the other simple data type boxes.

`sdtbgel` `sdtbgeh` `sdtbges`

The length (`sdtbgel`), height (`sdtbgeh`) and inset for GENERICENT data type boxes. The value of `sdtbgel` is `38u`, and the other values are the same as for the other simple data type boxes.

`pconl` `pconh`

The length (`pconl`) of an average numeric (proxy) page connector (e.g., for `(9,9 (9,9)))` and the height (`pconh`) of an average page connector, which is initialised for one line of text.

`enth` `typeh` `schemah`

The heights of an average entity (`enth`), type — including enumeration and select — (`typeh`), and schema (`schemah`) boxes. These are all initialised to cater for one line of text.

`ish` `isrh`

`ish` is the height of an interschema reference box with one line of text. `isrh` is the height of an interschema reference box with a rename, where there is one line of text each for the original and renamed names.

`eventh` `eventslope`

The height (`eventh`) of a one-lined event box, and the slope (`eventslope`) of the sides of an event box (e.g., 0.25 or 1/4).

The next set of variables control the amount of space surrounding text on EXPRESS-G diagrams.

`nextra`

`nextra` is the margin around names when put into entity, schema, interschema reference, page connector, and event boxes.

`niextra`

`niextra` is the margin around names when put into type boxes.

`ndextra`

`ndextra` is the margin around names when used on attribute lines.

2.2 Routines

The principal aspects of an EXPRESS-G diagram are lines and line ends of various styles, and boxes of several different kinds. Many routines are provided for

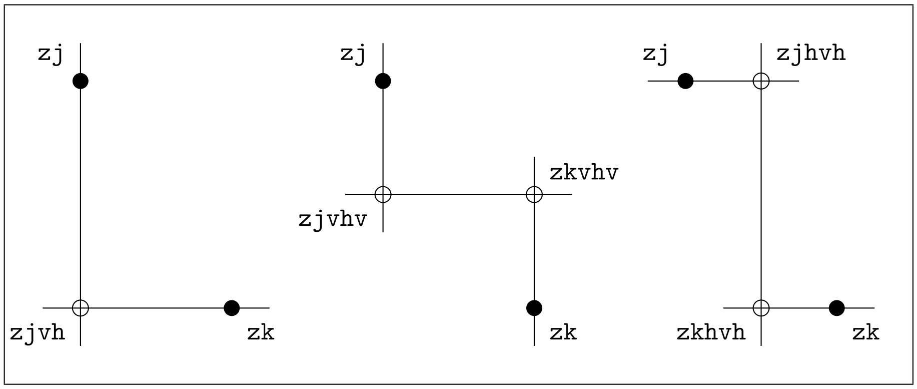

Figure 3: Results of the `VH` (left), `VhV` (middle) and `HvH` (right) routines

positioning and drawing these on a diagram.

The following sections describe most of the routines. Short descriptions of the others, which are effectively only used as internal support routines, can be found in Section 4.

A note on naming conventions used in describing the routines:

- As usual with MetaPost, a name starting with `z` is a point, and names staring with `x` or `y` are x- and y-coordinates respectively.
- The letters `i` through `n` denote (point, coordinate) suffixes. Suffixes are usually a number, such as `37`, but may also have alphanumeric characters after the number (e.g., `37C2D`).

2.2.1 Utility routines

Several utility routines are provided defining some shortcuts and to ease some of the calculations that may be required when producing a diagram.

`VH` `VH(j, k)` calculates the intersection point, `zjvh`, between the vertical line through the point `zj` and the horizontal line through the point `zk`, as shown in Figure 3.

`VhV` `VhV(j, k)` calculates the point `zjvhv` on the vertical line through the point `zj` and the point `zkvhv` on the vertical line through the point `zk`, such that the line `zjvhv--zkvhv` is horizontal and centered vertically between the two given points, as shown in Figure 3.

`HvH` `HvH(j, k)` calculates the point `zjhvh` on the horizontal line through the point `zj` and the point `zkhvh` on the horizontal line through the point `zk`, such that the line `zjhvh--zkhvh` is vertical and centered horizontally between the two given points, as shown in Figure 3.

`VyV` `VyV(j, i, k)` is a generalisation of the `VhV` routine. It calculates the point `zjvyv` which is on the vertical line through `zj` and the horizontal line through `yi`,

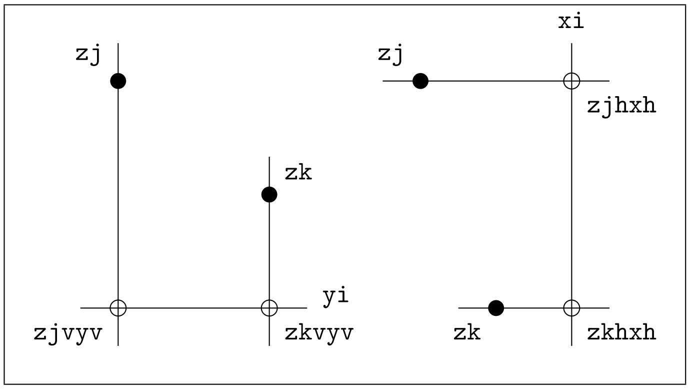

Figure 4: Results of the `VyV` (left) and `HxH` (right) routines

and the point `zkvyv` which is on the vertical line through `zk` and the horizontal line through `yi`. An example result is shown in Figure 4.

`HxH` `HxH(j, i, k)` is a generalisation of the `HvH` routine. It calculates the point `zjhxh` which is on the horizontal line through `zj` and the vertical line through `xi`, and the point `zkhxh` which is on the horizontal line through `zk` and the vertical line through `xi`. An example result is shown in Figure 4.

`namespace` `namespace(`⟨*name*⟩`)(`⟨*margin*⟩`)` calculates the length of the text string ⟨*name*⟩ plus the ⟨*margin*⟩ length. You may use it like:

```
c = 3/2namespace(btex name etex)(4u)
```

`dashes` `dashes` is shorthand for a dashed linestyle. Use it like: `draw ... dashes;`.

`dots` `dots` is shorthand for a dotted linestyle. Use it like: `draw ... dots;`.

`drawgrid` `drawgrid` draws a 16 by 21cm, 1cm spaced grid composed of thin dashed lines. Labels are typeset giving the distance of the lines from the bottom left corner of the grid in millimetres.

2.2.2 Path routines

The package provides some general path making routines.

`sharply` `sharply(zi, zj, ... zn)` will generate a piecewise linear path consisting of the straight lines from point `zi` to point `zj` . . . to point `zn`. It may be used like:

```
draw sharply(z1, z2, z3, z4);
```

to draw a line through the given points.

`smoothly` Like the `sharply` routine, `smoothly(zi, zj, ... zn)` will generate a piecewise linear path consisting of the straight lines from point `zi` to point `zj` . . . to point `zn`, except that the corners of the path at the interior points are rounded with an arc of radius `smoothrad`. It may be used like:

```
draw smoothly(z1, z2, z3, z4) dashes;
```

to draw a dashed line with rounded corners at the given points. Or if a dotted line is needed:

```
 pickup dotspen;
```

```
draw smoothly(z1, z2, z3, z4) dots;
pickup normalpen;
```

`~` Like the `&` operator, the binary `~` operator concatenates two paths replacing the sharp join with a circular arc of radius `smoothrad`.

2.2.3 Line drawing routines

Routines are provided for drawing a variety of lines with differing line ending styles. Line styles include dotted, dashed, and continuous, and line thicknesses may be normal or thick. Line end styles include open circles, black dots, open and closed (black) arrowheads, and a fanin style.

The final uppercase characters of a line drawing routine indicate the kind of line end style that will be used. The character `O` (uppercase O, not zero) is for an **O**pen circle, `D` is for a black **D**ot, `OA` is for an **O**pen **A**rrowhead, `CA` is for a **C**losed (black) **A**rrowhead, and `F` is for a **F**anin.

`drawO` The line ending drawing routines are described first. `drawO(i, j)` draws an open circle, diameter `dotdiam`, at the `zj` end of the vector from `zi` to `zj`.

`drawD` `drawD(i, j)` draws a black dot, diameter `dotdiam`, at the `zj` end of the vector from `zi` to `zj`.

`drawOA` `drawOA(i, j)` draws an open arrowhead, length `gal` and base width `gab`, at the `zj` end of the vector from `zi` to `zj`.

`drawCA` `drawCA(i, j)` draws a closed (black) arrowhead, length `gal` and base width `gab`, at the `zj` end of the vector from `zi` to `zj`.

`drawF` `drawF(i, j)` draws a fanin, length `gfl` and base width `gfb`, at the `zj` end of the vector from `zi` to `zj`.

The line drawing routines are essentially prepackaged versions of different combinations of the path and line ending routines.

`drawdots` `drawdots(i, j)` draws the dotted line `zi--zj`.

`drawdotsthree` `drawdotsthree(i, j, k)` draws the dotted piecewise linear line `zi--zj--zk`.

`drawdotsfour` `drawdotsfour(i, j, k, l)` draws the dotted piecewise linear line `zi--zj--zk--zl`.

`drawdotsO` `drawdotsO(i, j)` draws the dotted line `zi--zj`, ending in an open circle, diameter `dotdiam`, at `zj`.

`drawdotsthreeO` `drawdotsthreeO(i, j, k)` draws the dotted piecewise linear line `zi--zj--zk`. ending in an open circle, diameter `dotdiam`, at `zk`.

`drawdotsfourO` `drawdotsfourO(i, j, k, l)` draws the dotted piecewise linear line `zi--zj--zk--zl`, ending in an open circle, diameter `dotdiam`, at `zl`.

`drawdotsOO` `drawdotsOO(i, j)` draws the dotted line `zi--zj`, ending in open circles, diameter `dotdiam`, at `zi` and `zj`.

`drawdash` `drawdash(i, j)` draws the dashed line `zi--zj`.

`drawdashthree` `drawdashthree(i, j, k)` draws the dashed piecewise linear line `zi--zj--zk`.

`drawdashfour` `drawdashfour(i, j, k, l)` draws the dashed piecewise linear line `zi--zj--zk--zl`.

`drawdashO` `drawdashO(i, j)` draws the dashed line `zi--zj`, ending in an open circle, diameter `dotdiam`, at `zj`.

`drawdashthreeO` `drawdashthreeO(i, j, k)` draws the dashed piecewise linear line `zi--zj--zk`.

ending in an open circle, diameter `dotdiam`, at `zk`.

`drawdashfourO`	`drawdashfourO(i, j, k, l)` draws the dashed piecewise linear line `zi--zj--zk--zl`. ending in an open circle, diameter `dotdiam`, at `zl`.
`drawdashOO`	`drawdashOO(i, j)` draws the dashed line `zi--zj--zk--zl`. ending in open circles, diameter `dotdiam`, at `zi` and `zj`.
`drawnormal`	`drawnormal(i, j)` draws the normal thickness line `zi--zj`.
`drawnormalthree`	`drawnormalthree(i, j, k)` draws the normal thickness piecewise linear line `zi--zj--zk`.
`drawnormalfour`	`drawnormalfour(i, j, k, l)` draws the normal thickness piecewise linear line `zi--zj--zk--zl`.
`drawnormalO`	`drawnormalO(i, j)` draws the normal thickness line `zi--zj`, ending in an open circle, diameter `dotdiam`, at `zj`.
`drawnormalthreeO`	`drawnormalthreeO(i, j, k)` draws the normal thickness piecewise linear line `zi--zj--zk`, ending in an open circle, diameter `dotdiam`, at `zk`.
`drawnormalfourO`	`drawnormalfourO(i, j, k, l)` draws the normal thickness piecewise linear line `zi--zj--zk--zl`, ending in an open circle, diameter `dotdiam`, at `zl`.
`drawnormalOO`	`drawnormalOO(i, j)` draws the normal thickness line `zi--zj`, ending in open circles, diameter `dotdiam`, at `zi` and `zj`.
`drawnormalD`	`drawnormalD(i, j)` draws the normal thickness line `zi--zj`, ending in a black dot, diameter `dotdiam`, at `zj`.
`drawnormalthreeD`	`drawnormalthreeD(i, j, k)` draws the normal thickness piecewise linear line `zi--zj--zk`, ending in a black dot, diameter `dotdiam`, at `zk`.
`drawnormalfourD`	`drawnormalfourD(i, j, k, l)` draws the normal thickness piecewise linear line `zi--zj--zk--zl`, ending in a black dot, diameter `dotdiam`, at `zl`.
`drawnormalDD`	`drawnormalDD(i, j)` draws the normal thickness line `zi--zj`, ending in black dots, diameter `dotdiam`, at `zi` and `zj`.
`drawnormalCA`	`drawnormalCA(i, j)` draws the normal thickness line `zi--zj`, ending in a closed (black) arrowhead, length `gal` and base width `gab`, at `zj`.
`drawnormalthreeCA`	`drawnormalthreeCA(i, j, k)` draws the normal thickness piecewise linear line `zi--zj--zk` ending in a closed (black) arrowhead, length `gal` and base width `gab`, at `zk`.
`drawnormalfourCA`	`drawnormalfourCA(i, j, k, l)` draws the normal thickness piecewise linear line `zi--zj--zk--zl` ending in a closed (black) arrowhead, length `gal` and base width `gab`, at `zl`.
`drawnormalOA`	`drawnormalOA(i, j)` draws the normal thickness line `zi--zj`, ending in an open arrowhead, length `gal` and base width `gab`, at `zj`.
`drawnormalF`	`drawnormalF(i, j)` draws the normal thickness line `zi--zj`, ending in a fanin, length `gfl` and base width `gfb`, at `zj`.
`drawnormalFO`	`drawnormalFO(i, j)` draws the normal thickness line `zi--zj`, ending in a fanin, length `gfl` and base width `gfb`, at `zi` and an open circle, diameter `dotdiam`, at `zj`.
`drawthick`	`drawthick(i, j)` draws the thick line `zi--zj`.
`drawthickO`	`drawthickO(i, j)` draws the thick line `zi--zj`, ending in an open circle, diameter `dotdiam`, at `zj`.
`drawthickOO`	`drawthickOO(i, j)` draws the thick line `zi--zj`, ending in open circles, diameter `dotdiam`, at `zi` and `zj`.

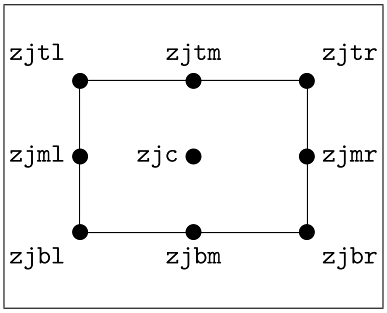

Figure 5: Calculated points on rectangular boxes (suffix is `j` and located at `zjbl=zj`)

`smooth` `smooth(i, j, k)` replaces the sharp corner on the line `zi--zj--zk` with an arc radius `smoothrad`.

`smoothtwo` `smoothtwo(i, j, k, l)` replaces the sharp corners on the line `zi--zj--zk--zl` with arcs radius `smoothrad`.

`smoothdash` `smoothdash(i, j, k)` replaces the sharp corner on the dashed line `zi--zj--zk` with an arc radius `smoothrad`.

`smoothdots` `smoothdots(i, j, k)` replaces the sharp corner on the dotted line `zi--zj--zk` with an arc radius `smoothrad`.

2.2.4 Box drawing routines

Routines are provided for drawing the different kinds of EXPRESS-G boxes.

The first argument to the box drawing routines is the suffix of the point that will be the left-hand bottom corner point of the box. The last argument, except where noted below, is the text that is to be placed at the center of the box. Intermediate arguments, if any, are the length and height of the box.

Every box routine calculates the four corner points of the box and the mid-points of each side of the box. For example, if the suffix is `j`, then as illustrated in figure 5, the corner points will be called `zjbl`, `zjbr`, `zjtr` and `zjtl` and the mid-points will be called `zjbm`, `zjmr`, `zjtm` and `zjml`. The center point for postioning the text is `zjc` and the center of the text is placed at this point.

The text center point is not necessarily the geometric center of the box. The geometric center is not calculated but lies at the intersection of the lines `zjml--zjmr` and `zjbm--zjtm`.

`drawSCHEMA` A SCHEMA box is rectangular with a normal perimeter and the box is divided into top and bottom halves by a horizontal line.

`drawSCHEMA(j, len, ht)(btex schema\_a etex)` draws a SCHEMA double rectangular box of length `len` and height `ht` with its left-hand bottom corner at `zj`. The string `schema_a` is placed at the text center of the box, which is central within the upper half.

`drawBINARY` A simple data type box is rectangular with a solid boundary and a solid interior vertical line near the right hand edge.

`drawBINARY(j)` draws a simple data type rectangular box of length `sdtbl` and height `sdtbh` with its left-hand bottom corner at `zj`. The string `BINARY` is placed at the text center of the box, which is central within the main part of the box.

`drawBOOLEAN`

`drawBOOLEAN(j)` draws a simple data type rectangular box of length `sdtbl` and height `sdtbh` with its left-hand bottom corner at `zj`. The string `BOOLEAN` is placed at the text center of the box, which is central within the main part of the box.

`drawCOMPLEX`

`drawCOMPLEX(j)` draws a simple data type rectangular box of length `sdtbl` and height `sdtbh` with its left-hand bottom corner at `zj`. The string `COMPLEX` is placed at the text center of the box, which is central within the main part of the box.

`drawEXPRESSION`

`drawEXPRESSION(j)` draws a simple data type rectangular box of length `sdtbel` and height `sdtbeh` with its left-hand bottom corner at `zj`. The string `EXPRESSION` is placed at the text center of the box, which is central within the main part of the box.

`drawGENERIC`

`drawGENERIC(j)` draws a simple data type rectangular box of length `sdtbl` and height `sdtbh` with its left-hand bottom corner at `zj`. The string `GENERIC` is placed at the text center of the box, which is central within the main part of the box.

`drawINTEGER`

`drawINTEGER(j)` draws a simple data type rectangular box of length `sdtbl` and height `sdtbh` with its left-hand bottom corner at `zj`. The string `INTEGER` is placed at the text center of the box, which is central within the main part of the box.

`drawLOGICAL`

`drawLOGICAL(j)` draws a simple data type rectangular box of length `sdtbl` and height `sdtbh` with its left-hand bottom corner at `zj`. The string `LOGICAL` is placed at the text center of the box, which is central within the main part of the box.

`drawNUMBER`

`drawNUMBER(j)` draws a simple data type rectangular box of length `sdtbl` and height `sdtbh` with its left-hand bottom corner at `zj`. The string `NUMBER` is placed at the text center of the box, which is central within the main part of the box.

`drawREAL`

`drawREAL(j)` draws a simple data type rectangular box of length `sdtbl` and height `sdtbh` with its left-hand bottom corner at `zj`. The string `REAL` is placed at the text center of the box, which is central within the main part of the box.

`drawSTRING`

`drawSTRING(j)` draws a simple data type rectangular box of length `sdtbl` and height `sdtbh` with its left-hand bottom corner at `zj`. The string `STRING` is placed at the text center of the box, which is central within the main part of the box.

`drawENUM`

An ENUMERATION box is a rectangular box with a dashed perimeter and an interior dashed vertical line near the right hand edge.

`drawENUM(j, len, ht)(btex an\_enum etex)` draws a rectangular dashed ENUMERATION type box of length `len` and height `ht` with its left-hand bottom corner at `zj`. The string `an_enum` is placed at the text center of the box, which is central within the main part of the box.

`drawSELECT`

A SELECT box is rectangular with a dashed boundary and an interior dashed vertical line near the left hand edge.

`drawSELECT(j, len, ht)(btex a\_select etex)` draws a rectangular dashed SELECT type box of length `len` and height `ht` with its left-hand bottom corner at `zj`. The string `a_select` is placed at the text center of the box, which is central within the main part of the box.

`drawTYPE` A TYPE box is rectangular with a dashed boundary.

`drawTYPE(j, len, ht)(btex a\_type etex)` draws a rectangular dashed TYPE type box of length `len` and height `ht` with its left-hand bottom corner at `zj`. The string `a_type` is placed at the text center of the box, which is central within the main part of the box.

`drawENT` An ENTITY box is a rectangular box with a solid boundary.

`drawENT(j, len, ht)(btex an\_entity etex)` draws a rectangular ENTITY type box of length `len` and height `ht` with its left-hand bottom corner at `zj`. The string `an_entity` is placed at the text center of the box, which is also the geometric center of the box.

`drawPREF` A page reference box is a rectangle with rounded ends and a solid boundary.

`drawPREF(j, len, ht)(btex page ref etex)` draws an oval page reference box of length `len` and height `ht` with its left-hand bottom corner at `zj`. The string `page ref` is placed at the text center of the box, which is also the geometric center of the box.

`drawISU` An interschema box is a rectangle surrounding a page reference like box. The rectangular box may have either a solid or a dashed boundary.

`drawISU(j, len, ht)(btex sch.ent etex)` draws a three-part rectangular interschema USE box of length `len` and height `ht` with its left-hand bottom corner at `zj`. The string `sch.ent` is placed at the text center of the box, which is also the geometric center of the box.

`drawISUR` `drawISUR(j, len, ht)(btex sch.ent etex)(btex newname etex)` draws a three-part rectangular interschema USE and RENAME box of length `len` and height `ht` with its left-hand bottom corner at `zj`. The string `sch.ent` is placed at the text center of the box, which is also the geometric center of the box. The string `newname` is placed at the geometric center of the bottom third of the box (this is the point `zjrnm`).

`drawISR` `drawISR(j, len, ht)(btex sch.ent etex)` draws a three-part rectangular dashed interschema REFERENCE box of length `len` and height `ht` with its left-hand bottom corner at `zj`. The string `sch.ent` is placed at the text center of the box, which is also the geometric center of the box.

`drawISRR` `drawISRR(j, len, ht)(btex sch.ent etex)(btex newname etex)` draws a three-part rectangular dashed interschema REFERENCE and RENAME box of length `len` and height `ht` with its left-hand bottom corner at `zj`. The string `sch.ent` is placed at the text center of the box, which is also the geometric center of the box. The string `newname` is placed at the geometric center of the bottom third of the box (this is the point `zjrnm`).

`drawLEVENT` An EVENT box is a rhomboid with a solid boundary.

`drawLEVENT(j, len, ht)(btex local\_event etex)` draws a rhomboidal Local EVENT box of length `len`, height `ht` and slope `eventslope` with its left-hand bottom corner at `zj`. The string `local_event` is placed at the text center of the box, which is also the geometric center of the box. Like the rectangular boxes, the calculated bottom and top middle points are vertically below and above the

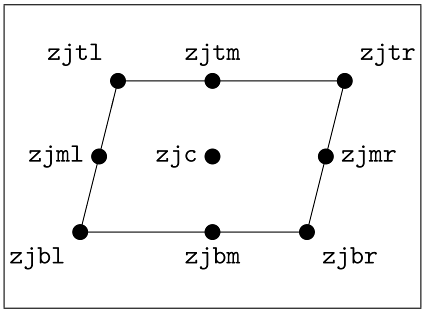

Figure 6: Calculated points on rhomboidal boxes (suffix is `j` and located at `zjbl=zj`)

geometric center, but this means that they are *not* midway beteen the bottom and top corner points. This is illustrated in Figure 6.

`drawGEVENT`

`drawGEVENT(j, len, ht)(btex global\_event etex)` draws a thick rhomboidal Global EVENT box of length `len`, height `ht` and slope `eventslope` with its left-hand bottom corner at `zj`. The string `global_event` is placed at the text center of the box, which is also the geometric center of the box.

`drawcirclebox`

A circle box is a circle with a solid boundary.

`drawcirclebox(j, diam)(btex ABC etex)` draws a circle diameter `diam` and center at `zj`. The string `ABC` is placed at the center of the circle. In this case the four midside points, `zjbm`, `zjmr`, `zjtm` and `zjml`, are at on the circle at the extreme bottom, right, top and left (i.e., at the S, E, N and W compass points). The corner points, `zjbl`, `zjbr`, `zjtr` and `zjtl`, are also on the circumference midway between the midpoints (i.e., at the SW, SE, NE and NW compass points).

`drawdashcircle`

`drawdashcircle(j, diam)` draws a dashed circle. The location and size parameters are the same as for `drawcirclebox`, as are the calculated points.

2.3 Extra BLA variables and routines

In addition to the facilities described above for EXPRESS-G diagrams, extra line styles and boxes are provided for other kinds of diagrams.

Additional line end styles are coded by: `A` for an **A**rrowhead, `OD` for an **O**pen **D**iamond, `CD` for a **C**losed **D**iamond.

`gdl`
`gdb`

The variables `gdl` and `gdb` hold the the length and base width of diamond line end styles (see Figure 7), analagous to the corresponding variables for arrow heads and fanin. By default, `gdl` is twice the default arrowhead length and `gdb` is three quarters of the default arrowhead base width.

`drawA`

`drawA(i, j)` draws a simple arrowhead, length `gal` and base width `gab`, at the `zj` end of the vector from `zi` to `zj`.

`drawDCA`

`drawDCA(i, j)` draws a double closed arrowhead, each of length `gal` and base width `gab`, at the `zj` end of the vector from `zi` to `zj`.

`drawOD`

`drawOD(i, j)` draws an open diamond, length `gdl` and base width `gdb`, at

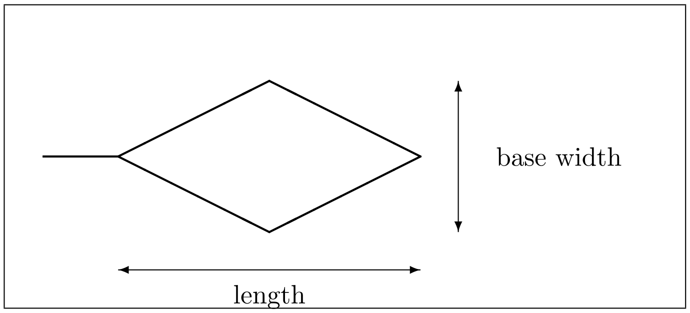

Figure 7: Length parameters for diamond line end styles

the `zj` end of the vector from `zi` to `zj`.

`drawCD`

`drawCD(i, j)` draws a closed (black) diamond, length `gdl` and base width `gdb`, at the `zj` end of the vector from `zi` to `zj`.

`drawdashA`

`drawdashA(i, j)` draws the dashed line `zi--zj`, ending in an arrowhead, length `gal` and base width `gab`, at `zj`.

`drawdashOA`

`drawdashOA(i, j)` draws the dashed line `zi--zj`, ending in an open arrowhead, length `gal` and base width `gab`, at `zj`.

`drawnormalOD`

`drawnormalOD(i, j)` draws the normal thickness line `zi--zj`, ending in an open diamond, length `gdl` and base width `gdb`, at `zj`.

`drawnormalCD`

`drawnormalCD(i, j)` draws the normal thickness line `zi--zj`, ending in a closed (black) diamond, length `gdl` and base width `gdb`, at `zj`.

`drawcircleA`

`drawcircleA(j, diam)` draws a circle diameter `diam` centered at `zj`. A counterclockwise pointing arrow is placed at the top of the circle. The midside and 'corner' points are the same as calculated by the `drawcirclebox` routine.

`drawDot`

`drawDot(j, diam)` draws a black dot, diameter `diam` centered at `zj`. There are no calculated points.

`drawCircledDot`

`drawCircledDot(j, diam)` draws a black dot with a circle outside it, overall diameter `diam`, centered at `zj`. There are no calculated points. This could be used for a UML 'bullseye'. The white ring should obscure anything drawn earlier that lies underneath it (by definition the black dot covers up anything underneath itself). For example:

```
draw z1--z2;            % line from z1 to z2
drawCircledDot(2, len); % obscures last len of above line
draw z3--z2;            % not obscured
```

Commands of the form `draw...box` have a text argument which is printed at the center of the box. If an empty box is required the text argument should be an empty string (`""`). For example `draw...box(...)("")`.

`drawcardbox`

`drawcardbox(j, len, ht, fold)(btex note etex)` draws a rectangular box, length `len` and height `ht`, with its bottom left corner at `zj`. The top right corner of the box is folded down, with the side length of the fold given by `fold`. The string `note` is placed at the center of the box. If no text is required, then use

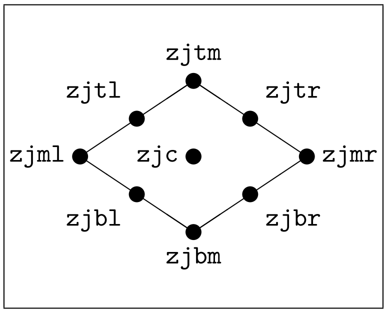

Figure 8: Calculated points on diamond boxes (suffix is `j` and located at `zjc=zj`)

an empty string (i.e., `""`), as in
`\drawcardbox(23, el, eh, 1/8eh)("")`

`drawdiamondbox` `drawdiamondbox(j, len, ht)(btex str etex)` draws a diamond shaped box, length `len` and height `ht`, with its center at `zj`. The usual corner, center and midpoints are also calculated except that the points `z$tm` etc., are at the corners of the diamond and the points `z$tl` etc., are at the middle of each line, as shown in Figure 8. The text `str` is placed at the center of the box.

`drawtwodiamondbox` `drawtwodiamondbox(j, len, ht, inset)(btex str etex)` draws a diamond shaped box, length `len` and height `ht`, with its center at `zj`. The text `str` is placed at the center of the box. A second diamond is drawn inside the first. The parameter `inset` is the space between adjacent outer and inner lines. No calculated points are available for the inner diamond.

`drawdoublerectangle` `drawdoublerectangle(j, len, ht, tf)` draws a rectangular box, length `len` and height `ht`, located with its bottom left corner at `zj`. The box is divided horizontally into a top and bottom portion, with the height of the top portion being the fraction `tf` of the total height of the box. The center points of the two portions are calculated as `zjct` and `zjcb` for the top and bottom respectively. The ends of the dividing line are at `zjtfl` and `zjtfr`.

`drawtriplerectangle` `drawtriplerectangle(j, len, ht, tf, bf)` draws a rectangular box, length `len` and height `ht`, located with its bottom left corner at `zj`. The box is divided horizontally into a top, middle and bottom portion. The height of the top portion is the fraction `tf` of the total height of the box. The height of the bottom portion is the fraction `bf` of the total height of the box. The center points of the three portions are calculated as `zjct`, `zjcm` and `zjcb` for the top, middle and bottom respectively. The ends of the dividing lines are at `zjtfl` and `zjtfr` for the upper line, and at `zjbfl` and `zjbfr` for the lower line.

`hiderectangle` `hiderectangle(j, len, ht)` draws an invisible rectangular box, length `len` and height `ht`, located with its bottom left corner at `zj`. The box covers up anything underneath it. Note that unlike all the other boxes the only point handle to the box is the bottom left corner — the `zjbl`, `zjtr`, etc., points are not available and hence cannot be used for positioning or alignment purposes.

`drawdashboxover` `drawdashboxover(j, len, ht)` draws a dashed rectangular box, length `len` and height `ht`, located with its bottom left corner at `zj`. The box covers up anything underneath it.

`drawindexbox`

`drawindexbox(j, len, ht, lenp, htp)(btex str etex)` draws a rectangular box, length `len` and height `ht`, located with its bottom left corner at `zj`. Another rectangular box, length `lenp` and height `htp`, is drawn with its bottom left corner at the top left corner of the first box (like an index card). The main box points are `zjbl` etc., but the secondary box points are `zjP.bl`, `zjP.bm` etc. The text `str` is placed at the center (`zjP.c`) of the secondary box.

`drawroundedbox`

`drawroundedbox(j, len, ht, rad)(btex str etex)` draws a rectangular box, length `len` and height `ht`, located with its bottom left corner at `zj`. The corners of the box are rounded using radius `rad`. If `2rad` is greater than `len` or `ht`, then `2rad` is reduced to the lesser of `len` and `ht`, thereby producing at least one pair of semicircular sides. The text `str` is placed at the center of the box.

`drawovalbox`

`drawovalbox(j, len, ht)(btex str etex)` draws an elliptical box, horizontal diameter `len` and vertical diameter `ht`, located with its center at `zj`. Like the diamond box, `zjtm`, `zjmr`, etc., are at the extremes of the ellipse and the points `zjtr`, `zjbr`, etc., are on the ellipse approximately halfway between the extremum points, similar to the positions shown in Figure 8. The text `str` is placed at the center of the box.

`drawoutputbox`

`drawoutputbox(j, len, ht)(btex str etex)` draws a box, length `len` and height `ht`, located with its bottom left corner at `zj`. The bottom of the box is a wavy line and the other three sides are straight. The text `str` is placed at the center of the box. This kind of box is often used as a flowchart symbol for output; it is meant to be a represention of printed paper which has been torn from a long continuous roll.

`drawdashellipse`

`drawdashellipse(j, len, ht)` draws a dashed ellipse. The location and dimensional parameters correspond to those for `drawovalbox`, as do the calculated points.

`drawdrum`

`drawdrum(j, len, ht)(btex str etex)` draws a 'drum' box. This is constructed as a rectangular box of width width `len` and height `ht`, with its bottom left hand corner at `zj`. The horizontal line at the bottom of the rectangle is replaced by a half-ellipse with a major diameter of `len` and the minor diameter given by `drumlid*len`. The horizontal line at the top is replaced by a full ellipse with the same dimensions.

As shown in Figure 9, the calculated points on the vertical sides of the drum are the same as for a rectangular box. There are three calculated points on the base of the drum, called `zjbml`, `zjbm` and `zjbmr`. Similarly there are three points on the top of the drum called `zjtml`, `zjtm` and `zjtmr`. The center point, `zjc` is halfway between the sides and halfway between the two lower ellipse halves. The text `str` is placed at this center point.

`drawstickman`

`drawstickman(j, len, ht)` draws a genderless full frontal (or rear) stick figure enclosed in an invisible box, length `len` and height `ht`, located with its bottom left corner at `zj`. This can be used, for example to represent an actor in a UML Use Case diagram. Setting the height to be twice the length produces a reasonable result, for instance: `drawstickman(3, wd, 2wd);`

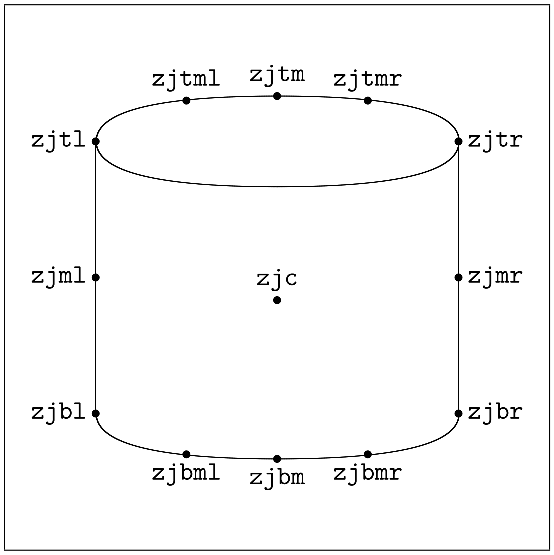

Figure 9: Calculated points on a drum (suffix is `j` and located at `zjbl=zj`)

3 An example

The example attempts to bring out the main methods that can be used in defining EXPRESS-G diagrams. The code for eleven diagrams is provided, none of which necessarily has anything to do with a real example of EXPRESS-G modeling, but rather is meant to be illustrative of diagramming techniques.

This section is best studied with a copy of the printed diagrams to hand. Section 3.5 on page 55 explains briefly how to run the example and obtain the diagrams.

The default extension for a MetaPost file is `.mp`.

```
⟨*eg⟩
%%% EXPEG.MP  Example MetaPost EXPRESS-G diagrams

```

Assuming that the diagrams are to be eventually embedded in, or processed through, a LaTeX document then any TeX macros must come at the start of the file between a `verbatimtex` ... `etex` pair. When dealing with text, MetaPost does not handle more than a single line. The macro below is to enable two lines of text to be handled by MetaPost.

```

% use btex \twolines{first}{second} etex for two-line labels
verbatimtex \def\twolines#1#2{\vbox{\hbox{#1} \hbox{#2}}} etex

```

The expressg package file has to be input to make the routines available.

```
input expressg

```

3.1 Diagram 1: Some boxes and lines

Many diagrams can be defined in one file. Each diagram starts with `beginfig(N)`, where N is a unique integer, and ends with `endfig;`. Encapsulated PostScript for each diagram is written to a file called `mproot.N`, where `mproot` is the root name of the input file. In our case the output files will be `expeg.1` through `expeg.11`.

```
beginfig(1)

```

The first diagram consists of some of the `expressg` box and line styles. The main point is to illustrate how MetaPost allows the boxes to be aligned. Start off by defining a point `z1` at the origin. By default, a variable name starting with the letter `z` is assumed to be a point. Similarly, variables starting with `x` and `y` are assumed to be x- and y-coordinates. If `zk` is defined with a value then automatically `xk` is the value of the x coordinate of the point `zk` and `yk` is the value of the y coordinate. Conversely, if `xj` and `yj` are defined, `zj` is automatically defined.

Note that the `=` sign denotes an equation, not an assignment. The power of MetaPost comes from its ability to solve linear equations.

Having defined the point `z1`, draw an EXPRESSION box with its bottom left hand corner at `z1`.

```
z1=(0,0);
drawEXPRESSION(1);

```

Remember that when a box is drawn, its corner and mid points are calculated. These can be used in defining other points. In particular there is the point `z1tr` which is the top right hand corner of the EXPRESSION box. Also, if another box is drawn at point `z2` then it will have its bottom left hand corner at `z2bl`.

The point `z2bl` is defined by a linear equation with respect to the point `z2`. As MetaPost can solve linear equations, given the point `z2bl` it can solve for the point `z2`.

The next box has its bottom left hand corner at the top right hand corner of the EXPRESSION box, by specifying that the point `z2bl` and `z1tr` are the same location and then drawing a box located at `z2`. The third (BOOLEAN) box is also stacked in the same way with respect to the second (INTEGER) box.

```
z2bl=z1tr;
drawINTEGER(2);

z3bl=z2tr;
drawBOOLEAN(3);

```

Next a piece of circled text is drawn. Using the same method as before, the bottom of the circle is made to be coincident with the middle of the top of the BOOLEAN box. The `namespace` routine is used to specify the circle diameter. Note that an argument to a routine can itself be a routine.

```
z4bm=z3tm;
```

```
drawcirclebox(4, namespace(btex ABS etex)(nextra))(btex ABS etex);

```

MetaPost is a typed language, and new variables can be declared of a particular type. A `numeric` variable holds a number (or a length). The diameter of the circle just drawn is going to be used as a length in later parts of this diagram, so it is sensible to store its value in a new variable called `diam`. We then use it in drawing another circle directly on top of the first one.

```
numeric diam;
diam = namespace(btex ABS etex)(nextra);

z5bm=z4tm;
drawcirclebox(5, diam)(btex SAB etex);

```

The next element in the diagram is an ENTITY box. We use the default height for the entity box and the default page connector length as its length (no reason, but it saves inventing another variable or value).

The ENTITY box is positioned with respect to the SAB circle, but in a slightly more complicated way than earlier positions. We want the left hand side of the box to align with the right hand side of the circle. Also, the middle of the box should align with the top of the circle.

The ENTITY box will be located at point `z6`. The easiest way is to specify `z6` is `z6ml=(x5mr, y5tm)`, but to demonstrate more of MetaPost's equation solving abilities a more complicated approach is taken to the specification of the x coordinate. The horizontal distance between the points `z5ml` and `z5tl` on the circle is the same as the distance between the points `z5tr` and `z5mr`. We can express this as `x5ml-x5tl = x5tr-x5mr`. As we want `x6ml`, which is the same as `x6bl`, to be the same as `x5mr` we can substitute into this equation, as done in the code below. Note that MetaPost can solve for the unknown `x6bl` even though it is on the right hand side of the equation because the other three values are known.

The y coordinate of the point `z6mr` is specified as equal to the y coordinate of the top of the circle. Because of the linear relations involving all the points on a box, it is sufficient to provide one x coordinate and one y coordinate to fix all the points on the box.

```
x5ml-x5tl = x5tr-x6bl;
y6mr=y5tm;
drawENT(6, pconl, enth)(btex an\_ent etex);

```

The geometric center of the next box, an LEVENT, is vertically above the geometric center of the ENTITY box, and the bottom of LEVENT is slightly (one lenth unit) above the top of the ENTITY box. This is accomplished here by specifying `z7bm` in terms of expressions involving the coordinates of the point `z6tm`.

```
z7bm=(x6tm,y6tm+u);
drawLEVENT(7, pconl, eventh)(btex levent etex);

```

At point `z8` a GEVENT box is drawn. It's bottom left hand corner is vertically above the top left hand corner of the LEVENT box. If you look at the resulting diagram, you will see that GEVENT is shifted right with respect to LEVENT. For vertical alignment of EVENT boxes it is best to use the middle rather than corner points.

```
z8=(x7tl, y7tr+u);
drawGEVENT(8, pconl, eventh)(btex gevent etex);

```

`mediation`

For the last of the aligned boxes another aspect of MetaPost is used. MetaPost provides a *mediation* facility whose syntax is `frac[a,b]`. The result of a mediation is a value that is `frac` between the value `a` and the value `b`. For example, the result of `0.25[10,30]` is 15. Similarly, the result of `1/3[zj,zk]` is the point one third of the way between point `zj` and point `zk`.

The SCHEMA box is positioned with its bottom right hand corner at one eighth of the way along, and at the top of, the BOOLEAN box.

```
z9br=1/8[z3tl,z3tr];
drawSCHEMA(9, pconl, schemah)(btex a\_schema etex);

```

The second half of the diagram illustrates various line and end styles. These are all horizontal, the same length and are aligned vertically, so define the x coordinate `xmid` of the left hand end of the lines, and the length `len` of the lines.

```
numeric xmid, len;
xmid = x8tr + diam;
len = 2diam;

```

`shifted`

The next line of code exhibits another MetaPost function. A point can be specified as being some other point that is *shifted* by given amounts in the x and y directions. More precisely, the shift is specified by a pair of values, written as `(a,b)`. The first line is aligned vertically with the center of the first box and its start point is `z11`. The end point, `z111`, is specified as being the start point shifted horizontally by the length of the line.

```
z11=(xmid, y1ml); z111=z11 shifted (len, 0);
drawdash0(11, 111);

```

Because we will be using the same horizontal shift for all the lines, define a *pair*, called `moveright`, that is the required shift. The next two lines are also aligned vertically with the centers of the second and third boxes.

```
pair moveright;
moveright = (len, 0);

z12=(xmid, y2ml); z121=z12 shifted moveright;
drawnormal0(12, 121);

z13=(xmid, y3ml); z131=z13 shifted moveright;
drawthick0(13, 131);

```

The remaining lines will have the same vertical spacing as the first three, so define another pair, `moveup`, that we can use for vertical shift.

```
pair moveup;
moveup = (0, y13-y12);

z14=z13 shifted moveup; z141=z14 shifted moveright;
drawnormalD(14, 141);

z15=z14 shifted moveup; z151=z15 shifted moveright;
drawnormalCA(15, 151);

z16=z15 shifted moveup; z161=z16 shifted moveright;
drawnormalOA(16, 161);

z17=z16 shifted moveup; z171=z17 shifted moveright;
drawnormalF(17, 171);

```

The upper part of the diagram illustrates some of the additional line styles and boxes provided that are not part of the EXPRESS-G language.

The upper half is seperated from the bottom half by a dashed line, just to make the distinction. The line is set above the highest box so far, which is located at `z8`, and extends across the width of the diagram.

```
z300=(0,y8tm+diam); z301=(x111,y300);
draw z300--z301 dashes;

```

Define `mup` to be a distance, which we will use for vertical seperation.

```
numeric mup; mup := onelineh;

```

As previously, we'll start with the boxes which will be placed in a column at the left side of the diagram. The first box, high enough for one line of enclosed text, has its top right hand corner folded down by one quarter the box height. Text is placed at the center of the box, but if this is not required then it can be given as an empty string (i.e., `""`).

```
z302=(0,y300+mup);
drawcardbox(302, pconl, onelineh, 1/4onelineh)("");

```

The next box is a diamond shape. Unlike the rectangular and rhomboidal boxes, these are positioned via the center of the box instead of the bottom left hand corner.

```
z303bm=(x302tm, y302tm+mup);
drawdiamondbox(303, pconl, 2onelineh)(btex jewel etex);

```

Now there is a double box with the top portion being 3/4 of the box height. Text is put into the top half of the box using METAPOST's labeling routine (see page 26).

```
z304=(0, y303tm+mup);
drawdoublerectangle(304, pconl, 2onelineh, 3/4);
label(btex top etex, z304ct);
```

Next comes a triple box with the bottom portion being one fifth of the height, and the top portion being two fifths of the height. Text is placed in the top and middle portions, but not into the bottom.

```
z305=(0, y304tm+mup);
drawtriplerectangle(305, pconl, 3onelineh, 2/5, 0.2);
label(btex top etex, z305ct);
label("middle", z305cm);
```

Some languages include a box that looks a little like an index card, and this is shown next. A small box is also drawn inside the main part of the index box, just to illustrate that drawings may be placed inside boxes.

```
z306=(0,y305tm+mup);
drawindexbox(306, 2pconl, 3onelineh, 3/4pconl, onelineh)(btex pack etex);
z307=1/8[z306bl,z306tr];
drawENT(307, 1/4pconl, onelineh)(btex E etex);
```

Another box shown here is a dotted box that covers up anything that is underneath it. To demonstrate, a folded corner box is drawn first, and then the dashed box is drawn second. They have to be done in this order. (The reason for the notation `y306P.tl` is explained under diagram 2 on page 27).

```
z308=(0, y306P.tl+mup);
z309ml=z308c;
drawcardbox(308, pconl, 2onelineh, 1/4onelineh)(btex covered etex);
drawdashboxover(309, pconl, onelineh);
```

An oval box comes next. The perimeter is an ellipse.

```
z310bm=(x306bm,y309tm+mup);
drawovalbox(310, 2pconl, 2onelineh)("");
```

Now we have two rounded boxes. In the first one the corner radius is small and the corners are neatly rounded. The radius is large in the second case, and it is automatically reduced to give semicircular ends. Further, the center portion of the second of these boxes is ‘deleted’ by putting a `hiderectangle` over it.

```
z311ml=(x310mr+mup, y310ml);
drawroundedbox(311, 2pconl, 2onelineh, 1/2onelineh)("rounded");

z312ml=(x311mr+mup, y311mr);
drawroundedbox(312, 2pconl, 2onelineh, 3/2onelineh)(btex large radius etex);

z314=(x312bm-1/4pconl, y312bm-1/2mup);
hiderectangle(314, 1/2pconl, (3/2onelineh + mup));
```

The last of the simple enclosed shapes are a dashed ellipse and a dashed circle, which are aligned below the partially obscured rounded box.

```
z315tm=(x312bm, y312bm-mup);
drawdashellipse(315, 2pconl, 2onelineh);

z316tm=(x315bm, y315bm-mup);
drawdashcircle(316, diam);

```

A drum box is now presented. This is centered with respect to the rounded box and aligned with the bottom of the triple box. It is the same nominal size as the triple box.

```
z317bc=(x311bm,y304br);
drawdrum(317, pconl, 3onelineh)(btex drum etex);
```

To be a little different, I'll add a 'handle' to the top of the drum. The top of the handle will be above the drum at the point `z317A`, and the attachement points are at the `tml` and `tmr` points on the drum. The normally sharp join in the middle of the handle is smoothed off.

```
z317A=(x317tc, y317tc+2onelineh);
drawnormalthree(317tml, 317A, 317tmr);
smooth(317tml, 317A, 317tmr);

```

Below the drum box is a stick figure. It is centered with respect to the drum box and based on the bottom row.

```
z320bm = (x317bm, y302);
drawstickman(320, onelineh, 2onelineh);

```

Above the drum box there is an output box, vertically aligned with the stick figure, labelled 'output'.

```
z325bm = (x320bm, y306);
drawoutputbox(325, pconl, 2onelineh)(btex output etex);

```

Above the output box there is a simple circled dot.

```
z330 = (x320bm, y309ml);
drawCircledDot(330, 3/2onelineh);

```

As in the bottom half, line styles are drawn on the right side of of the diagram. First, though, there is an arrowed circle symbol. This is vertically aligned with the centers of the lines.

```
z401bm=(1/2[x11,x111], y301+mup);
drawcircleA(401, diam);

```

Finish off with assorted linestyles.

```
z402=(xmid, y401tm+mup); z502=(x111, y402);
drawdashA(402, 502);
```

```

z403=(xmid, y402+mup); z503=(x111, y403);
drawdashOA(403, 503);

z404=(xmid, y403+mup); z504=(x111, y404);
drawnormalOD(404, 504);

z405=(xmid, y404+mup); z505=(x111, y405);
drawnormalCD(405, 505);

z406=(xmid, y405+mup); z506=(x111, y406);
drawnormalDCA(406, 506);

```

Lines can be drawn at any angle, and the non-circular line end styles remain aligned.

`rotatedaround`

A set of lines and endstyles are drawn rotating about the common start point of the lines. `pt1 rotatatedaround(pt2, ang)` is MetaPost code that rotates the point `pt1` around the point `pt2` counterclockwise through the angle `ang` in degrees.

```
numeric ang; ang = 180/7;
z600=(xmid, y406+2mup);
z699=(x600+len, y600);
z601=z699 rotatedaround(z600, ang);
z602=z699 rotatedaround(z600, 2ang);
z603=z699 rotatedaround(z600, 3ang);
z604=z699 rotatedaround(z600, 4ang);
z605=z699 rotatedaround(z600, 5ang);
z606=z699 rotatedaround(z600, 6ang);
z607=z699 rotatedaround(z600, 7ang);
drawnormalCA(600,601);
drawnormalOA(600,602);
drawnormalF(600,603);
drawdashA(600,604);
drawnormalOD(600,605);
drawnormalCD(600,606);

```

This is the end of the first diagram.

```
endfig;   % end figure 1

```

3.2 Diagrams 2–5: EXPRESS-G like models

The diagrams in this section are all drawn using the EXPRESS-G specific routines. Nevertheless, they do illustrate general BLA techniques.

3.2.1 Diagram 2: Schema level diagram

In contrast to the first diagram, this one is much more like an EXPRESS-G diagram. It illustrates a diagram consisting of SCHEMA boxes and interconnections.

```
beginfig(2)                      % schema level model

```

First define some lengths to assist in laying out the diagram, which is going to consists of three rows of boxes with varying numbers of boxes in each row; the bottom row has two, the middle has four, and the top row has one. We also want to align the boxes vertically in a 'nice' manner.

```
numeric lb, hb;                % length & height of boxes
lb=20u; hb=schemah;
numeric upshift;               % vertical distance between rows
upshift = 20u;
numeric sideshift;             % horizontal distance between boxes
sideshift = 10u;

```

The bottom row has two boxes and the second four. The two bottom boxes will be aligned vertically under the two middle boxes in the middle row. Allowance must be made for a 'missing' box at the start of the bottom row. The second box on the bottom row is spaced from the first one.

```
%%% bottom row

z9=(lb+sideshift,0);
drawSCHEMA(9, lb, hb)("nine");

z11=(x9br+sideshift,y9);
drawSCHEMA(11, lb, hb)("eleven");

```

The bottom of the boxes on the second row are all at a fixed distance above the top of the boxes on the first row. We only need to specify the y coordinate of the bottom of one box in the row, and use that for the remainder,

```
%%% middle row

z1=(0,y9tl+upshift);
drawSCHEMA(1, lb, hb)("one");

z3=(x1br+sideshift, y1);
drawSCHEMA(3, lb, hb)("three");

z5=(x3br+sideshift, y1);
drawSCHEMA(5, lb, hb)("five");

z7=(x5br+sideshift, y1);
drawSCHEMA(7, lb, hb)("seven");

```

The single box on the top row will be position centrally above the two middle boxes of the middle row. Mediation is used to calculate the x coordinate that is halfway between the right side of the left box and the left side of the right box.

```
%%% top row

x50 = 1/2[x3br,x5bl];
z31bm=(x50,y1tl+upshift);
drawSCHEMA(31, lb, hb)("thirtyone");

```

This completes the boxes. The diagram is finished off by drawing connections between the boxes. The first set of connections are just straight lines between known points on boxes.

```
%%% connectors

drawdashOO(9tm, 3bm);   % three/nine
drawdashO(5bm, 11tm);   % five/eleven
drawdashO(3ml, 1mr);    % three/one
drawdashO(3mr, 5ml);    % three/five
drawdashOO(9mr, 11ml);  % nine/eleven

```

The next connection is more complex. It goes from schema thirtyone to schema nine on a three-legged route. It goes horizontally left from thirtynine, then vertically down, and finally horizontally right to nine. It also passes to the left of schema one.

The route is constructed by specifying a point `z90` to the left of schema nine, and the point `z31A` on the left hand of schema thirtynine. The `VH` routine is then called to generate the corner point on the (rotated) L shaped path between `z90` and `z31A`. The connection is then drawn using these four points.

```
%%% thirtyone/nine
z90=((x1bl-sideshift), y9ml);
z31A=3/4[z31bl,z31tl];
VH(90, 31A);
drawdashO(90vh, 31A);
drawdashthreeO(90vh, 90, 9ml);
```

`pickup`

To demonstrate another aspect of MetaPost, a dot is drawn at the location of `z90`. If the drawing pen is thick enough, just drawing one point is sufficient to make a noticeable dot. The code `pickup thickpen;` drops the current pen and picks up the thickpen for drawing. The code `pickup normalpen;` picks up the normal pen, which is the one that is normally used.

`label`
`dotlabel`

The last line of code here is `label.lft("z90", z90);` which puts the text 'z90' at the left of point `z90`. The label routine has two arguments, the first the text and the second the point at or near which the text is to be placed. The simplest form of the routine is `label("text", zk)` which centers the text over the point `zk`. A suffix can be used after the `label` (e.g., `.lft` in the code below) to modify the placement. `.lft` positions the text at the left of the point. The other modifiers are: `.rt`, `.top`, `.bot`, `.ulft`, `.urt`, `.llft` and `.lrt`. There is also a `dotlabel.suffix("text", zk)` routine which draws the point as well. For complicated diagrams it can be useful to label some points during development, and then comment them out for the final picture.

```
pickup thickpen;
draw z90;
pickup normalpen;
label.lft("z90", z90);

```

The next two connectors, between schemas one and eleven and between thirtyone and eleven are of a similar U shape. There is, though, one tricky point to remember. Up till now, all point arguments to the expressg routines have been of the form `3` or `4bm`; that is, either a number or a number followed by letter(s). For example, `VH(3,4bm)` will calculate the point `z3vh`. However, if a routine adds more letters to an argument that already has letters, such as `VH(4bm,3)`, the calculated point is *not* `z4bmvh` as might be expected, but instead is `z4bm.vh`. This has tripped me up on several occasions.

```
%%% one/eleven
z1A=1/3[z1bl,z1br];
z11A=(x11bm, (y11bm-sideshift));
VH(1A, 11A);
drawdashthree(1A, 1A.vh, 11A);   % note it is 1A.vh and not 1Avh
drawdashO(11A, 11bm);
dotlabel.rt("z11A", z11A);
dotlabel.lft("z1A.vh", z1A.vh);   % note it is z1A.vh and not z1Avh

```

The connector between thirtyone and eleven is the same general shape as the thirtyone to nine connector. This time, I'll use the `HxH` routine to calculate the turning points

```
%%% thirtyone/eleven
z31B=3/4[z31br,z31tr];
x31BC=x7br+sideshift;
HxH(31B, 31BC, 11mr);
drawdashfourO(31B, 31B.hxh, 11mr.hxh, 11mr);

```

The next two connectors, between thirtyone and seven, and between thirtyone and one, are simple L shapes, which are easier to produce than the U shaped ones.

```
%%% thirtyone/seven
z31D=1/4[z31br,z31tr];
VH(7tm, 31D);
drawnormalthreeO(31D, 7tm.vh, 7tm);

%%% thirtyone/one
z31C=1/4[z31bl,z31tl];
VH(1tm, 31C);
drawdashO(1tm.vh, 31C);
drawdashO(1tm.vh, 1tm);

```

The last two connectors are also simple, each consisting of vertical then horizontal then vertical lines and the turning points can be calculated by the `VhV` routine..

```
%%% thirtyone/three
z31E=1/4[z31bl,z31br];
VhV(31E, 3tm);
drawnormalfourO(31E, 31E.vhv, 3tm.vhv, 3tm);

%%% thirtyone/five
z31F=3/4[z31bl,z31br];
VhV(31F, 5tm);
drawnormalfourO(31F, 31F.vhv, 5tm.vhv, 5tm);

```

The last element in the diagram is a vertical closed arrowheaded line with a label alongside. The arrowhead is on the top line of the connector between thirtyone and eleven. Note the point mediation expression expression in the `label` routine call.

```
%%% labeled arrow
z3111=1/4[z31B,z31B.hxh];
numeric arrowsize;
arrowsize := schemah;
z3112=(x3111, y3111+arrowsize);
drawnormalCA(3112, 3111);
label.rt(btex an\_import etex, 1/2[z3111, z3112]);

```

The end of this diagram.

```
endfig;  % end fig 2

```

3.2.2 Diagram 3: Tree structure

This diagram is the first of three that are related in style. It is useful at this point to define some variables, and values for some of these as they will probably be used in all three diagrams.

Values are assigned (e.g., `var := val;`) so that they can be changed later if necessary (e.g., `var := newval;`). If values were equated like `var = val;` then they cannot be changed afterwards[3] as `var = newval;` will then have specified two incompatible equations for `var` and MetaPostwill report a problem.

```

%%%% some commonly used variables & values

numeric pl, ph;          % length & height of numeric page connectors
pl:=pconl; ph:=pconh;    % for eg., 9,9 (9,9)
numeric plnamed;         % length of named page connector
numeric irh;             % height of interschema boxes
```

---

[3] Except by assignment.

```
irh:=ish;                 % normally one line of text
numeric sdtl, sdth;       % length & height of SDT (e.g. INTEGER) boxes
sdtl:=sdtbl; sdth:=sdtbh;
numeric edtl, edth;       % length & height of enum boxes (inset is sdtbs)
edth:=sdtbh;              % normally one text line
numeric el, eh;           % length & height of entity boxes
eh:=enth;                 % normally one text line
numeric upshift;          % vertical space between rows of boxes
numeric nextup;           % vertical space between bases of rows of boxes
```

Now another bit of MetaPost. There is another type of variable called a `picture` variable. This can hold something as trivial as a dot or a text string, up to a complete diagram. References to an Index type is a common element in some of the diagrams. It is useful to create a shorthand for these by using picture variables.

```
picture sindex, pindex;
sindex := btex eleven.Index etex;      % interschema reference text
pindex := btex 9,9 Index etex;         % page reference text
numeric slindex, plindex;
slindex := namespace(sindex)(nextra);  % length of Index interschema
plindex := namespace(pindex)(nextra);  % length of Index page reference
```

Yet more new MetaPost. Array variables are supported. These are declared by giving the variable name immediately followed by a pair of square brackets. An element of an array can be simply accessed by suffixing the variable name with the integer number denoting the desired position. Actually we have been using array variables all along, as `z3` or `x7` are elements of the arrays `z[]` and `x[]` respectively. We define two arrays, one to be used for specifying horizontal spaces (lengths) and the other for vertical (y) coordinates.

```
numeric hspace[];         % horizontal spaces
numeric lyc[];            % Y coords of box bases

%%%%%%%%%%% end commonly used variables & values

```

Diagram 3 consists of type boxes. We will make these all the same length and height.

```
beginfig(3)                      % example fig 3
%%%drawgrid;

```

The space required for the longest name is `edtl`.

```
edtl := namespace(btex standard\_data\_name etex)(niextra);
```

There are several rows in the diagram. Specify the vertical space between the rows, and the space between the bottoms of the boxes (the height of a box is the normal height, `sdth`, for a type box). Also specify the y coordinates for each row, counting from the bottom.

```
upshift=2sdth;
nextup=sdth+upshift;
```

```
%%         the y coords of box bases
lyc1:=0;
lyc2:=lyc1+nextup;
lyc3:=lyc2+nextup;
lyc4:=lyc3+nextup;
lyc5:=lyc4+nextup;
lyc6:=lyc5+nextup;
lyc7:=lyc6+nextup;
lyc8:=lyc7+nextup;
```

Specify two horizontal spaces.

```
hspace1 := 1/3edtl;  % space between data enums
hspace2 := 2/3edtl;  % initial space at start of second row

```

Draw the 3 boxes in the first row and the 3 in the second; these are staggered with respect to each other and overlap as otherwise there there is not enough space to get them all onto one page. I'll use a numbering scheme for points so that `zCR` is a point in the Cth column and the Rth row.

```
%%% bottom two rows (1 & 2)

z11=(0,lyc1);
drawENUM(11, edtl, edth)(btex coordinates etex);

z22=(hspace2, lyc2);
drawENUM(22, edtl, edth)(btex solution\_data etex);

z31=(x11br+hspace1, lyc1);
drawENUM(31, edtl, edth)(btex pressure\_loads etex);

z42=(x22br+hspace1, lyc2);
drawENUM(42, edtl, edth)(btex nondimensional etex);

z51=(x31br+hspace1, lyc1);
drawENUM(51, edtl, edth)(btex temperature etex);

z62=(x42br+hspace1, lyc2);
drawENUM(62, edtl, edth)(btex force\_loads etex);

```

The third 'row' consists of a horizontal line stretching from the middle of the leftmost box to the middle of the rightmost box. There are connections from this line to the top middle of each of the boxes. We can use the `VyV` routine for the end points of the line.

```
%%% third row (horizontal line between 2 & 4 row)

z13=(x11tm,lyc3);
VyV(11tm, 13, 62tm);
drawnormal(11tm.vyv, 62tm.vyv);
```

Calculate the points on this line vertically above the midpoints of the boxes and draw the connections.

```
z23=(x22tm,lyc3); z33=(x31tm,lyc3); z43=(x42tm,lyc3); z53=(x51tm,lyc3);
drawnormalO(11tm.vyv,11tm);
drawnormalO(23,22tm);
drawnormalO(33,31tm);
drawnormalO(43,42tm);
drawnormalO(53,51tm);
drawnormalO(62tm.vyv,62tm);

```

The fourth row consists of three boxes. There is a vertical line from the bottom of the leftmost box to the horizontal line drawn in row 4. The three boxes are aligned horizontally with respect to this line.

```
%%% fourth row

z3224=1/3[z13,z62tm.vyv];
z3554=2/3[z13,z62tm.vyv];

%%% standard_data_name
z24bm=(x3224,lyc4);
drawSELECT(24, edtl, edth)(btex standard\_data\_name etex);
drawnormal(24bm,3224);

%%% adhoc_data_name
z54bm=(x3554,lyc4);
drawTYPE(54, edtl, edth)(btex adhoc\_data\_name etex);

%%% STRING
z64br=(x62br,lyc4);
drawSTRING(64);

```

The fifth ‘row’ consists of two vertical connectors to the tops of the SELECT and TYPE boxes, with a horizontal line connecting the tops of the connectors.

```
%%% fifth row

z35=(1/2[x24tm,x54tm], lyc5);
VyV(24tm, 35, 54tm);
drawnormalthreeO(35, 24tm.vyv, 24tm);
drawnormalthreeO(35, 54tm.vyv, 54tm);

```

The top row consists of a SELECT box vertically aligned above the center of the horizontal line from the 5th row, together with a line joining these two, and also a page reference box off towards the right hand side

```
%%% sixth row

z36bm=(x35,lyc6);
drawSELECT(36,edtl, edth)(btex data\_name etex);
```

```
drawnormal(36bm,35);

%%% data_name
z66=(x62,lyc6);
drawPREF(66, sdtl, ph)(btex 3,1 (4) etex);

```

The remaining elements on the diagram are: a connector on the 4th row from adhoc_data_name to STRING; and a connector on the 6th row from the page reference to data_name.

```
%%%% rest of the connectors

%%% adhoc/STRING (54, 64)
drawnormalO(54mr,64ml);

%%% PREF/data_name (36, 66)
drawdashO(66ml,36mr);

```

The end of this diagram

```
endfig;                        % end fig 3

```

3.2.3 Diagram 4: Supertypes and subtypes

This diagram consists of a supertype entity and its four subtypes. The entities have various attributes.

```

beginfig(4)                    % example 4
%%%drawgrid;

```

Define some useful lengths.

```
%%% length of named page reference boxes
plnamed := namespace(btex 9,9 data\_name etex)(nextra);

%%% length of entity boxes
el := namespace(btex (ABS) Data\_t etex)(nextra);

upshift := 2sdth;         % vertical space between boxes
nextup := sdth+upshift;   % vertical space between bases of boxes
hspace1 := 1/2el;         % space between entity boxes

%%% the y coords of box bases
lyc1:=0;
lyc2:=lyc1+nextup;
lyc3:=lyc2+nextup;
lyc4:=lyc3+nextup;
lyc5:=lyc4+nextup;
lyc6:=lyc5+nextup;
```

```
lyc7:=lyc6+nextup;
lyc8:=lyc7+nextup;

```

We'll start with the second row which consists of 4 entity (subtype) boxes. The numbering scheme for point suffixes is `zCR`, where `C` is the column number and `R` is the row number.

```
%%%% 2nd row

%%% IntArray
z12=(0,lyc2);
drawENT(12, el, eh)(btex IntArray etex);

%%% RealArray
z22=(x12br+hspace1, lyc2);
drawENT(22, el, eh)(btex RealArray etex);

%%% CharArray
z42=(x22br+hspace1, lyc2);
drawENT(42, el, eh)(btex CharArray etex);

%%% BitArray
z52=(x42br+hspace1, lyc2);
drawENT(52, el, eh)(btex BitArray etex);

```

The first row is four simple data type boxes, aligned underneath the entity boxes.

```
%%%% 1st row

z11bm=(x12bm,lyc1);
drawINTEGER(11);

z21bm=(x22bm,lyc1);
drawREAL(21);

z41bm=(x42bm,lyc1);
drawSTRING(41);

z51bm=(x52bm,lyc1);
drawBINARY(51);

```

The third 'row' is the thick lines connecting the subtype boxes.

```
%%% 3rd row

y13=       y23=       y43=       y53=lyc3;
x13=x12tm; x23=x22tm; x43=x42tm; x53=x52tm;
drawthick(13,53);      % horizontal line
%%% vertical connectors
drawthick0(13,12tm); drawthick0(23,22tm);
```

```
drawthickO(43, 42tm); drawthickO(53, 52tm);

z33=1/2[z13,z53];   % midpoint of horizontal line

```

The 4th row is an abstract supertype entity box, which is made 3 times taller than the default entity box height. The entity box is centered with respect to the row of subtype boxes and is connected to the subtype boxes connector. The digit 1 is placed near the connection point. There is also a page reference and an INTEGER box at the left and right (as attributes of the supertype).

```
%%% 4th row

%%% Data_t
z34bm=(x33,lyc4);
drawENT(34, el, 3eh)(btex \twolines{(ABS)}{Data\_t} etex);
drawthick(34bm,33);
label.urt("1", z33);   % label the connection point

%%% Index
z14=(0,lyc4);
drawPREF(14, plindex, ph)(pindex);

%%% INTEGER
z54br=(x51br,lyc4);
drawINTEGER(54);

```

The 5th row has page reference boxes at the left and right (as attributes of the supertype).

```
%%% 5th row

lyc5:=y34tm-ph;   % change the initial y coord value

%%% Index
z15=(0,lyc5);
drawPREF(15, plindex, ph)(pindex);

%%% DataType
z55=(x54,lyc5);
drawPREF(55, plnamed, ph)(btex 3,1 data\_name etex);

```

The 6th, and last, row, just consists of a page reference centered above the supertype box.

```
%%% 6th row

%%% PREF to Data_t
lyc6:=lyc5+nextup;
z36bm=(x34tm,lyc6);
drawPREF(36, pl, ph)(btex 7,5 (6) etex);

```

The rest of the diagram consists of the labelled attribute lines.

```
%%%% attributes

%%% Sizes  Data_t/Index (34, 14)
z3414=(x34,y14mr);
drawnormal(3414,14mr);
label.ulft(btex Sizes A[1:Limit] etex, z3414);

%%% num  Data_t/INTEGER (34, 54)
z3454=(x34mr,y54ml);
drawnormalO(3454,54ml);
label.urt(btex (DER) num etex, z3454);

%%% Limit Data_t/Index (34, 15)
z3415=(x34,y15mr);
drawnormal(3415,15mr);
label.ulft(btex Limit etex, z3415);

%%% Kind Data_t/data_name (34, 55)
z3455=(x34mr, y55ml);
drawnormal(3455, 55ml);
label.urt(btex Kind etex, z3455);

%%% PREF into Data_t (36, 34)
drawdashO(36bm, 34tm);

%%% Data IntArray/INTEGER (12, 11)
drawnormalO(12bm, 11tm);
label.lrt(btex Data A[1:num] etex, z12bm);

%%% Data RealArray/REAL (22, 21)
drawnormalO(22bm, 21tm);
label.lrt(btex Data A[1:num] etex, z22bm);

%%% Data CharArray/STRING (42, 41)
drawnormalO(42bm, 41tm);
label.lrt(btex Data A[1:num] etex, z42bm);

%%% Data BitArray/BINARY (52, 51)
drawnormalO(52bm, 51tm);
label.lrt(btex Data A[1:num] etex, z52bm);

```

The end of this diagram.

```
endfig;                              % end fig 4

```

3.2.4 Diagram 5: Portion of larger model

This diagram is more complex than the previous ones. It is, nevertheless, presented with virtually no commentary. It consists of three entity boxes and various page reference and interschema boxes to support the entities' attributes.

```
beginfig(5)                 % example 5
%%drawgrid;

```

As usual, specify commonly used variables and values.

```
plnamed := namespace(btex 7,4 Unstructured\_Donor etex)(nextra);

numeric irls, irll;             % length of short and long ISR boxes
irls = namespace(btex eleven.MeshLocation etex)(nextra);
irll = namespace(btex eleven.MeshConnectivityType etex)(nextra);

upshift := irh;
nextup := irh+upshift;

numeric maxrx;         % max x coordinate value of LH of right column boxes
maxrx := maxx-irrl;
numeric xmid;          % x coord of figure center
xmid := 1/2[irls, maxx-irll];

el := namespace(btex MeshConnectivity1\_1 etex)(nextra);
eh := irh;

%%% y coords
lyc1 := 0;
lyc2 := lyc1+nextup;
lyc3 := lyc2+nextup;
lyc4 := lyc3+nextup+irh;
lyc5 := lyc4+nextup;
lyc6 := lyc5+nextup;
lyc7 := lyc6+nextup;
lyc8 := lyc7+nextup;
lyc9 := lyc8+nextup;

```

Rows one and two have one centered box each.

```
%%% Zcol/row
%%% row 1

%%% PREF for Overlapping
z21bm=(xmid,lyc1);
drawPREF(21, pl, ph)(btex 5,3 (8) etex)

%%% row 2

%%% OverlappingArea
```

```
z22bm=(x21bm,lyc2);
drawENT(22, el, eh)(btex OverlappingArea etex);

```

Row three has three boxes, and the middle one is centered.

```
%%% row 3

%%% IndexArray
z13=(pl,lyc3);
drawISR(13, irls, irh)(btex eleven.IndexArray etex);

%%% Index
z23bm=(x22bm,lyc3);
drawISR(23, slindex, irh)(sindex);

%%% MeshLocation
z33br=(maxx,lyc3);
drawISR(33, irls, irh)(btex eleven.MeshLocation etex);

```

Row four has three boxes, and the middle one is centered.

```
%%% row 4

%%% MeshConnectivity
z24bm=(x23bm,lyc4);
drawENT(24, el, eh)(btex MeshConnectivity etex);

%%% PREF for MeshConnectivity
z14ml=(pl,y24ml);
drawPREF(14, pl, ph)(btex 5,2 (8) etex);

%%% Structured_Donor
z34br=(maxx,lyc4);
drawPREF(34, plnamed, ph)(btex 7,5 Structured\_Donor etex);

```

Rows five and six have one box each, at the right hand side.

```
%%% row 5

%%% Unstructured_donor
z35br=(maxx,lyc5);
drawPREF(35, plnamed, ph)(btex 7,4 Unstructured\_Donor etex);

%%% row 6

%%% MeshConnType
z36br=(maxx,lyc6);
drawISR(36, irll, irh)(btex eleven.MeshConnectivityType etex);

```

Row seven has three boxes, and the middle one is centered.

```
%%% row 7

%%% IndexRange
z17=(0,lyc7);
drawISR(17, irls, irh)(btex eleven.IndexRange etex);

%%% INTEGER
x27bm=x24bm; y27ml=y17mr;
drawINTEGER(27);

%%% Zone
z37br=(maxx,lyc7);
drawISR(37, irls, irh)(btex seven.Zone etex);

```

Rows eight and nine have one centered box each.

```
%%% row 8

%%% 1to1
z28bm=(x24bm,lyc8);
drawENT(28, el, eh)(btex MeshConnectivity1\_1 etex);

%%% row 9

%%% PREF to 1to1
z29bm=(x28bm,lyc9);
drawPREF(29, pl, ph)(btex 5,1 (8) etex);

```

All the boxes have been drawn. The remainder is adding and labelling attribute lines. First page references to subtype entities.

```
%%%% attributes, etc

drawthickO(21tm,22bm);   % PREF to Overlap (21, 22)
drawthickO(14mr,24ml);   % PREF to MeshConn (14, 24)
drawthickO(29bm,28tm);   % PREF to 1to1 (29, 28)

```

The attributes of entity OverlappingArea (z22)

```
%%% PointListSize (22, 23)
drawdashO(22tm,23bm);
label.llft(btex PointListSize etex, z23bm);

%%% PointRange (22, 17)
z22A=1/4[z22bl,z22tl]; z17A=(1/2[0,x13], y17);
VH(17A,22A);
drawdashthreeO(22A, 17A.vh, 17A);
label.ulft(btex PointRange etex, z22A);

%%% PointList (22, 13)
z22B=3/4[z22bl,z22tl];
```

```
VH(13bm,22B);
drawdashthreeO(22B, 13bm.vh, 13bm);
label.ulft(btex PointList etex, z22B);

%%% MeshLocation (22, 33)
z22C=1/4[z22br,z22tr];
z33B=2/3[z33bl,z33br];
VH(33B,22C);
drawnormalthreeO(22C, 33B.vh, 33B);
label.urt(btex (DER) Location etex, z22C);

%%% Meshloc (22, 33)
z22D=3/4[z22br,z22tr];
z33A=1/3[z33bl,z33br];
VH(33A,22D);
drawdashthreeO(22D, 33A.vh, 33A);
label.urt(btex Meshloc etex, z22D);

```

The attributes of entity MeshConnectivity (z24).

```
%%% PointListSize (24, 23)
drawnormalO(24bm, 23tm);
label.ulft(btex PointListSize etex, z23tm);

%%% StructuredDonor (24, 34)
z24A=(x24br,y34ml);
draw z24A--z34ml dashes;
label.lrt(btex StructuredDonor etex, z24A);

%%% UnstructuredDonor (24, 35)
z24B=3/4[z24br,z24tr];
HvH(24B,35ml);
drawdashfourO(24B, 24B.hvh, 35ml.hvh, 35ml);
label.urt(btex UnstructuredDonor etex, z24B.hvh);

%%% Meshloc (24, 33)
z24C=4/6[z24bl,z24br];
z33C=(x33A,y33tm);
y2433C=1/3[y33C,y24C];
VyV(24C, 2433C, 33C);
drawdashfourO(24C, 24C.vyv, 33C.vyv, 33C);
label.lrt(btex Meshloc etex, z24C.vyv);

%%% Location (24, 33)
z24D=5/6[z24bl,z24br];
z33D=(x33B,y33tm);
y2433D=2/3[y33D,y24D];
VyV(24D, 2433D, 33D);
drawnormalfourO(24D, 24D.vyv, 33D.vyv, 33D);
label.lrt(btex (DER) Location etex, z24D.vyv);
```

```

%%% ConnectivityType (24, 36)
z24H=(x24D,y24tr);
z36H=z36bm;
y2436H=1/2[y35tr,y36br];
VyV(24H, 2436H, 36H);
drawnormalfourO(24H, 24H.vyv, 36H.vyv, 36H);
label.urt(btex (DER) ConnectivityType etex, z24H.vyv);

%%% Meshcon (24, 36)
z24G=(x24C,y24tr);
z36G=z36ml;
VH(24G,36G);
drawdashthreeO(24G, 24G.vh, 36G);
label.urt(btex Meshcon etex, z24G.vh);

%%% K (24, 27)
drawnormalO(24tm, 27bm);
label.ulft(btex K etex, z24tm);

%%% PointRange (24, 17)
z24E=1/4[z24tl,z24tr];
x17br-x17E = x17A-x17bl;
y17E = y17A;
VhV(24E,17E);
drawdashfourO(24E, 24E.vhv, 17E.vhv, 17E);
label.ulft(btex PointRange etex, z24E);

%%% PointList (24, 13)
z24F=1/4[z24bl,z24br];
z13F=1/2[z13tl,z13tr];
VhV(24F,13F);
drawnormalfourO(24F, 24F.vhv, 13F.vhv, 13F);
label.llft(btex PointList etex, z24F);

```

The attributes of entity MeshConnectivity1_1 (z28)

```
%%% N (28, 27)
z27A=1/5[z27tl,z27tr];
z28A=(x27A,y28bl);
drawnormalO(28A,27A);
label.llft(btex N etex, z28A);

%%% trans (28, 27)
z27B=1/2[z27tl,z27tr];
z28B=(x27B,y28bl);
drawdashO(28B,27B);
label.lrt(btex \twolines{trans}{A[1:N]} etex, z28B);

%%% Transform (28, 27)
```

```
z27C=1/2[z27br,z27tr];
z28C=(1/2[x27C,x28br], y28br);
VH(28C,27C);
drawnormalthreeO(28C,28C.vh,27C);
label.lrt(btex (DER) Transform A[1:N] etex , z28C);

%%% DonorZone (28, 37)
z28D=1/2[z28br,z28tr];
z37D=z37tm;
VH(37D,28D);
drawdashthreeO(28D,37D.vh,37D);
label.urt(btex DonorZone etex, z28D);

%%% DonorPointRange (28, 17)
z28G=3/4[z28bl,z28tl];
z17G=(x17A,y17tl);
VH(17G,28G);
drawnormalthreeO(28G,17G.vh,17G);
label.ulft(btex DonorPointRange etex, z28G);

%%% PointRange (28, 17)
z28H=1/4[z28bl,z28tl];
z17H=(x17E,y17G);
VH(17H,28H);
drawnormalthreeO(28H,17H.vh,17H);
label.ulft(btex PointRange etex, z28H);

```

The end of this diagram.

```
endfig;                                 % end fig 5

```

3.3 Diagrams 6–11: Car model

The diagrams in this section are all taken from [SW94, Chapter 2]. A simple information/data model of a car, car model, and car manufacturer is presented in a variety of BLA languages, among which is, of course, EXPRESS-G. In words, the model is:

> *A car is made by a manufacturer. Each manufacturer has a unique name. A manufacturer constructs cars in several models and a car is of a particular model. A manufacturer gives a serial number to each car he produces and this is unique across all cars produced by that manufacturer. Each model also has a name, and this is unique across all models. A car has a year of production.*

This model is a small portion of a conceptual model developed as a common specification for comparing modeling languages [ISO87].

The purpose of these diagrams is to illustrate how the expressg package supports a range of graphical languages.

3.3.1 Diagram 6: express-g

Having already seen several examples of EXPRESS-G diagrams, this one is presented with no further comments.

```

beginfig(6)            % example 6  EXPRESS-G
%%%drawgrid;

upshift := 3/2onelineh;
nextup := onelineh+upshift;

el := namespace(btex manufacturer etex)(nextra);
eh := onelineh;

hspace1 := 1/2el;

%%% y coords
lyc1 := 0;
lyc2 := lyc1+nextup;
lyc3 := lyc2+nextup;
lyc4 := lyc3+nextup;
lyc5 := lyc4+nextup;

%%% Zcol/row

%%% rows 4 and 5
z14=(0,lyc4);
drawENT(14, el, eh)(btex manufacturer etex);

z25=(x14br+hspace1, lyc5);
drawSTRING(25);

z34=(x25br+hspace1, lyc4);
drawENT(34, el, eh)(btex car\_model etex);

%%% row 3

z33bm=(x34bm, lyc3);
drawENT(33, el, eh)(btex car etex);

%%% row 2

z22=(x25, lyc2);
drawINTEGER(22);

%%% row 1

z21=(x22, lyc1);
drawSTRING(21);

```

```
%%% attributes

%%% manufacturer name (14, 25)
VH(14tm, 25ml);
drawnormalthreeO(14tm, 14tm.vh, 25ml);
label.urt(btex *name etex, z14tm.vh);

%%% model name (34, 25)
VH(34tm, 25mr);
drawnormalthreeO(34tm, 34tm.vh, 25mr);
label.ulft(btex *name etex, z34tm.vh);

%%% made_by (34, 14)
drawnormalO(34ml, 14mr);
label.ulft(btex made\_by etex, z34ml);

%%% model_type (33, 34)
drawnormalO(33tm, 34bm);
label.urt(btex model\_type etex, z33tm);

%%% DER made_by (33, 14)
VH(14bm, 33ml);
drawnormalthreeO(33ml, 14bm.vh, 14bm);
label.ulft(btex (DER) *made\_by etex, z33ml);

%%% year (33, 22)
z33A=1/3[z33bl, z33br];
VH(33A, 22mr);
drawnormalthreeO(33A, 33A.vh, 22mr);
label.ulft(btex year etex, z33A.vh);

%%% serial_no
z33B=2/3[z33bl, z33br];
VH(33B, 21mr);
drawnormalthreeO(33B, 33B.vh, 21mr);
label.ulft(btex *serial\_no etex, z33B.vh);

```

The end of this diagram

```
endfig;               % end fig 6

```

3.3.2 Diagram 7: Shlaer-Mellor

The Shlaer-Mellor graphical language [SM88] is fairly simple, consisting basically of rectangles and arrowheaded lines. However, unlike EXPRESS-G each box usually has several lines of text. These are placed individually at the desired positions within each box.

Notice below that I have used `drawENT` with an empty string to draw the rectangular boxes; the routines for EXPRESS-G diagrams can, if suitable, be used in any sort of diagram.

```
beginfig(7)            % example 7
%%%drawgrid;

el := namespace(btex MANUFACTURER etex)(niextra);
eh := onelineh;

hspace1 := namespace(btex made by maker of etex)(4ndextra);

hspace2 := 6onelineh;   % height of CAR
hspace3 := 5onelineh;   % space between rows
hspace4 := 3onelineh;   % height of MANUFACTURER

%%% y coords
lyc1 := 0;
lyc2 := lyc1+hspace2+hspace3;

%%% Zcol/row

%%% row 2

z12=(0, lyc2);
drawENT(12, el, hspace4)("");
x121=x12+sdtbs;
label.rt(btex MANUFACTURER etex, (x121, 2/3[y12bl, y12tl]));
label.rt(btex * manuf-name etex, (x121, 1/3[y12bl, y12tl]));

z22=(x12br+hspace1, lyc2);
drawENT(22, el, hspace4)("");
x221=x22+sdtbs;
label.rt(btex CAR-MODEL etex, (x221, 2/3[y12bl, y12tl]));
label.rt(btex * model-name etex, (x221, 1/3[y12bl, y12tl]));

%%% row 1

z21=(x22, lyc1);
drawENT(21, el, hspace2)("");
x211=x21+sdtbs;
label.rt(btex CAR etex,         (x211, 5/6[y21bl, y21tl]));
label.rt(btex o year etex,      (x211, 4/6[y21bl, y21tl]));
label.rt(btex o model (R) etex, (x211, 3/6[y21bl, y21tl]));
label.rt(btex * serial-no etex, (x211, 2/6[y21bl, y21tl]));
label.rt(btex * maker (R) etex, (x211, 1/6[y21bl, y21tl]));

%%%%% relationship lines

```

In this diagram the labels on the relationship lines tend to bump into the line ending symbol. METAPOST ignores initial and final spaces within `btex...etex`.

The paired braces below, each enclosing one space, supply non-discarded spaces. (These could also have been written as `\space`).

```
%%% MANUFACTURER / CAR-MODEL (12, 22)
drawnormalDCA(12mr, 22ml);
drawnormalCA(22ml, 12mr);
label.urt(btex { }{ }made by etex, z12mr);
label.llft(btex maker of{ }{ } etex, z22ml);

%%% MANUFACTURER / CAR (12, 21)
VH(12bm, 21ml);
drawnormalCA(12bm.vh, 12bm);
drawnormalDCA(12bm.vh, 21ml);
label.llft(btex { }{ }made by etex, z12bm);
label.ulft(btex maker of{ }{ }  etex, z21ml);

%%% CAR-MODEL / CAR (22, 21)
drawnormalDCA(22bm, 21tm);
drawnormalCA(21tm, 22bm);
label.lrt(btex { }one of etex, z22bm);
label.ulft(btex type of{ }  etex, z21tm);

```

The end of this diagram

```
endfig;              % end fig 7

```

3.3.3 Diagram 8: IDEF1X

IDEF1X is a popular language in America for representing the structure of relational databases, and was developed under the auspices of the US Air Force [IDE85].

The general layout of the diagram mimics the previous one. All the boxes are divided horizontally into an upper and lower part. The `drawdoublerectangle` is used for the rectangular boxes, but the dividing line in the rounded box has to be added by hand, as it were.

```

beginfig(8)            % example 8  IDEF1X
%%%drawgrid;

el := namespace(btex MANUFACTURER etex)(niextra);
eh := onelineh;

hspace1 := namespace(btex made by maker of etex)(4ndextra);

hspace2 := 5onelineh;   % height of CAR
hspace3 := 5onelineh;   % space between rows
hspace4 := 3onelineh;   % height of MANUFACTURER

%%% y coords
```

```
lyc1 := 0;
lyc2 := lyc1+hspace2+hspace3;

%%% Zcol/row

%%% row 2

z12=(0, lyc2);
drawdoublerectangle(12, el, hspace4, 1/2);
label.urt(btex Manufacturer etex, z12tl);
x121=x12+sdtbs;
label.rt(btex manuf-name etex, (x121, 3/4[y12bl, y12tl]));

z22=(x12br+hspace1, lyc2);
drawdoublerectangle(22, el, hspace4, 1/2);
label.urt(btex Car-model etex, z22tl);
x221=x22+sdtbs;
label.rt(btex Model-name etex, (x221, 3/4[y12bl, y12tl]));
label.rt(btex Manuf-name (FK) etex, (x221, 1/4[y12bl, y12tl]));

%%% row 1

z21=(x22, lyc1);
drawroundedbox(21, el, hspace2, 2sdtbs)("");
label.urt(btex Car etex, z21tl);
draw z21ml--z21mr;
x211=x21+sdtbs;
label.rt(btex Serial-no etex,       (x211, 5/6[y21bl, y21tl]));
label.rt(btex Manuf-name (FK) etex, (x211, 4/6[y21bl, y21tl]));
label.rt(btex Model-name (FK) etex, (x211, 2/6[y21bl, y21tl]));
label.rt(btex Year etex,            (x211, 1/6[y21bl, y21tl]));

%%%%% relationship lines

%%% MANUFACTURER / CAR-MODEL (12, 22)
drawnormalD(12mr, 22ml);
label.top(btex of etex, 1/2[z12mr, z22ml]);

%%% MANUFACTURER / CAR (12, 21)
VH(12bm, 21ml);
drawnormalthreeD(12bm, 12bm.vh, 21ml);
label.top(btex makes etex, 1/2[z12bm.vh, z21ml]);
label.ulft(btex P{ }  etex, z21ml);

%%% CAR-MODEL / CAR (22, 21)
drawnormalD(22bm, 21tm);
label.lft(btex of etex, 1/2[z21tm,z22bm]);
label.urt(btex { }P etex, z21tm);

```

The end of this diagram

```
endfig;               % end fig 8

```

3.3.4 Diagram 9: OMT

OMT [RBP+91] was the precursor to the currently popular UML structure modeling language. If you are curious as to why I believe that lexical languages are in general better than BLA languages, the chapter on developing a compiler for OMT highlights, probably unintentionally, some of the disadvantages of BLAs.

The general look of the OMT diagram is almost identical to the IDEF1X diagram.

```

beginfig(9)            % example 9  OMT
%%%drawgrid;

el := namespace(btex {\bf MANUFACTURER} etex)(niextra);
eh := onelineh;

hspace1 := namespace(btex made by maker of etex)(4ndextra);

hspace2 := 4onelineh;   % height of CAR
hspace3 := 4onelineh;   % space between rows
hspace4 := 3onelineh;   % height of MANUFACTURER

%%% y coords
lyc1 := 0;
lyc2 := lyc1+hspace2+hspace3;

%%% Zcol/row

%%% row 2

z12=(0, lyc2);
drawdoublerectangle(12, el, hspace4, 1/2);
x121=x12+sdtbs;
label.rt(btex {\bf Manufacturer} etex, (x121, 3/4[y12bl, y12tl]));
label.rt(btex name : String etex,      (x121, 1/4[y12bl, y12tl]));

z22=(x12br+hspace1, lyc2);
drawdoublerectangle(22, el, hspace4, 1/2);
x221=x22+sdtbs;
label.rt(btex {\bf Car Model} etex, (x221, 3/4[y22bl, y22tl]));
label.rt(btex name: String etex,    (x221, 1/4[y22bl, y22tl]));

%%% row 1

z21=(x22, lyc1);
drawdoublerectangle(21, el, hspace2, 1/3);
```

```
x211=x21+sdtbs;
label.rt(btex {\bf Car} etex,          (x211, 5/6[y21bl, y21tl]));
label.rt(btex serial-no : String etex, (x211, 4/9[y21bl, y21tl]));
label.rt(btex year : Integer etex,     (x211, 2/9[y21bl, y21tl]));

%%%%% relationship lines

%%% MANUFACTURER / CAR-MODEL (12, 22)
drawnormalD(12mr, 22ml);
label.top(btex {\it makes} etex, 1/2[z12mr, z22ml]);
label.ulft(btex 1+{ } etex, z22ml);

%%% MANUFACTURER / CAR (12, 21)
VH(12bm, 21ml);
drawnormalthreeD(12bm, 12bm.vh, 21ml);
label.top(btex {\it makes} etex, 1/2[z12bm.vh, z21ml]);
label.ulft(btex 1+{ }  etex, z21ml);

%%% CAR-MODEL / CAR (22, 21)
drawnormalD(22bm, 21tm);
label.lft(btex {\it produced as} etex, 1/2[z21tm,z22bm]);

```

The end of this diagram

```
endfig;               % end fig 9

```

3.3.5 Diagram 10: E-R

There are several flavours of Entity-Relationship modeling icons. I have used a style from [EN89].

This example introduces some new icons and MetaPost modeling.

```

beginfig(10)            % example 10 E-R
%%%drawgrid;

numeric diam; diam := 2onelineh;

upshift := diam;            % circle diameter
nextup := diam+upshift;

el := namespace(btex MANUFACTURER etex)(niextra);
eh := onelineh;

numeric dl, dh;             % length and height of diamond boxes
dl := 1/2el; dh := 2eh;

numeric marg; marg := 4u;   % double the seperation between parallel lines

```

```
hspace1 := 1/2el;

%%% y coords
lyc1 := 0;
lyc2 := lyc1+nextup;
lyc3 := lyc2+nextup;
lyc4 := lyc3+nextup;
lyc5 := lyc4+nextup;

```

The diagram has 5 rows. Start with the 4th as it is the widest row.

```
%%% Zcol/row
%%% row 4

z14=(0,lyc4);
drawENT(14, el, eh)(btex manufacturer etex);

z34ml=(x14br+hspace1, y14mr);
drawdiamondbox(34, dl, dh)(btex has etex);

z54ml=(x34mr+hspace1, y34mr);
drawENT(54, el, eh)(btex car\_model etex);

%%% row 5

z15bm=(x14bm, lyc5);
drawcirclebox(15, diam)(btex Name etex);

z55bm=(x54bm, lyc5);
drawcirclebox(55, diam)(btex Name etex);

```

The third row consists of an ordinary diamond box at the right and a doubly enclosed diamond on the left. The `drawtwodiamondbox` routine is used for the latter. Note that values of `marg` are used to make the oustside of the double diamond larger than the single diamonds.

```
%%% row 3

z53bm=(x54bm, lyc3);
drawdiamondbox(53, dl, dh)(btex of etex);

z13c=(x14bm, y53c);
drawtwodiamondbox(13, dl+marg, dh+marg, 2/5marg)(btex Makes etex);

```

The second row just consists of a doubly enclosed rectangular box. There is no explicit routine for drawing this, so two `drawENT` routines are used for drawing smaller and larger superimposed boxes (one with an empty text argument).

```
%%% row 2

```

```
z32bm=(x34bm, lyc2);
drawENT(32, el, eh)(btex car etex);
z31c=z32c;
drawENT(31, el+marg, eh+marg)("");

%%% row 1

z21bm=(x32bl, lyc1);
drawcirclebox(21, 5/4diam)(btex \twolines{Serial}{No} etex);

z41bm=(x32br, lyc1);
drawcirclebox(41, diam)(btex Year etex);

%%%%%% lines

%%% Name / manufacturer (15, 14)
drawnormal(15bm, 14tm);

%%% Name / model (55, 54)
drawnormal(55bm, 54tm);

%%% Mnf / has (14, 34)
drawnormal(14mr, 34ml);
label.top(btex 1 etex, 1/2[z14mr,z34ml]);

%%% has/ model (34, 54)
drawnormal(34mr, 54ml);
label.top(btex N etex, 1/2[z34mr,z54ml]);

%%% Mnf / makes (14, 13)
drawnormal(14bm, 13tm);
label.rt(btex 1 etex, 1/2[z14bm,z13tm]);

%%% model / of (54, 53)
drawnormal(54bm, 53tm);
label.rt(btex 1 etex, 1/2[z54bm,z53tm]);

```

In the diagrams so far, except trivially in the first, the lines have all been vertical and/or horizontal. Now we meet lines in arbitrary directions, and parallel to boot.

A pair of parallel lines joins the bottom of the doubly enclosed diamond to the left side of the doubly enclosed rectangle. Drawing one of these is simple — just draw a straight line between the bottom middle of the diamond to the middle left of the rectangle.

```
%%% makes / car (13, 31)
drawnormal(13bm, 31ml);
```

The point `z31A` is `1/2marg` above the point `z31ml`. This is one end of the parallel line.

```
z31A=(x31ml, y31ml+1/2marg);
```

The MetaPost equation `z1-z2 = frac*(z3-z4)` means that (a) the line `z1--z2` is parallel to the line `z3--z4`, and (b) the length of the line `z1--z2` is `frac` of the length of `z3--z4`. We next specify that the line `z13U--z31A` is parallel to, and the same length as, the line `z13bm--z31ml`;

```
z13U-z31A = z13bm-z31ml;
```

The final part of the calculation is to determine the point `z13A` so that the line `z31A--z13A` is parallel to our first line and starts and ends at the outer perimeters of the boxes (and we already know what `z31A` is).

`whatever`

Remember mediation? A point, `z1234` is on the line `z1--z2` if
`z1234=frac1[z1,z2` holds.
Similarly a point `z1234` is on the line `z3--z4` if
`z1234=frac2[z3,z4]` holds.

These two equations are linear and provided enough variables are known it is easy, even if tedious and error prone, to solve for all the unknowns using simple mathematics. Fortunately MetaPost has an idiom for this kind of calculation using the built-in variable called `whatever`. Each time `whatever` is used it is internally given a new name which saves a lot of name inventions on the part of the user. The following line of code results in MetaPost calculating the intersection point, which I call `z13A`, between the two lines `z13U,z31A`, which is our parallel lines whose points are known, and the line `z13bm--z13mr` which is the lower right line of the outer diamond whose points are known.

```
z13A = whatever[z13U,z31A] = whatever[z13bm,z13mr];
```

Finally, draw the second parallel line in the desired position and length.

```
draw z13A--z31A;
label.urt(btex N etex, 1/2[z13A,z31A]);

```

Another pair of parallel lines connects the doubly bounded rectangle and the singly bounded diamond, and we use a similar calculation as above.

```
%%% of / car (53, 31)
drawnormal(53bm, 31mr);
z31B=(x31mr, y31mr+1/2marg);
z53U - z31B = z53bm - z31mr;
z53B = whatever[z53U,z31B] = whatever[z53bm,z53ml];
draw z53B--z31B;
label.ulft(btex N etex, 1/2[z53B,z31B]);

```

That's the end of the tricky bits for this diagram.

```
%%% car / serial no (31, 21)
z31C=(x21tm, y31bl);
drawnormal(31C, 21tm);

%%% car / year (31, 41)
z31D=(x41tm, y31br);
drawnormal(31D, 41tm);

```

The end of this diagram

```
endfig;               % end fig 10

```

3.3.6 Diagram 11: NIAM

Like IDEF1X, NIAM [NH89] is a language for relational database structures. In Europe it tends to be preferred to IDEF1X.

Although the layout and icons of the NIAM illustration differ from the prior diagrams, no new techniques are used. The most obvious of the differences are the use of oval and circular boxes in place of the rectangular boxes, and the use of pairs of rectangular boxes in the middle of the relationship lines.

```

beginfig(11)            % example 11  NIAM
%%%drawgrid;

numeric diam; diam := 2onelineh;    % circle diameter

upshift := diam;
nextup := diam+upshift;

numeric hel, vel;        % length of horizontal and vertical box pairs
eh := onelineh;
hel = namespace(btex made by etex)(nextra);
vel = namespace(btex of prod etex)(nextra);

numeric del, deh;        % horizontal & vertical diameters of oval boxes
del := namespace(btex MANUFACTURER etex)(niextra);
deh := 2eh;

numeric marg; marg := 2u;       % margin for doubly enclosed oval boxes
numeric lmarg; lmarg := marg;   % gap for short lines

numeric dashel, dasheh;         % diameters of dashed ellipses
dashel := del+2marg;
dasheh := deh+2marg;

hspace1 := diam;

%%% y coords
lyc1 := 0;
lyc2 := lyc1+nextup;
lyc3 := lyc2+nextup;

%%% Zcol/row

%%% row 3

x13ml=0; y13bm=lyc3;
```

```
drawovalbox(13, del, deh)(btex manufacturer etex);
z14=z13c;
drawdashellipse(14, dashel, dasheh);

z23ml=(x14mr+hspace1, y14mr);
drawENT(23, hel, eh)(btex of etex);

z33bl=z23br;
drawENT(33, hel, eh)(btex made by etex);

z44ml=(x33mr+hspace1, y33mr);
drawdashellipse(44, dashel, dasheh);
z43=z44c;
drawovalbox(43, del, deh)(btex car model etex);

z53ml=(x44mr+hspace1, y44mr);
drawdashcircle(53, diam);
label(btex year etex, z53c);

%%% row 2

z22c=(x23bm,lyc2);
drawcirclebox(22, namespace(btex U etex)(2nextra))(btex U etex);

x42tm=x43bm; y42ml=lyc2;
drawdoublerectangle(42, vel, 2eh, 1/2);
label(btex of etex, z42ct);
label(btex is of etex, z42cb);

%%z52c=(x53x,lyc2);
x52tm=x53bm; y52ml=lyc2;
drawdoublerectangle(52, vel, 2eh, 1/2);
label(btex of prod etex, z52ct);
label(btex prod in etex, z52cb);

%%% row 1

z11ml=(x13ml,lyc1);
drawdashellipse(11, del, deh);
label(btex serial no etex, z11c);

z21ml=(x23ml,lyc1);
drawENT(21, hel, eh)(btex of etex);

z31bl=z21br;
drawENT(31, hel, eh)(btex has etex);

z41c=(x43c,lyc1);
drawcirclebox(41, diam)(btex car etex);

```

```
%%%%%% lines

%%% manufacturer / of (14, 23)
drawnormal(14mr, 23ml);

%%% made by / car model (33, 44)
drawnormal(33mr, 44ml);
draw (x33tl, y33tl+lmarg)--(x33tr, y33tl+lmarg);
draw (x33mr, y33mr+lmarg)--(x33mr+eh, y33mr+lmarg);

%%% serial no / of (11, 21)
drawnormal(11mr, 21ml);

%%% of / of (23, 22, 21)
drawnormal(23bm, 22tm); drawnormal(22bm, 21tm);
label.rt(btex id etex, z22mr);

%%% has / car (31, 41)
drawnormal(31mr, 41ml);
draw (x31tl, y31tl+lmarg)--(x31tr, y31tl+lmarg);
draw (x31mr, y31mr+lmarg)--(x33mr+eh, y31mr+lmarg);

%%% model / of (44, 42)
drawnormal(43bm, 42tm);

%%% is of / car (42, 41)
drawnormal(42bm, 41tm);
draw (x42tfr+lmarg, y42tfr)--(x42tfr+lmarg, y42br);
draw (x42bm+lmarg, y42bm)--(x42bm+lmarg, y42bm-eh);

%%% year / of prod (53, 52)
drawnormal(53bm, 52tm);

%%% prod in / car (52, 41)
VH(52bm,41mr);
drawnormalthree(52bm, 52bm.vh, 41mr);
draw (x52tfr+lmarg, y52tfr)--(x52tfr+lmarg, y52br);
draw (x52bm+lmarg, y52bm)--(x52bm+lmarg, y52bm-eh);

```

The end of this diagram

```
endfig;              % end fig 11

```

Note that the centers of the boxes on the bottom row are all at `y=0`, and hence the bottom halves have negative y values. This does not matter as MetaPost determines the actual space required for a diagram and does not base it on the origin of the coordinate system.

The end of the MetaPost examples file.

```
end
```

The end of the example.

```
⟨/eg⟩
```

3.4 Comments

Everyone is likely to have a different style of diagramming. You have seen how I do it but that might not be what suits you. Nevertheless, the following comments may be of assistance.

In my own work I tend to draw a lot of diagrams, using different `.mp` files, that have similar sets of 'commonly used variables & values'. Rather than repeating the list in each file, I create a MetaPost file, called say `mycommons.mp`, that includes these and then `input mycommons` after the `input expressg` near the start of each diagram file.

I find it a help to sketch out the diagrams using pencil and paper before coding them up. I have to admit, though, that the final results often look very different from my initial ideas of the layout.

You might have noticed that I start the coding at different places in the diagrams. Basically I try and plan on getting the boxes aligned roughly in rows and/or columns. I then typically start with the longest column, or more often with the widest row, and then fill in the rest from there.

3.5 Running the MetaPost example file

The `expeg.mp` file has to be processed by MetaPost to obtain Encapsulated PostScript (`eps`) files for viewing or printing. Depending on your installation the command to run MetaPost may be either:
`mpost expeg` or
`mp expeg`
or perhaps something similar. This will produce eleven files, `expeg.1` to `expeg.11`, each of which is an `eps` file for the corresponding `beginfig(N)` in `expeg.mp`. MetaPost defaults to using TeX for processing labels which outputs them using Computer Modern fonts. Unfortunately PostScript applications do not understand these so it is necessary to get the fonts inserted. This is most simply done by LaTeXing a document that includes the MetaPost output. A suitable file is provided below and is configured to be processed by either LaTeX or pdfLaTeX. Run LaTeX and then dvips (or what you normally use to generate printable output) and you will get all eleven diagrams.

```
⟨*egt⟩
%%% expeg.tex    display expressg.dtx MetaPost examples

\documentclass[11pt]{article}
\newif\ifpdf
\ifx\pdfoutput\undefined
  \pdffalse
```

```
\else
  \pdftrue
\fi

\ifpdf
  \pdfoutput=1
  \usepackage[pdftex,final]{graphicx}
  \DeclareGraphicsRule{*}{mps}{*}{}
\else
  \usepackage[final]{graphicx}
\fi

%%%% page sizes for ISO document on A4 paper
\setlength{\headheight}{11pt}
\setlength{\headsep}{10mm}
\setlength{\topskip}{11pt}
\setlength{\footskip}{11mm}
\setlength{\textwidth}{160mm}
\setlength{\textheight}{221.5mm}
\setlength{\columnsep}{10mm}
\setlength{\topmargin}{0mm}
\setlength{\oddsidemargin}{0mm}
\setlength{\evensidemargin}{0mm}
\setlength{\marginparwidth}{0pt}
\setlength{\marginparsep}{0pt}
\setlength{\marginparpush}{0pt}
\setlength{\footnotesep}{12pt}
  %%%% for US letterpaper need to change some margins
\setlength{\topmargin}{-9.4mm}
\setlength{\oddsidemargin}{1.55mm}
\setlength{\evensidemargin}{1.55mm}

\begin{document}

\begin{figure}
\centering
  \includegraphics{expeg.1}
\caption{Some boxes and line styles}
\end{figure}

\begin{figure}
\centering
  \includegraphics{expeg.2}
\caption{Example schema level diagram}
\end{figure}

\begin{figure}
\centering
  \includegraphics{expeg.3}
\caption{Example diagram of a tree structure}
```

```
\end{figure}

\begin{figure}
\centering
  \includegraphics{expeg.4}
\caption{Supertypes and subtypes}
\end{figure}

\begin{figure}
\centering
  \includegraphics{expeg.5}
\caption{A portion of a large model}
\end{figure}

\begin{figure}
\centering
  \includegraphics{expeg.6}
\caption{Car model using EXPRESS-G}
\end{figure}

\begin{figure}
\centering
  \includegraphics{expeg.7}
\caption{Car model using Shlaer-Mellor}
\end{figure}

\begin{figure}
\centering
  \includegraphics{expeg.8}
\caption{Car model using IDEF1X}
\end{figure}

\begin{figure}
\centering
  \includegraphics{expeg.9}
\caption{Car model using OMT}
\end{figure}

\begin{figure}
\centering
  \includegraphics{expeg.10}
\caption{Car model using E-R}
\end{figure}

\begin{figure}
\centering
  \includegraphics{expeg.11}
\caption{Car model using NIAM}
\end{figure}

```

```
\end{document}

```

The end of the LaTeX file.

```
⟨/egt⟩
```

3.5.1 Using pdfLaTeX

pdfLaTeX generates a `.pdf` file directly from LaTeX source instead of a `.dvi` file. Unfortunately pdfLaTeX does not understand Encapsulated PostScript in general, but it *does* understand the restricted form of Encapsulated PostScript generated by MetaPost. By default pdfLaTeX will recognise files with an `.mps` extension as coming from MetaPost, but MetaPost generates files with a numeric extension. If you like to use both MetaPost and pdfLaTeX, then it can get tedious changing all the numeric extensions to `.mps`. The following two scripts can help with this. The first is a shell script[4] which calls a Perl script.

The shell script is called `n2mps.sh`. To get suitable versions of the example MetaPost output files for use with pdfLaTeX, just do:
`n2mps.sh expeg`
after having run MetaPost on `expeg.mp`.

```
⟨*shell⟩
#! /bin/sh

######################################################################
# Shell script n2mps.sh
# Call as: n2mps.sh basename
# List each file basename.* in the directory and run the Perl script
# to copy each basename.N to basenameN.mps, where N is an integer
#
# Copyright 2000, Mauro S. Costa and Peter R. Wilson
######################################################################

basename=${1:?"A file basename is required."}
extname=N.mps
echo Files $basename.N, where N is a number, will be copied to $basename$extname
for file in `ls $basename.*`
do
 n2mpsprl.prl $file
done

####################### end shell script ###########################

⟨/shell⟩
```

The Perl script, which is called by the `n2mps` script:

```
⟨*perl⟩

```

---

[4] You may well have to modify this to run on your operating system.

```
#!/usr/local/bin/perl -w

###################################################################
# Perl script: n2mpsprl.prl
# Call as: n2mpsprl.prl filename
# If filename is of the form basename.N, where N is an integer,
# copies file basename.N to file basenameN.mps
#
# Copyright 2000, Mauro S. Costa and Peter R. Wilson
##################################################################

# Test for correct number of input parameters
die "Invalid command line arguments.\nTry $0 <src> \n" if($#ARGV > 1);
die "Invalid command line arguments.\nTry $0 <src> \n" if($#ARGV < 0);

# Assign input file name to variable
$input_file = $ARGV[0];

## test if ends with a number, exit if not
if ($input_file =~ /\w\.\d/) { ; } else { exit; }

# Remove the "dot" from the string variable
# holding the input file name
$input_file =~ s/\.// ;

# Create a list variable composed of the string variable holding
# the concatenated input file name and the extension ".mps"
@name_list = ($input_file,'.mps') ;

# Join the string variables in the name_list variable into
# a single string variable
$output_file = join("",@name_list) ;

# create a list variable composed to the parameters needed
# for the system copy command excution
@exec_list = ('cp', $ARGV[0], $output_file) ;

# Execute the system copy  ("cp") command
system(@exec_list) ;

############################ end perl script ##########################

⟨/perl⟩
```

As an alternative when using pdfLaTeX and the graphicx package, putting
`\DeclareGraphicsRule{*}{mps}{*}{}`
will cause included graphics files with unknown extensions (e.g., as generated by METAPOST) to be treated as `.mps` files.

```
\documentclass...
\usepackage{ifpdf}%   you should have this
```

```
\ifpdf
  \pdfoutput=1
  \usepackage[pdftex,final]{graphicx}
  \DeclareGraphicsRule{*}{mps}{*}{}
\else
  \usepackage[final]{graphicx}
\fi
...
\includegraphics{mpfig.23}
...
```

3.6 Using LaTeX instead of TeX

As already noted, MetaPost defaults to using TeX for typesetting labels. If you would prefer it to use LaTeX instead, then two things have to be done.

1. MetaPost looks at the value of an environment variable called `TEX` to see what typesetting system it should use. If the variable is not set, then it uses TeX. To get it to use LaTeX the environment variable has to be set to `latex`. For example, on the system I use this is done by:

   ```
   TEX=latex
   export TEX
   ```

2. LaTeX `\documentclass` and `\begin{document}` commands must be put into the `verbatimtex`...`etex` group at the start of the diagramming file. For the example, this might look like

   ```
   verbatimtex
     \documentclass{article}
     % \usepackage{...} % for any packages
     \def\twolines#1#2{\vbox{\hbox{#1} \hbox{#2}}}
     \begin{document}
   etex
   ```

NOTE: Both the above are required for LaTeX. On the other hand, if `TEX` is either undefined or has a value other than `tex`, then there must be no LaTeX code anywhere in the `.mp` file.

If you need to generate individual self-contained `ps` files for inclusion in non-LaTeX documents where the font used for labels is a member of the Computer Modern family, then you need to use LaTeX and dvips on `*.tex` files that consist of only a single diagram and no other text at all. At least on my setup I have found it best to run dvips as normal but with output to a file instead of a printer. Then use whatever tools are available to you to convert the `.ps` file to an `.eps` one (or at least set the correct bounding box values). The primary source on generating and using Encapsulated PostScript in the LaTeX world is [Rec97], and [GRM97] also has useful information.

3.7 Using PostScript fonts

METAPOST can be made to use PostScript fonts and, providing the labels don't require any special (TEX) processing, then use of LATEX may be avoided altogether. In order to do this, certain requirements have to be put on the content on the `.mp` file.

`prologues`

- The diagram file must begin with the line
  `prologues:=2;`
  which will add a prologue for PostScript font to the output file.

`defaultfont`

- The `defaultfont` must be changed. What it has to be changed to depends on the font you want to use. For example, to use the Times Roman font, put:
  `defaultfont:="ptmr8r";`
  just after the `prologues` line.

  For this incantation to work, the file `ptmr8r.tfm` must exist in a location where METAPOST looks for `.tfm` files (on the system I use it is in directory `../texmf/fonts/tfm/adobe/times`). Also, the file `psfonts.map` must include a line like
  `ptmr8r  Times-Roman ...`
  Again, on my system `psfonts.map` is in directory `../texmf/dvips/misc`.

  I don't know whether or not this works if you don't have a LATEX distribution — it probably doesn't. But it's unlikely that you don't have one if you intend to use METAPOST.

As a final note on changing fonts, the METAPOST `defaultfont` and `defaultscale` values only apply to quoted text (e.g., `"quoted text"`) and not to 'btexed' text (e.g., `btex btexed text etex`). Conversely, TEX or LATEX commands only apply to btexed text and not to quoted text. For more details on this, see [Wil99].

4 The MetaPost code

We start by announcing what it is for.

```
⟨*up⟩
%%% EXPRESSG.MP  MetaPost macros for EXPRESS-G and other BLA diagrams
%%% version 1.5, 31 July 2003
%%% version 1.6, 29 February 2004
%%% version 1.61, 17 March 2004

show "expressg.mp version 1.61, 2004/03/17";

```

4.1 Variables

`u` The unit of length.

```
newinternal u; numeric u;
u := 1mm;

```

`maxx` `maxy` The maximum x and y coordinates for a diagram.

```
newinternal maxx, maxy;
numeric maxx, maxy;
maxx := 159.5u;        % smidgeon under 160mm for ISO
maxy := 210u;          % 210mm for ISO

```

`defaultdotdiam` `dotdiam` The default and actual diameter of circle and dot line end styles.

```
newinternal defaultdotdiam, dotdiam;
numeric defaultdotdiam, dotdiam;
defaultdotdiam := 2u;
dotdiam := defaultdotdiam;

```

`normalpensize` `normalpensize` `normalpensize` The diameters of the pens for drawing normal, thick and thin lines. (The size of the default METAPOST pen is 0.5bp).

```
newinternal normalpensize, thickpensize, thinpensize;
numeric normalpensize, thickpensize, thinpensize;
normalpensize := 0.5bp;
thickpensize  := 1.5bp;
thinpensize   := 0.25bp;
```

`dotsscale` `dotsscale` is the amount the `...pensize` has to be increased so that a dotted line looks as thick as a continuous or dashed line.

```
newinternal dotsscale; numeric dotsscale;
dotsscale := 2;

```

`normalpen` `thickpen` `thinpen` `dotspen` `dotpen` The pens for drawing lines, plus `dotpen` for drawing a dot line end style.

```
newinternal normalpen, thickpen, thinpen, dotspen, dotpen;
pen normalpen, thickpen, thinpen, dotspen, dotpen;
normalpen := pencircle scaled normalpensize; % i.e. defaultpen;
thickpen  := pencircle scaled thickpensize;
thinpen   := pencircle scaled thinpensize;
dotspen   := pencircle scaled (dotsscale*normalpensize);
dotpen    := pencircle scaled dotdiam;

```

`defaultsmoothrad` `smoothrad` The default and initial values for the radius of the arc used for joining two lines.

```
newinternal defaultsmoothrad;
numeric smoothrad, defaultsmoothrad;
smoothrad := defaultsmoothrad := 2u;

```

`defaultdrumlid` `drumlid` The default and initial values for the ratio of the minor to major diameter of ellipses for the top and bottom of drums.

```
newinternal defaultdrumlid;
numeric drumlid, defaultdrumlid;
drumlid := defaultdrumlid := 0.2;

```

`defaultgal` `defaultgab` `defaultgfl` `defaultgfb` The default values for the length and base width width of arrowheads and fanins.

```
%%% default length and base width for arrowheads and fanin
newinternal defaultgal, defaultgab, defaultgfl, defaultgfb;
numeric defaultgal, defaultgab, defaultgfl, defaultgfb;
defaultgal := defaultgfl := defaultdotdiam;
defaultgab := defaultgfb := defaultgal;

```

`gal` `gab` `gfl` `gfb` Values of length and base width of arrowheads and fanins.

```
%%% length and base width of arrowheads and fanins
newinternal gal, gab, gfl, gfb;
numeric gal, gab, gfl, gfb;
gal := defaultgal;
gab := defaultgab;
gfl := defaultgfl;
gfb := defaultgfb;

```

`onelineh` The minimum height of a box that encloses a single line of text.

```
newinternal onelineh;
numeric onelineh;
onelineh := 5u;

```

`sdtbl` `sdtbh` `sdtbs` The length, height and inset for a simple data type box. The length is sufficient for the name 'BOOLEAN' and the height for one line of text.

```
%%% length, height & inset for simple data type boxes
newinternal sdtbl, sdtbh, sdtbs;
numeric sdtbl, sdtbh, sdtbs;
sdtbl := 22u;
sdtbh := onelineh;
sdtbs := 2u;

```

`sdtbel` `sdtbeh` `sdtbes` The length, height and inset for a simple EXPRESSION data type box. The length is sufficient for the name 'EXPRESSION' and the height for one line of text.

```
%%% length, height & inset for EXPRESSION data type boxes
newinternal sdtbel, sdtbeh, sdtbes;
numeric sdtbel, sdtbeh, sdtbes;
sdtbel := 28u;
```

```
sdtbeh := sdtbh; sdtbes := sdtbs;

```

`sdtbgel` `sdtbgeh` `sdtbges` The length, height and inset for a simple GENERICENT data type box. The length is sufficient for the name 'GENERIC_ENTITY' and the height for one line of text.

```
%%% length, height & inset for GENERICENT data type boxes
newinternal sdtbgel, sdtbgeh, sdtbges;
numeric sdtbgel, sdtbgeh, sdtbges;
sdtbgel := 38u;
sdtbgeh := sdtbh; sdtbges := sdtbs;

```

`pconl` The average length of a numeric page connector (e.g., for `9,9 (9,9)`).

```
%%% average length of numeric page connector
newinternal pconl;
numeric pconl; % length of page connector (for e.g., 9,9 (9,9) )
pconl:=15u;
```

`pconh` Height of average page connector (one line of text).

```
%%% height of page connectors (one line)
newinternal pconh;
numeric pconh;
pconh := onelineh;

```

`enth` `typeh` The height of a one line ENT(ITY) box (`enth`) and the height of a one line ENUM(ERATION), SELECT or TYPE box.

```
%%% heights of entity, enum, select, type boxes (One text line)
newinternal enth, typeh;
numeric enth, typeh;
enth := onelineh;
typeh := onelineh;

```

`ish` `isrh` The height of a normal one line interschema box (`ish`) and an interschema box with one line each for the name and rename (`isrh`).

```
%%% height of interschema boxes (one text line, no rename), and (name + rename)
newinternal ish, isrh;
numeric ish, isrh;
ish  := 2onelineh;
isrh := 3onelineh;

```

`schemah` Height of a one line SCHEMA box.

```
%%% height of schema boxes (one text line)
newinternal schemah;
numeric schemah;
schemah := 2onelineh;

```

`eventh` `eventslope` The height (`eventh`) of a one line (G/L)EVENT box and the slope of the sides of a (G/L)EVENT box.

```
%%% height and slope of event boxes
newinternal eventh, eventslope;
numeric eventh, eventslope;
eventh := onelineh;
eventslope := 0.25;

```

`nextra` `niextra` Margins for namespaces in normal (`nextra`) and inset (`niextra`) boxes. Inset boxes are the ENUM, SELECT and TYPE boxes.

```
%%% extra namespaces for boxed names
newinternal nextra, niextra;
numeric nextra, niextra;
nextra := 2u;
niextra := nextra+sdtbs;

```

`ndextra` Margin for namespace on a relationship line.

```
%%% extra namespace for attribute names
newinternal ndextra;
numeric ndextra;
ndextra := nextra+dotdiam;

```

4.2 Utility routines

`VH` Calculates the intersection point `z$vh` between the vertical line through `z$` and the horizontal line through `z$$` (see Figure 3).

```
%%% calculates mid-point on a path like |_ (vertical, horizontal)
%%% final points are: z$, z$vh, z$$, where z$vh=(x$,y$$)
def VH(suffix $, $$) =
  z$vh=(x$,y$$);
enddef;

```

`VhV` Calculates the point `z$vhv` on the vertical line through `z$` and the point `z$$vhv` on the vertical line through `z$$`, such that the line `z$vhv--z$$vhv` is horizontal and centered vertically between the given points (see Figure 3).

```
%%% Calculates mid-points on a path like |_
%%%                                         |  (vertical, horizontal, vertical)
%%% final points are: z$, z$vhv, z$$vhv, z$$
def VhV(suffix $, $$) =
  y$$vhv=1/2[y$,y$$]; x$$vhv=x$$;
  z$vhv=(x$,y$$vhv);
enddef;

```

HvH Calculates the point `z$hvh` on the horizontal line through `z$` and the point `z$$hvh` on the horizontal line through `z$$`, such that the line `z$hvh--z$$hvh` is vertical and centered horizontally between the given points (see Figure 3).

```
%%% Calculates mid-points on a path like -|_ (horizontal, vertical, horizontal)
%%% final points are: z$, z$hvh, z$$hvh, z$$
def HvH(suffix $, $$) =
  x$hvh=1/2[x$,x$$]; y$hvh=y$;
  z$$hvh=(x$hvh,y$$);
enddef;

```

VyV A generalisation of `VhV`. Calculates the point `z$vyv` on the vertical line through `z$` and the horizontal line through `y@`, and the point `z$$vyv` on the vertical line through `z$$` and the horizontal line through `y@` (see Figure 4).

```
%%% Calculates turning points on a U shaped path (vertical, horizontal, vertical)
%%% final points are: z$, z$vyv, z$$vyv, z$$
def VyV(suffix $, @, $$) =
  z$$vyv=(x$$,y@);
  z$vyv =(x$, y@);
enddef;

```

HxH A generalisation of `HvH`. Calculates the point `z$hxh` on the horizontal line through `z$` and the vertical line through `x@`, and the point `z$$hgh` on the horizontal line through `z$$` and the vertical line through `x@` (see Figure 4).

```
%%% Calculates corner points rotated U shaped path (horizontal, vertical, horizontal)
%%% final points are: z$, z$hxh, z$$hxh, z$$
def HxH(suffix $, @, $$) =
  z$$hxh=(x@,y$$);
  z$hxh =(x@,y$);
enddef;

```

namespace `namespace(`⟨*name*⟩`)(`⟨*margin*⟩`)` calculates the length of the text string ⟨*name*⟩ plus the ⟨*margin*⟩ length. The body is enclosed in parentheses[5] so you can do `c=2namespace...`

```
%%% calculates length taken up by typesetting str, plus margin
def namespace(text str)(expr margin) =
  (xpart lrcorner(str) - xpart llcorner(str) + margin)
enddef;

```

dashes Shorthand for a dashed line style. Use as: `draw ... dashes;`.

```
def dashes =
  dashed evenly
enddef;

```

---

[5] Requested by Stephan Hennig, `ctt` thread *[MP, expressg, latexmp] namespace and textext*, 16 March 2004

`dots` Shorthand for a dotted line style. Use as: `draw ... dots;`.

```
def dots =
  dashed withdots
enddef;

```

`dashedgrid` `dashedgrid(`⟨*nx*⟩, ⟨*ny*⟩, ⟨*dist*⟩`)` draws a thin dashed grid with ⟨*nx*⟩ and ⟨*ny*⟩ divisions in the x and y directions, with ⟨*dist*⟩ between the grid lines. The grid lines are numbered.

```
% draw a general dashed grid
def dashedgrid(expr nx, ny, dist) =
  save zg_, zg__, oldpen;
  pair zg_[], zg__[];
  pen oldpen; oldpen = currentpen;
  pickup thinpen;
  for i = 0 upto nx:
    zg_[i] = (i*dist, 0); zg_[i+1000] = (i*dist, ny*dist);
    draw zg_[i]--zg_[i+1000] dashes;
    label.bot(decimal(i), zg_[i]);
  endfor
  for i = 0 upto ny:
    zg__[i] = (0, i*dist); zg__[i+1000] = (nx*dist, i*dist);
    draw zg__[i]--zg__[i+1000] dashes;
    label.lft(decimal(i), zg__[i]);
  endfor
  pickup oldpen;
enddef;

```

`drawgrid` `drawgrid` draws a thin dashed grid filling the permitted space for an ISO Standard diagram. The units are in mm and the lines are at 10mm intervals with labeling in terms of mm.

```
%%% draw a 1cm spaced grid of 16(x) by 21(y) cms. (units are mm)
def drawgrid =
  dashedgrid(16, 21, 10mm);
enddef;

```

The box drawing routines take as an argument the suffix (`$`) of the left-hand bottom corner point of the box, and normally the length and height of the box. The coordinates of the box corners are calculated (`z$bl`, `z$br`, `z$tr` and `z$tl`). The midpoints of the sides of the box (`z$bm`, `z$mr`, `z$tm`, `z$ml`) are calculated as well. The text 'center' of the box is `z$c`, while the geometric center of the box will be at the intersection point of the lines `z$bm--z$tm` and `z$ml--z$mr`. See Figure 5 for an illustration.

`rectpoints` The routine `rectpoints($, l, h)` calculates the corner and midpoints of a rectangular box with bottom left at `z$`, length `l` and height `h`. Note that it does *not* calculate the center point (see Figure 5).

```
%%% calculates corner and midpoints of a rectangle,
%%% bottom left at $, length l, height h
def rectpoints(suffix $)(expr l, h) =
  z$bl = z$;
  z$tr = (x$+l, y$+h);
  z$br = (x$tr, y$bl);
  z$tl = (x$bl, y$tr);
  z$ml = 1/2[z$bl, z$tl];
  z$mr = 1/2[z$br, z$tr];
  z$bm = 1/2[z$bl, z$br];
  z$tm = 1/2[z$tl, z$tr];
enddef;

```

`rhompoints` The routine `rhompoints($, l, h, s)` calculates the corner and midpoints of a rhomboid length `l`, height `h`, sideslope `s`, and positioned with its bottom left hand corner at `z$`. The center point is *not* calculated (see Figure 6).

```
%%% calculate corner, midpoints and center point of a rhomboid.
def rhompoints(suffix $)(expr l, h, s) =
  save eshift;
  numeric eshift; eshift = s*h;
  z$bl = z$;
  z$tr = (x$+l+eshift, y$+h);
  z$br = (x$bl+l, y$bl);
  z$tl = (x$bl+eshift, y$tr);
  z$ml = 1/2[z$bl, z$tl];
  z$mr = 1/2[z$br, z$tr];
  z$bm = (1/2[x$ml,x$mr], y$bl);
  z$tm = (x$bm, y$tr);
enddef;

```

`circpoints` The routine `circpoints($, d)` calculates the ‘corner’ and ‘midpoints’ on the circumference of a circle, center `z$` and diameter `d`.

```
%%% calculate circumferential points on a circle
def circpoints(suffix $)(expr diam) =
  save rad, sinrad, cosrad;
  numeric rad, sinrad, cosrad;
  rad = diam/2;
  sinrad = rad*(sind 45);
  cosrad = rad*(cosd 45);
  z$c=z$;
  z$ml=(x$c-rad, y$c);
  z$mr=(x$c+rad, y$c);
  z$bm=(x$c, y$c-rad);
  z$tm=(x$c, y$c+rad);
  z$tr=(x$c+cosrad, y$c+sinrad);
  z$bl=(x$c-cosrad, y$c-sinrad);
  z$br=(x$tr, y$bl);
  z$tl=(x$bl, y$tr);
enddef;

```

4.3 Path routines

`~` The binary operator `~` is a reimplementation of the METAFONT plain base `softjoin` path connector (METAFONTbook, page 266); it connects paths via a small circular arc radius `smoothrad`.

```
%%% circular arc join between two paths
tertiarydef p ~ q =
  begingroup
  c_ := fullcircle scaled 2smoothrad shifted point 0 of q;
  a_ := ypart(c_ intersectiontimes p);
  b_ := ypart(c_ intersectiontimes q);
  if a_ < 0: point 0 of p{direction 0 of p} else: subpath(0,a_) of p fi
    ... if b_ < 0: {direction infinity of q}point infinity of q
        else: subpath(b_,infinity) of q fi
  endgroup
enddef;

```

`sharply` `sharply(zi, zj, ...zn)` creates a piecewise linear path through the given points. (The code is based on the `flex` routine, page 267 of the METAFONTbook).

```
%%% piecewise linear path between the given points
def sharply(text t) =     % t is a list of pairs
  hide(n_:=0; for z=t: z_[incr n_]:=z; endfor)
  z_1 for k=2 upto n_: --z_[k] endfor
enddef;

```

`smoothly` `smoothly(zi, zj, ...zn)` creates a piecewise linear path through the given points, with the sharp corners replaced by circular arcs of radius `smoothrad`. page 267 of the METAFONTbook).

```
%%% piecewise linear path between the given points with smooth corners
def smoothly(text t) =     % t is a list of pairs
  hide(n_:=0; for z=t: z_[incr n_]:=z; endfor)
  (z_1 for k=2 upto n_-1: --z_[k]) ~ (z_[k] endfor --z_[n_])
enddef;

```

4.4 Line end drawing routines

`drawO` Draw an open circle, diameter `dotdiam` at the end of the vector `z$` to `z$$`. Any underlying graphic/text will be hidden. All the line end drawing routines have similar code.

```
%%% Draw an open circle at end of vector from $ to $$
```

```
def drawO(suffix $, $$) =
```

Keep everything inside a group and specify the local variables.

```
  begingroup
  save v_, c_, l, p;
```

Specify the types of the local variables.

```
  pair v_, c_;
  numeric l;
  path p;
```

`v_` is the difference between the start and end points of the line. That is, it is the vector from the end point to the start point.

```
  v_ := z$-z$$;
```

`++` `xpart` `ypart`

Using METAPOST's Pythagorean addition operator `++`, where `a++b` means $\sqrt{a^2+b^2}$, we can calculate the length `l` of the line from the x and y dimension (`xpart` and `ypart`) of the vector `v_`.

```
  l := (xpart v_)++(ypart v_);  % length of the line
```

Using mediation involving the ratio of the circle radius to the length of the line, calculate `c_`, the required position of the center of the circle at the end of the line.

```
  c_ := (dotdiam/(2l))[z$$,z$];
```

`p` is the path for drawing the circle.

```
  p := fullcircle scaled dotdiam shifted c_;
```

Erase anything underneath the circle.

```
  unfill p;
```

Now we can draw the circle (twice to make sure it shows).

```
  draw p; draw p;
```

Finish the local group and end.

```
  endgroup
enddef;

```

`drawD` Draw a closed circle, diameter `dotdiam` at the end of the vector `z$` to `z$$`. Any underlying graphic/text will be hidden.

```
%%% Draw a closed circle at end of vector from $ to $$
def drawD(suffix $, $$) =
  begingroup
  save v_, c_, l, p;
  pair v_, c_;
  numeric l;
  path p;
  v_ := z$-z$$;
  l := (xpart v_)++(ypart v_);  % length of the line
  c_ := (dotdiam/(2l))[z$$,z$];
  p := fullcircle scaled dotdiam shifted c_;
  fill p;
  endgroup
```

```
enddef;

```

drawOA Draw an open arrowhead, length `gal` and base width `gab`, at the end of the vector from `z$` to `z$$`. The same general code pattern is used for non-circular line end styles, and is described here. It is not too different from drawing circular ends, except that the orientation of the line has to be taken into account.

```
%%% draw an open arrowhead at end of vector from $ to $$
def drawOA(suffix $, $$) =
  begingroup
  save v_, c_, v_u, c_t, c_b, l, hb, p;
  pair v_, c_, v_u, c_t, c_b;
  numeric l, hb;
  path p;
  hb := gab/2;
  v_ := z$-z$$;
  l := (xpart v_)++(ypart v_);  % length of the line
  c_ := (gal/(l))[z$$,z$];      % base of arrowhead
```

Calculate the unit vector in the direction of the line.

```
  v_u := unitvector v_;
```

Calculate one of the corner points on the base of the triangle by shifting the midbase point in the direction of the normal to the line by half the base width.

```
  c_t := c_ shifted (-hb*(ypart v_u), hb*(xpart v_u));
```

Specify that the other corner point is symmetrically opposite the first.

```
  c_b - c_ = c_ - c_t;
```

Draw the triangular arrowhead.

```
  p = c_b--z$$--c_t--cycle;
  unfill p;
  draw p; draw p;
  endgroup
enddef;

```

drawCA Draw a closed (black) arrowhead, length `gal` and base width `gab`, at the end of the vector from `z$` to `z$$`.

```
%%% draw a closed arrowhead at end of vector from $ to $$
def drawCA(suffix $, $$) =
  begingroup
  save v_, c_, v_u, c_t, c_b, l, hb, p;
  pair v_, c_, v_u, c_t, c_b;
  numeric l, hb;
  path p;
  hb := gab/2;
  v_ := z$-z$$;
  l := (xpart v_)++(ypart v_);  % length of the line
  c_ := (gal/(l))[z$$,z$];      % base of arrowhead
  v_u := unitvector v_;
  c_t := c_ shifted (-hb*(ypart v_u), hb*(xpart v_u));
  c_b - c_ = c_ - c_t;
  p = c_b--z$$--c_t--cycle;
  filldraw p;
  endgroup
enddef;

```

`drawA` Draw a simple arrowhead, length `gal` and base width `gab`, at the end of the vector from `z$` to `z$$`.

```
%%% draw a simple arrowhead at end of vector from $ to $$
def drawA(suffix $, $$) =
  begingroup
  save v_, c_, v_u, c_t, c_b, l, hb, p;
  pair v_, c_, v_u, c_t, c_b;
  numeric l, hb;
  path p;
  hb := gab/2;
  v_ := z$-z$$;
  l := (xpart v_)++(ypart v_);  % length of the line
  c_ := (gal/(l))[z$$,z$];      % base of arrowhead
  v_u := unitvector v_;
  c_t := c_ shifted (-hb*(ypart v_u), hb*(xpart v_u));
  c_b - c_ = c_ - c_t;
  p = c_b--z$$--c_t;
  draw p;
  endgroup
enddef;

```

`drawF` Draw a fanin, length `gfl` and base width `gfb`, at the end of the vector from `z$` to `z$$`.

```
%%% draws a fanin at the end of the vector from $ to $$
def drawF(suffix $, $$) =
  begingroup
  save v_, c_, v_u, c_t, c_b, l, hb, p;
  pair v_, c_, v_u, c_t, c_b;
  numeric l, hb;
  path p;
  hb := gfb/2;
  v_ := z$-z$$;
  l := (xpart v_)++(ypart v_);  % length of the line
  c_ := (gfl/(l))[z$$,z$];      % apex of fan
  v_u := unitvector v_;
  c_t := z$$ shifted (-hb*(ypart v_u), hb*(xpart v_u));
  c_b - z$$ = z$$ - c_t;
  p = c_b--c_--c_t;
  draw p;
  endgroup
```

```
enddef;

```

4.5 Line drawing routines

`drawdots` Draw a dotted line between the two points `z$` and `z$$`.

```
%%% Draw a dotted line from $ to $$
def drawdots(suffix $, $$) =
  pickup dotspen;
  draw z$--z$$ dots;
  pickup normalpen;
enddef;

```

`drawdotsthree` Draw a straight dotted line between the three points `z$`, `z@` and `z$$`.

```
%%% Draws the dotted line $--@--$$
def drawdotsthree(suffix $, @, $$) =
  pickup dotspen;
  draw z$--z@--z$$ dots;
  pickup normalpen;
enddef;

```

`drawdotsfour` Draw a straight dotted line between the four points `z$`, `z@`, `z@@` and `z$$`.

```
%%% Draw the dotted line $--@--@@--$$
def drawdotsfour(suffix $, @, @@, $$) =
  pickup dotspen;
  draw z$--z@--z@@--z$$ dots;
  pickup normalpen;
enddef;

```

`drawdotsO` Draw a dotted line between the two points `z$` and `z$$`, ending in an open circle, diameter `dotdiam`, at `z$$`.

```
%%% Draw a dotted line from $ to $$ with open circle at $$
def drawdotsO(suffix $, $$) =
  pickup dotspen;
  draw z$--z$$ dots;
  pickup normalpen;
  drawO($, $$);
enddef;

```

`drawdotsthreeO` Draw a straight dotted line between the three points `z$`, `z@` and `z$$`, ending in an open circle, diameter `dotdiam`, at `z$$`.

```
%%% Draws the dotted line $--@--$$
def drawdotsthreeO(suffix $, @, $$) =
  pickup dotspen;
  draw z$--z@--z$$ dots;
  pickup normalpen;
  drawO(@, $$);
enddef;

```

`drawdotsfourO` Draw a straight dotted line between the four points `z$`, `z@`, `z@@` and `z$$`, ending in an open circle, diameter `dotdiam`, at `z$$`.

```
%%% Draw the dotted line $--@--@@--$$
def drawdotsfourO(suffix $, @, @@, $$) =
  pickup dotspen;
  draw z$--z@--z@@--z$$ dots;
  pickup normalpen;
  drawO(@@, $$);
enddef;

```

`drawdotsOO` Draw a dotted line between the two points `z$` and `z$$`, ending in open circles, diameter `dotdiam`, at `z$` and `z$$`.

```
%%% Draw the dotted line from $ to $$, ending in circles diameter dotdiam at $ and $$
def drawdotsOO(suffix $, $$) =
  pickup dotspen;
  draw z$--z$$ dots;
  pickup normalpen;
  drawO($, $$); drawO($$, $);
enddef;

```

`drawdash` Draw a dashed line between the two points `z$` and `z$$`.

```
%%% draws a dashed line from $ to $$
def drawdash(suffix $, $$) =
  draw z$--z$$ dashes;
enddef;

```

`drawdashthree` Draw a straight dashed line between the three points `z$`, `z@` and `z$$`.

```
%%% draws the dashed line $--@--$$
def drawdashthree(suffix $, @, $$) =
  draw z$--z@--z$$ dashes;
enddef;

```

`drawdashfour` Draw a straight dashed line between the four points `z$`, `z@`, `z@@` and `z$$`.

```
%%% draws the dashed line $--@--@@--$$
def drawdashfour(suffix $, @, @@, $$) =
  draw z$--z@--z@@--z$$ dashes;
enddef;

```

`drawdashO` Draw a dashed line between the two points `z$` and `z$$`, ending in an open circle, diameter `dotdiam`, at `z$$`.

```
%%% draws a dashed line from $ to $$, ending in a circle diameter dotdiam at $$
def drawdashO(suffix $, $$) =
  draw z$--z$$ dashes;
  drawO($, $$);
enddef;

```

`drawdashthreeO` Draw a straight dashed line between the three points `z$`, `z@` and `z$$`, ending in an open circle, diameter `dotdiam`, at `z$$`.

```
%%% draws the dashed line $--@--$$ with circle at $$
def drawdashthreeO(suffix $, @, $$) =
  draw z$--z@--z$$ dashes;
  drawO(@, $$);
enddef;

```

`drawdashfourO` Draw a straight dashed line between the four points `z$`, `z@`, `z@@` and `z$$`, ending in an open circle, diameter `dotdiam`, at `z$$`.

```
%%% draws the dashed line $--@--@@--$$ with circle at $$
def drawdashfourO(suffix $, @, @@, $$) =
  draw z$--z@--z@@--z$$ dashes;
  drawO(@@, $$);
enddef;

```

`drawdashOO` Draw a dashed line between the two points `z$` and `z$$`, ending in open circles, diameter `dotdiam`, at `z$` and `z$$`.

```
% drawdashOO($, $$)
%%% draws a dashed line from $ to $$, ending in circles diameter dotdiam at $ and $$
def drawdashOO(suffix $, $$) =
  draw z$--z$$ dashes;
  drawO($, $$); drawO($$, $);
enddef;

```

`drawnormal` Draw a normal thickness line between the two points `z$` and `z$$`.

```
%%% draws a normal line from $ to $$.
def drawnormal(suffix $, $$) =
  draw z$--z$$;
enddef;

```

`drawnormalthree` Draw a straight normal thickness line between the three points `z$`, `z@` and `z$$`.

```
% drawnormalthree($, @, $$)
%%% draws the normal line $--@--$$
def drawnormalthree(suffix $, @, $$) =
  draw z$--z@--z$$;
```

```
enddef;

```

drawnormalfour Draw a straight normal thickness line between the four points `z$`, `z@`, `z@@` and `z$$`.

```
%%% draws the normal line $--@--@@--$$
def drawnormalfour(suffix $, @, @@, $$) =
  draw z$--z@--z@@--z$$;
enddef;

```

drawnormalO Draw a straight normal thickness line between the two points `z$` and `z$$`, ending in an open circle, diameter `dotdiam`, at `z$$`.

```
% drawnormalO($, $$)
%%% draws a normal line from $ to $$, ending in a circle diameter dotdiam at $$
def drawnormalO(suffix $, $$) =
  draw z$--z$$;
  drawO($, $$);
enddef;

```

drawnormalthreeO Draw a straight normal thickness line between the three points `z$`, `z@` and `z$$`, ending in an open circle, diameter `dotdiam`, at `z$$`.

```
%%% draws the normal line $--@--$$, ending in a circle at $$
def drawnormalthreeO(suffix $, @, $$) =
  draw z$--z@--z$$;
  drawO(@, $$);
enddef;

```

drawnormalfourO Draw a straight normal thickness line between the four points `z$`, `z@`, `z@@` and `z$$`, ending in an open circle, diameter `dotdiam`, at `z$$`.

```
%%% draws the line $--@--@@--$$, ending in a circle at $$
def drawnormalfourO(suffix $, @, @@, $$) =
  draw z$--z@--z@@--z$$;
  drawO(@@, $$);
enddef;


```

drawnormalOO Draw a straight normal thickness line between the two points `z$` and `z$$`, ending in open circles, diameter `dotdiam`, at `z$` and `z$$`.

```
% drawnormalOO($, $$)
%%% draws a normal line from $ to $$, ending in circles diameter dotdiam at $ and $$
def drawnormalOO(suffix $, $$) =
  draw z$--z$$;
  drawO($, $$); drawO($$, $);
enddef;

```

drawnormalD Draw a straight normal thickness line between the two points `z$` and `z$$`, ending in a black dot, diameter `dotdiam`, at `z$$`.

```
%%% draws a normal line from $ to $$, ending in a dot diameter dotdiam at $$
def drawnormalD(suffix $, $$) =
  draw z$--z$$;
  drawD($, $$);
enddef;

```

drawnormalthreeD Draw a straight normal thickness line between the three points `z$`, `z@` and `z$$`, ending in a black dot, diameter `dotdiam`, at `z$$`.

```
%%% draws the normal line $--@--$$, ending in a dot at $$
def drawnormalthreeD(suffix $, @, $$) =
  draw z$--z@--z$$;
  drawD(@, $$);
enddef;

```

drawnormalfourD Draw a straight normal thickness line between the four points `z$`, `z@`, `z@@` and `z$$`, ending in a black dot, diameter `dotdiam`, at `z$$`.

```
%%% draws the normal line $--@--@@--$$, ending in a dot at $$
def drawnormalfourD(suffix $, @, @@, $$) =
  draw z$--z@--z@@--z$$;
  drawD(@@, $$);
enddef;


```

drawnormalDD Draw a straight normal thickness line between the two points `z$` and `z$$`, ending in black dots, diameter `dotdiam`, at `z$` and `z$$`.

```
%%% draws a normal line from $ to $$, ending in dots diameter dotdiam at $ and $$
def drawnormalDD(suffix $, $$) =
  draw z$--z$$;
  drawD($, $$); drawD($$, $);
enddef;

```

drawnormalOA Draw a straight normal thickness line between the two points `z$` and `z$$`, ending with an open arrowhead, length `gal` and base width `gab`, at `z$$`.

```
%%% draws a normal line from $ to $$, ending with an open arrowhead at $$
def drawnormalOA(suffix $, $$) =
  draw z$--z$$;
  drawOA($, $$);
enddef;

```

drawnormalCA Draw a straight normal thickness line between the two points `z$` and `z$$`, ending with a closed (black) arrowhead, length `gal` and base width `gab`, at `z$$`.

```
%%% draws a normal line from $ to $$, ending with a closed arrowhead at $$
def drawnormalCA(suffix $, $$) =
  draw z$--z$$;
  drawCA($, $$);
enddef;

```

`drawnormalthreeCA` Draw a straight normal thickness line between the three points `z$`, `z@` and `z$$`, ending in a closed arrowhead at `z$$`.

```
%%% draws the normal line $--@--$$, ending in a black arrowhead at $$
def drawnormalthreeCA(suffix $, @, $$) =
  draw z$--z@--z$$;
  drawCA(@, $$);
enddef;

```

`drawnormalfourCA` Draw a straight normal thickness line between the four points `z$`, `z@`, `z@@` and `z$$`, ending in a closed arrowhead at `z$$`.

```
%%% draws the normal line $--@--@@--$$, ending in a black arrowhead at $$
def drawnormalfourCA(suffix $, @, @@, $$) =
  draw z$--z@--z@@--z$$;
  drawCA(@@, $$);
enddef;

```

`drawnormalF` Draw a straight normal thickness line between the two points `z$` and `z$$`, ending with a fanin, length `gfl` and base width `gfb`, at `z$$`.

```
%%% draws a normal line from $ to $$, ending with a fanin at $$
def drawnormalF(suffix $, $$) =
  draw z$--z$$;
  drawF($, $$);
enddef;

```

`drawnormalFO` Draw a straight normal thickness line between the two points `z$` and `z$$`, ending with a fanin, length `gfl` and base width `gfb`, at `z$` and an open circle, diameter `dotdiam`, at `z$$`.

```
%%% draws a normal line from $ to $$, with a fanin at $ and an open circle at $$
def drawnormalFO(suffix $, $$) =
  draw z$--z$$;
  drawO($, $$); drawF($$, $);
enddef;

```

`drawthick` Draw a straight thick line between the two points `z$` and `z$$`

```
%%% draws a thick line from $ to $$.
def drawthick(suffix $, $$) =
  pickup thickpen;
  draw z$--z$$;
  pickup normalpen;
enddef;

```

drawthickO Draw a straight thick line between the two points `z$` and `z$$`, ending with an open circle, diameter `dotdiam`, at `z$$`.

```
%%% draws a thick line from $ to $$, ending in a circle diameter dotdiam at $$
def drawthickO(suffix $, $$) =
  pickup thickpen;
  draw z$--z$$;
  drawO($, $$);
  pickup normalpen;
enddef;

```

drawthickOO Draw a straight thick line between the two points `z$` and `z$$`, ending with open circles, diameter `dotdiam`, at `z$` and `z$$`.

```
%%% draws a thick line from $ to $$, ending in circles diameter dotdiam at $ and $$
def drawthickOO(suffix $, $$) =
  pickup thickpen;
  draw z$--z$$;
  drawO($, $$); drawO($$, $);
  pickup normalpen;
enddef;

```

smooth Rounds the join at `z@` between the two straight lines `z$--z@--z$$`. The radius of the circular arc is `smoothrad`. The code employs the same technique as used for putting dots, etc., at the end of a line. However, in this case we ‘undraw’ the sharp corner before drawing the arc.

```
%%% replaces the sharp corner on $--@--$$ with a circular arc radius smoothrad
def smooth(suffix $, @, $$) =
  begingroup
  save v_, c_, l, p;
  pair v_, v_', c_, c_';
  path p;
  v_ := z@-z$;
  l := (xpart v_)++(ypart v_);     % length of $--@
  c_ := (smoothrad/l)[z@,z$];      % start of arc on $--@
  v_' := z$$-z@;
  l := (xpart v_')++(ypart v_');   % length of @--$$
  c_' := (smoothrad/l)[z@,z$$];    % end of arc on @--$$
  undraw c_--z@--c_';              % blank original join
  draw c_{v_}..{v_'}c_';           % draw the arc
  endgroup
enddef;

```

smoothtwo Rounds the joins at `z@` and `z@@` between the three straight lines `z$--z@--z@@--z$$`. The radius of the circular arc is `smoothrad`.

```
%%% replaces the sharp corners on $--@--@@--$$ with a circular arc radius smoothrad
def smoothtwo(suffix $, @, @@, $$) =
  smooth($, @, @@); smooth(@, @@, $$);
enddef;

```

`smoothdash` Rounds the join at `z@` between the two dashed straight lines `z$--z@--z$$`. The radius of the circular arc is `smoothrad`.

```
%%% replaces the sharp corner on the dashed lines $--@--$$
%%% with a circular arc radius smoothrad
def smoothdash(suffix $, @, $$) =
  begingroup
  save v_, c_, l, p;
  pair v_, v_', c_, c_';
  path p;
  v_ := z@-z$;
  l := (xpart v_)++(ypart v_);     % length of $--@
  c_ := (smoothrad/l)[z@,z$];      % start of arc on $--@
  v_' := z$$-z@;
  l := (xpart v_')++(ypart v_');   % length of @--$$
  c_' := (smoothrad/l)[z@,z$$];    % end of arc on @--$$
  undraw c_--z@--c_';              % blank original join
  draw c_{v_}..{v_'}c_' dashes;    % draw the dashed arc
  endgroup
enddef;

```

`smoothdots` Rounds the join at `z@` between the two dotted straight lines `z$--z@--z$$`. The radius of the circular arc is `smoothrad`.

```
%%% replaces the sharp corner on the dotted line $--@--$$
%%% with a circular arc radius smoothrad
def smoothdots(suffix $, @, $$) =
  begingroup
  save oldpen;
  pen oldpen; oldpen := currentpen;
  save v_, c_, l, p;
  pair v_, v_', c_, c_';
  path p;
  v_ := z@-z$;
  l := (xpart v_)++(ypart v_);     % length of $--@
  c_ := (smoothrad/l)[z@,z$];      % start of arc on $--@
  v_' := z$$-z@;
  l := (xpart v_')++(ypart v_');   % length of @--$$
  c_' := (smoothrad/l)[z@,z$$];    % end of arc on @--$$
  undraw c_--z@--c_';              % blank original join
  pickup dotspen;
  draw c_{v_}..{v_'}c_' dots;      % draw the dotted arc
  pickup oldpen;
  endgroup
```

```
enddef;

```

4.6 Box drawing routines

The box drawing routines take as an argument the suffix (`$`) of the left-hand bottom corner point of the box, and normally the length and height of the box. There is normally also a text argument that gets typeset at the 'center' of the box. The coordinates of the box corners are calculated (`z$bl`, `z$br`, `z$tr` and `z$tl`). The midpoints of the sides of the box (`z$bm`, `z$mr`, `z$tm`, `z$ml`) are calculated as well. The text 'center' of the box is `z$c`, while the geometric center of the box will be at the intersection point of the lines `z$bm--z$tm` and `z$ml--z$mr`. See Figure 5 for an illustration.

`drawSCHEMA` `drawSCHEMA(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)` draws a SCHEMA box.

```
%%% draws a schema box, bottom left at $, length l, height h
def drawSCHEMA(suffix $)(expr l, h)(text str) =
  rectpoints($, l, h);
  x$c = 1/2[x$ml, x$mr];
  y$c = 1/2[y$ml, y$tl];
  draw z$bl--z$br--z$tr--z$tl--cycle;  % outer box
  draw z$ml--z$mr;                     % dividing line
  label(str, z$c);
enddef;

```

`drawSDT` `drawSDT(`⟨*suffix*⟩`)(`⟨*name*⟩`)` draws a simple data type box of length `sdtbl` and height `sdtbh`.

```
% drawSDT($)(name)
% draw a simple data type box, bottom left at $
def drawSDT(suffix $)(text str) =
  rectpoints($, sdtbl, sdtbh);
  z$ti = (x$tr-sdtbs, y$tr);
  z$bi = (x$ti, y$br);
  z$c = 1/2[z$bl,z$ti];
  draw z$bl--z$br--z$tr--z$tl--cycle;
  draw z$bi--z$ti;
  label(str, z$c);
enddef;

```

`drawASDT` `drawASDT(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)` draws a simple data type box.

```
% drawASDT($, l, h)(name)
% draw a simple data type box, bottom left at $, length l, height h
def drawASDT(suffix $)(expr l, h)(text str) =
  rectpoints($, l, h);
  z$ti = (x$tr-sdtbs, y$tr);
  z$bi = (x$ti, y$br);
  z$c = 1/2[z$bl,z$ti];
  draw z$bl--z$br--z$tr--z$tl--cycle;
  draw z$bi--z$ti;
  label(str, z$c);
enddef;

```

The `drawNAME(`⟨*suffix*⟩`)` routines draw a simple data type box of the given name using the applicable lengths and heights.

`drawBINARY`
`drawBOOLEAN`
`drawCOMPLEX`
`drawEXPRESSION`

```
def drawBINARY(suffix $) =  drawASDT($)(sdtbl, sdtbh)("BINARY");
  enddef;
def drawBOOLEAN(suffix $) = drawASDT($)(sdtbl, sdtbh)("BOOLEAN");
  enddef;
def drawCOMPLEX(suffix $) = drawASDT($)(sdtbl, sdtbh)("COMPLEX");
  enddef;
def drawEXPRESSION(suffix $) = drawASDT($)(sdtbel, sdtbeh)("EXPRESSION");
  enddef;
```

`drawGENERIC`
`drawINTEGER`
`drawLOGICAL`
`drawNUMBER`

```
def drawGENERIC(suffix $) = drawASDT($)(sdtbl, sdtbh)("GENERIC");
  enddef;
def drawINTEGER(suffix $) = drawASDT($)(sdtbl, sdtbh)("INTEGER");
  enddef;
def drawLOGICAL(suffix $) = drawASDT($)(sdtbl, sdtbh)("LOGICAL");
  enddef;
def drawNUMBER(suffix $) =  drawASDT($)(sdtbl, sdtbh)("NUMBER");
  enddef;
```

`drawREAL`
`drawSTRING`

```
def drawREAL(suffix $) =    drawASDT($)(sdtbl, sdtbh)("REAL");
  enddef;
def drawSTRING(suffix $) =  drawASDT($)(sdtbl, sdtbh)("STRING");
  enddef;

```

`drawENUM`

`drawENUM(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)` draws an ENUMERATION type box.

```
% drawENUM($, l, h)(name)
%%% draw an enumeration type box, bottom left at $, length l, height h
def drawENUM(suffix $)(expr l, h)(text str) =
  rectpoints($, l, h);
  z$ti = (x$tr-sdtbs, y$tr);
  z$bi = (x$ti, y$br);
  z$c = 1/2[z$bl,z$ti];
  draw z$bl--z$br--z$tr--z$tl--cycle dashes;
  draw z$bi--z$ti dashes;
  label(str, z$c);
enddef;

```

drawSELECT `drawSELECT(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)` draws a SELECT type box.

```
% drawSELECT($, l, h)(name)
%%% draw a select type box, bottom left at $, length l, height h
def drawSELECT(suffix $)(expr l, h)(text str) =
  rectpoints($, l, h);
  z$ti = (x$tl+sdtbs, y$tl);
  z$bi = (x$ti, y$bl);
  z$c = 1/2[z$br,z$ti];
  draw z$bl--z$br--z$tr--z$tl--cycle dashes;
  draw z$bi--z$ti dashes;
  label(str, z$c);
enddef;

```

drawTYPE `drawTYPE(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)` draws a user-defined TYPE type box.

```
% drawTYPE($, l, h)(name)
%%% draw a simple user defined TYPE box, bottom left at $, length l, height h
def drawTYPE(suffix $)(expr l, h)(text str) =
  rectpoints($, l, h);
  z$c = 1/2[z$bl,z$tr];
  draw z$bl--z$br--z$tr--z$tl--cycle dashes;
  label(str, z$c);
enddef;

```

drawENT `drawENT(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)` draws an ENTITY box.

```
% drawENT($, l, h)(name)
%%% draw an entity  box, bottom left at $, length l, height h
def drawENT(suffix $)(expr l, h)(text str) =
  rectpoints($, l, h);
  z$c = 1/2[z$bl,z$tr];
  draw z$bl--z$br--z$tr--z$tl--cycle;
  label(str, z$c);
enddef;

```

drawOB `drawOB(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)` draws a rectangular box with rounded corners (an EXPRESS-G oval box).

```
% drawOB($, l, h)
% draw an oval box, bottom left at $, length l, height h
def drawOB(suffix $)(expr l, h) =
  save rad;
  numeric rad; rad := h/2;
  rectpoints($, l, h);
  z$cl = 1/2[z$bl, z$tl];
  z$cr = 1/2[z$br, z$tr];
  z$bli = (x$bl+rad, y$bl);
  z$tli = (x$bli, y$tl);
  z$bri = (x$br-rad, y$br);
  z$tri = (x$bri, y$tr);
  z$c = 1/2[z$bl,z$tr];
  draw z$bli--z$bri..z$cr..z$tri--z$tli..z$cl..cycle;
enddef;

```

`drawPREF` `drawPREF(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)` draws a page reference oval box.

```
% drawPREF($, l, h)(name)
%%% draw a page reference box oval, bottom left at $, length l, height h
def drawPREF(suffix $)(expr l, h)(text str) =
  drawOB($, l, h);
  label(str, z$c);
enddef;

```

`drawISU` `drawISU(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)` draws an interschema USE box.

```
% drawISU($, l, h)(name)
%%% draw an interschema USE box, bottom left at $, length l, height h
def drawISU(suffix $)(expr l, h)(text str) =
  save quarter;
  numeric quarter; quarter := h/4;
  rectpoints($, l, h);
  z$o = (x$, y$+quarter);
  drawOB($o, l, 2quarter);
  z$c = 1/2[z$bl,z$tr];
  label(str, z$c);
  draw z$bl--z$br--z$tr--z$tl--cycle;
enddef;

```

`drawISUR` `drawISUR(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)(`⟨*rename*⟩`)` draws an interschema USE RENAME box.

```
% drawISUR($, l, h)(name)(rename)
%%% draw an interschema USE RENAME box, bottom left at $, length l, height h
def drawISUR(suffix $)(expr l, h)(text str, rname) =
  save third;
  numeric third; third := h/3;
  rectpoints($, l, h);
  z$o = (x$, y$+third);
  drawOB($o, l, third);
  z$c = 1/2[z$bl, z$tr];
  label(str, z$c);
  z$rnm = 1/2[z$bl,(x$br,y$br+third)];
  draw z$bl--z$br--z$tr--z$tl--cycle;
  label(rname, z$rnm);
enddef;

```

`drawISR` `drawISR(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)` draws an interschema RERERENCE box.

```
% drawISR($, l, h)(name)
%%% draw an interschema REFERENCE box, bottom left at $, length l, height h
def drawISR(suffix $)(expr l, h)(text str) =
  save quarter;
  numeric quarter; quarter := h/4;
  rectpoints($, l, h);
  z$o = (x$, y$+quarter);
  drawOB($o, l, 2quarter);
  z$c = 1/2[z$bl,z$tr];
  label(str, z$c);
  draw z$bl--z$br--z$tr--z$tl--cycle dashes;
enddef;
```

`drawISRR` `drawISRR(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)(`⟨*rename*⟩`)` draws an interschema REFERENCE RENAME box.

```
% drawISRR($, l, h)(name)(rename)
%%% draw an interschema REFERENCE RENAME box, bottom left at $, length l, height h
def drawISRR(suffix $)(expr l, h)(text str, rname) =
  save third;
  numeric third; third := h/3;
  rectpoints($, l, h);
  z$o = (x$, y$+third);
  drawOB($o, l, third);
  z$c = 1/2[z$bl,z$tr];
  label(str, z$c);
  z$rnm = 1/2[z$bl,(x$br,y$br+third)];
  draw z$bl--z$br--z$tr--z$tl--cycle dashes;
  label(rname, z$rnm);
enddef;
```

`drawLEVENT` `drawLEVENT(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)` draws a Local EVENT box, with side slope `eventslope`.

```
% drawLEVENT($, l, h)(name)
%%% draw a Local EVENT box, bottom left at $, length l, height h
def drawLEVENT(suffix $)(expr l, h)(text str) =
  rhompoints($, l, h, eventslope);
  z$c = 1/2[z$ml,z$mr];
  draw z$bl--z$br--z$tr--z$tl--cycle;
  label(str, z$c);
enddef;
```

`drawGEVENT` `drawGEVENT(`⟨*suffix*⟩, ⟨*l*⟩, ⟨*h*⟩`)(`⟨*name*⟩`)` draws a Global EVENT box, with side slope `eventslope`.

```
% drawGEVENT($, l, h)(name)
%%% draw a Global EVENT box, bottom left at $, length l, height h
def drawGEVENT(suffix $)(expr l, h)(text str) =
  rhompoints($, l, h, eventslope);
  z$c = 1/2[z$ml,z$mr];
  pickup thickpen;
  draw z$bl--z$br--z$tr--z$tl--cycle;
  pickup normalpen;
  label(str, z$c);
enddef;

```

drawcirclebox

`drawcirclebox`(⟨*suffix*⟩, ⟨*diam*⟩)(⟨*name*⟩) draws a circle, center `zsuffix` and diameter ⟨*diam*⟩, around ⟨*name*⟩.

```
% drawcirclebox($, diam)(name)
%%% draw a circled name, diameter diam centered at $
def drawcirclebox(suffix $)(expr diam)(text str) =
  circpoints($, diam);
  draw z$bl..z$bm..z$br..z$mr..z$tr..z$tm..z$tl..z$ml..cycle;
  label(str, z$c);
enddef;

```

4.7 Extra BLA variables and routines

Some extra facilities are provided for assistance in drawing non-EXPRESS-G BLA diagrams, such as flow charts, IDEF diagrams or UML structure diagrams.

gdl
gdb

`gdl` is the length of a diamond line end style and `gdb` is the base width.

```
%%% length and base width of diamond line end styles
newinternal gdl, gdb;
numeric gdl, gdb;
gdl := 2defaultgal;
gdb := 0.75defaultgab;

```

ellipsepoints

Calculates the points on an ellipse, center at `z$` with horizontal diameter `l` and vertical diameter `h`. It does *not* calculate the center point.

```
def ellipsepoints(suffix $)(expr l, h) =
  save epp, move;
  path epp;
  pair move;
  z$ml=(x$-l/2, y$); z$mr=(x$ml+l,y$);
  z$tm=(x$, y$+h/2); z$bm=(x$, y$tm-h);
  move = 1/2[z$ml,z$mr];
  epp = fullcircle scaled h xscaled (l/h) shifted move;
  z$tr = point 1.2 of epp;
  z$tl = point 2.8 of epp;
  z$bl = point 5.2 of epp;
  z$br = point 6.8 of epp;
enddef;

```

drawDCA Draw double closed arrowheads, each length `gal` and base width `gab`, at end of vector from `z$` to `z$$`.

```
%%% draws double closed arrowheads at end of vector from $ to $$
def drawDCA(suffix $, $$) =
  begingroup
  save v_, c_, v_u, c_t, c_b, l, hb, p;
  pair v_, c_, v_u, c_t, c_b;
  path p[];
  numeric l, hb;
  hb := gab/2;
  v_ := z$-z$$;
  l := (xpart v_)++(ypart v_);  % length of the line
  c_ := (gal/(l))[z$$,z$];      % base of arrowhead
  v_u := unitvector v_;
  c_t := c_ shifted (-hb*(ypart v_u), hb*(xpart v_u));
  c_b - c_ = c_ - c_t;
  p1 := c_b--z$$--c_t--cycle;
  filldraw p1;
  p2 := p1 shifted ((xpart c_ - x$$), (ypart c_ - y$$));
  filldraw p2;
  endgroup
enddef;

```

drawOD Draw an open diamond, length `gdl` and base width `gdb`, at end of vector from `z$` to `z$$`.

```
%%% draws an open diamond at end of vector from $ to $$
def drawOD(suffix $, $$) =
  begingroup
  save v_, c_, c__, v_u, c_t, c_b, l, hb, p;
  pair v_, c_, c__, v_u, c_t, c_b;
  numeric l, hb;
  path p;
  hb := gdb/2;
  v_ := z$-z$$;
  l := (xpart v_)++(ypart v_);  % length of the line
  c_ := (gdl/(2l))[z$$,z$];     % base of diamond
  c__ := (gdl/(l))[z$$,z$];     % interior tip of diamond
  v_u := unitvector v_;
  c_t := c_ shifted (-hb*(ypart v_u), hb*(xpart v_u));
  c_b - c_ = c_ - c_t;
  p = c_b--z$$--c_t--c__--cycle;
  unfill p;
  draw p; draw p;
  endgroup
enddef;

```

drawCD Draw a closed diamond, length `gdl` and base width `gdb`, at end of vector from `z$`

to `z$$`.

```
%%% draws a closed diamond at end of vector from $ to $$
def drawCD(suffix $, $$) =
  begingroup
  save v_, c_, c__, v_u, c_t, c_b, l, hb, p;
  pair v_, c_, c__, v_u, c_t, c_b;
  numeric l, hb;
  path p;
  hb := gdb/2;
  v_ := z$-z$$;
  l := (xpart v_)++(ypart v_);  % length of the line
  c_ := (gdl/(2l))[z$$,z$];     % base of diamond
  c__ := (gdl/(l))[z$$,z$];     % interior tip of diamond
  v_u := unitvector v_;
  c_t := c_ shifted (-hb*(ypart v_u), hb*(xpart v_u));
  c_b - c_ = c_ - c_t;
  p = c_b--z$$--c_t--c__--cycle;
  filldraw p;
  endgroup
enddef;

```

`drawdashA` Draw a straight dashed line between the two points `z$` and `z$$`, ending with an arrowhead, length `gal` and base width `gab`, at `z$$`.

```
%%% draws a dashed line from $ to $$, ending with an arrowhead at $$
def drawdashA(suffix $, $$) =
  draw z$--z$$ dashes;
  drawA($, $$);
enddef;

```

`drawdashOA` Draw a straight dashed line between the two points `z$` and `z$$`, ending with an open arrowhead, length `gal` and base width `gab`, at `z$$`.

```
%%% draws a dashed line from $ to $$, ending with an open arrowhead at $$
def drawdashOA(suffix $, $$) =
  draw z$--z$$ dashes;
  drawOA($, $$);
enddef;

```

`drawnormalDCA` Draw a straight normal line between the two points `z$` and `z$$`, ending with double closed arrowheads, each length `gal` and base width `gab`, at `z$$`.

```
%%% draws a normal line from $ to $$, ending with double closed arrowheads at $$
def drawnormalDCA(suffix $, $$) =
  draw z$--z$$;
  drawDCA($, $$);
enddef;

```

`drawnormalOD` Draw a straight normal thickness line between the two points `z$` and `z$$`, ending with an open diamond, length `gdl` and base width `gdb`, at `z$$`.

```
%%% draws a normal line from $ to $$, ending with an open diamond at $$
def drawnormalOD(suffix $, $$) =
  draw z$--z$$;
  drawOD($, $$);
enddef;

```

`drawnormalCD` Draw a straight normal thickness line between the two points `z$` and `z$$`, ending with a closed diamond, length `gdl` and base width `gdb`, at `z$$`.

```
%%% draws a normal line from $ to $$, ending with a closed diamond at $$
def drawnormalCD(suffix $, $$) =
  draw z$--z$$;
  drawCD($, $$);
enddef;

```

`drawdashcircle` Draw a normal thickness dashed open circle, center `z$`, diameter `diam`.

```
%%% draws an open dashed circle, center z$, diameter diam
def drawdashcircle(suffix $)(expr diam) =
  circpoints($, diam);
  draw z$bl..z$bm..z$br..z$mr..z$tr..z$tm..z$tl..z$ml..cycle dashes;
enddef;

```

`drawcircleA` Draw a normal thickness open circle, center `z$`, diameter `diam`, and with a counterclockwise pointing arrow at the topmost point. The arrow has length `gal` and base width `gab`.

```
%%% draws an open circle, center z$, diameter diam, with an arrow at the top
def drawcircleA(suffix $)(expr diam) =
  circpoints($, diam);
  begingroup
  save c_, c_t, c_b, hb;
  pair c_, c_t, c_b;
  numeric hb;
  hb := gab/2;
  c_ := z$tm shifted (gal*right);  % base of arrowhead
  c_t := c_ shifted (hb*up);
  c_b := c_ shifted (hb*down);
  draw z$bl..z$bm..z$br..z$mr..z$tr..z$tm..z$tl..z$ml..cycle;
  draw c_t--z$tm--c_b;
  endgroup
enddef;

```

`drawDot` Draw a black dot, center `z$`, diameter `diam`.

```
%%% draws black dot, center z$, diameter diam
def drawDot(suffix $)(expr diam) =
  begingroup
    save p;
    path p;
    p := fullcircle scaled diam shifted z$;
    filldraw p;
  endgroup
enddef;

%    \nd{macrocode}
% \end{routine}
%
% \begin{routine}{drawCircledDot}
% Draw a black dot, center |z$|, surrounded by a circle, overall
% diameter |diam|.
% \changes{v1.6}{2004/02/29}{Added drawCircledDot}
%    \begin{macrocode}
%%% draws black dot surrounded by a circle, center z$, diameter diam
def drawCircledDot(suffix $)(expr diam) =
  begingroup
    save l_, p;
    numeric l_;
    path p[];
    l_ := 5/7diam;
    p1 := fullcircle scaled diam shifted z$;
    unfill p1;
    draw p1;
    p2 := fullcircle scaled l_ shifted z$;
    filldraw p2;
  endgroup
enddef;

%    \nd{macrocode}
% \end{routine}
%
%
% \begin{routine}{drawcardbox}
% \changes{v1.1}{1999/10/30}{Added drawcardbox routine}
%  Draw a rectangular box that has its top righthand corner folded down.
% |m| is the height/length of the fold.
%    \begin{macrocode}
def drawcardbox(suffix $)(expr l, h, m)(text str) =
  rectpoints($, l, h);
  begingroup
  save c;
  pair c[];
  c1 = (x$tr-m, y$tr);
  c3 = (x$tr, y$tr-m);
  c2 = (xpart c1, ypart c3);
  draw z$bl--z$br--c3--c1--z$tl--cycle;
  draw c1--c2--c3;
  endgroup;
  z$c = 1/2[z$ml,z$mr];
  label(str, z$c);
enddef;

```

drawdiamondbox Draw a diamond shaped box. Like circles, the box is located at its center point and not at the bottom left corner (see Figure 8).

```
def drawdiamondbox(suffix $)(expr l, h)(text str) =
  z$ml=(x$-l/2, y$); z$tm=(x$, y$+h/2);
  z$mr=(x$ml+l, y$); z$bm=(x$, y$tm-h);
  z$bl=1/2[z$ml,z$bm];
  z$br=1/2[z$bm,z$mr];
  z$tr=1/2[z$mr,z$tm];
  z$tl=1/2[z$tm,z$ml];
  z$c=z$;
  draw z$ml--z$bm--z$mr--z$tm--cycle;
  label(str, z$c);
enddef;

```

drawtwodiamondbox Draw a diamond shaped box with a smaller diamond inside.

```
def drawtwodiamondbox(suffix $)(expr l, h, mrg)(text str) =
  z$ml=(x$-l/2, y$); z$tm=(x$, y$+h/2);
  z$mr=(x$ml+l, y$); z$bm=(x$, y$tm-h);
  z$bl=1/2[z$ml,z$bm];
  z$br=1/2[z$bm,z$mr];
  z$tr=1/2[z$mr,z$tm];
  z$tl=1/2[z$tm,z$ml];
  z$c=z$;
  begingroup
  save v_, p, tl, sf;
  pair v_;
  numeric tl, sf;
  path p[];
  p1 = z$ml--z$bm--z$mr--z$tm--cycle;
  draw p1;
  v_ := z$c-z$tr;
  tl := (xpart v_)++(ypart v_);
  sf = 1.0 - mrg/tl;
  p2 = p1 shifted -z$c scaled sf shifted z$c;
  draw p2;
  endgroup;
  label(str, z$c);

enddef;

```

drawdoublerectangle Draw a double box where `tf` is the fraction of the height of the top portion. The ends of the dividing line are `z$tfl` and `z$tfr`. The text centers are `z$ct` and `z$cb` for the top and bottom portions.

```
def drawdoublerectangle(suffix $)(expr l, h, tf) =
  rectpoints($, l, h);
  z$tfl=tf[z$tl,z$bl]; z$tfr=tf[z$tr,z$br];
  z$cb=1/2[z$bl,z$tfr];
  z$ct=1/2[z$tfl,z$tr];
  draw z$bl--z$br--z$tr--z$tl--cycle;
  draw z$tfl--z$tfr;
enddef;

```

`drawtriplerectangle`

Draws a triple box, where `tf` is the fraction of the height for the top portion and `bf` is the fraction of the height for the bottom portion. Text centers for the three portions are `z$ct`, `z$cm` and `z$cb` for the top, middle and bottom portions. The points at the ends of the dividing lines are `z$tfl` and `z$tfr` for the top portion and `z$bfl` and `z$bfr` for the bottom.

```
def drawtriplerectangle(suffix $)(expr l, h, tf, bf) =
  rectpoints($, l, h);
  z$tfl=tf[z$tl,z$bl]; z$tfr=tf[z$tr,z$br];
  z$bfl=bf[z$bl,z$tl]; z$bfr=bf[z$br,z$tr];
  z$cb=1/2[z$bl,z$bfr];
  z$cm=1/2[z$bfl,z$tfr];
  z$ct=1/2[z$tfl,z$tr];
  draw z$bl--z$br--z$tr--z$tl--cycle;
  draw z$bfl--z$bfr;
  draw z$tfl--z$tfr;
enddef;

```

`hiderectangle`

Draws an invisible rectangular box of the usual dimensions that covers up anything underneath it.

```
def hiderectangle(suffix $)(expr l, h) =
  begingroup
  save c;
  pair c[];
  c1=(x$,y$);
  c2=c1+(l,0);
  c3=c1+(l,h);
  c4=c1+(0,h);
  unfilldraw c1--c2--c3--c4--cycle;
  endgroup
enddef;

```

`drawdashboxover`

Draws a dashed box that covers up anything underneath it.

```
def drawdashboxover(suffix $)(expr l, h) =
  rectpoints($, l, h);
  hiderectangle($, l, h);
  z$c = 1/2[z$bl,z$tr];
  draw z$bl--z$br--z$tr--z$tl--cycle dashes;
enddef;

```

`drawindexbox` Draws an index box. The main box is `l` by `h` and the small box at the top left is `lp` by `hp`. The main box points are the usual `z$bl` etc, but the top box points are `z$P.bl` etc. The `str` is put at the center (`z$P.c`) of the small box.

```
def drawindexbox(suffix $)(expr l, h, lp, hp)(text str) =
  rectpoints($, l, h);
  z$c = 1/2[z$bl,z$tr];
  z$P = z$tl;
  rectpoints($P, lp, hp);
  z$P.c = 1/2[z$P.bl, z$P.tr];
  draw z$bl--z$br--z$tr--z$tl--cycle;
  draw z$P.bl--z$P.br--z$P.tr--z$P.tl--cycle;
  label(str, z$P.c);
enddef;

```

`drawroundedbox` Draws a rectangular box with rounded corners (like the LaTeX `\oval`). The box is `l` by `h` and located by the bottom left corner. The corners are rounded with a radius of `r`. If the radius is too large for the box it is reduced so that at least two opposite sides are semi-circular. The `str` is put at the center (`z$c`) of the box.

```
def drawroundedbox(suffix $)(expr l, h, r)(text str) =
  rectpoints($, l, h);
  begingroup
  save rad;
  numeric rad; rad := r;
  if rad > l/2:
    rad := l/2;
  fi
  if rad > h/2:
    rad := h/2;
  fi
  draw (x$br-rad, y$br){right}..{up}(x$br, y$br+rad)--
       (x$tr, y$tr-rad){up}..{left}(x$tr-rad, y$tr)--
       (x$tl+rad, y$tl){left}..{down}(x$tl, y$tl-rad)--
       (x$bl, y$bl+rad){down}..{right}(x$bl+rad, y$bl)--cycle;
  endgroup;
  z$c = 1/2[z$bl,z$tr];
  label(str, z$c);
enddef;

```

`drawovalbox` Draws an elliptical box with horizontal diameter `l` and vertical diameter `h`. The box is located by the center point.

```
def drawovalbox(suffix $)(expr l, h)(text str) =
  ellipsepoints($, l, h);
  z$c = 1/2[z$ml,z$mr];
  begingroup
  save p;
  path p;
  p = fullcircle scaled h xscaled (l/h) shifted z$c;
  draw p;
  endgroup;
  label(str, z$c);
enddef;

```

`drawdashellipse` Draws a dashed elliptical box with horizontal diameter `l` and vertical diameter `h`. The ellipse is located by the center point.

```
def drawdashellipse(suffix $)(expr l, h) =
  ellipsepoints($, l, h);
  z$c = 1/2[z$ml,z$mr];
  begingroup
  save p;
  path p;
  p = fullcircle scaled h xscaled (l/h) shifted z$c;
  draw p dashes;
  endgroup
enddef;

```

`drawdrum` Draws a drum box with length `l` and height `h`. The top and bottom ellipse minor/major diamter ratio id `drumlid`. The text is put at the (curved) center of the drum.

```
def drawdrum(suffix $)(expr l, h)(text str) =
  save vdia; numeric vdia;
  vdia := drumlid*l;                % ellipse vertical diameter
  save pf, ph; path pf, ph;         % full & half ellipse paths

  % points on the basic rectangle
  z$bl = z$;
  z$tr = (x$+l, y$+h);
  z$br = (x$tr, y$);
  z$tl = (x$, y$tr);
  z$ml = 1/2[z$bl,z$tl];
  z$mr = 1/2[z$br,z$tr];
  z$tc = 1/2[z$tl,z$tr];    % center of top rectangle line
  z$bc = 1/2[z$bl,z$br];
  % draw box sides
  draw z$tl--z$bl; draw z$tr--z$br;

  % points on top ellipse
  z$T’’’ = z$tc;            % ellipse center
  ellipsepoints($T’’’, l, vdia);
  z$tm  = z$T'''.tm;
  z$tml = z$T'''.tl;
  z$tmr = z$T'''.tr;

  % points on bottom ellipse
  z$B''' = z$bc;
  ellipsepoints($B''', l, vdia);
  z$bm  = z$B'''.bm;
  z$bml = z$B'''.bl;
  z$bmr = z$B'''.br;

  % box center point
  z$c = 1/2[z$T'''.bm, z$B'''.bm];

  % draw top ellipse
  pf = fullcircle scaled vdia xscaled (l/vdia) shifted z$T''';
  draw pf;
  % draw bottom half ellipse
  ph = (halfcircle rotated 180) scaled vdia  xscaled (l/vdia) shifted z$B''';
  draw ph;
  label(str, z$c);
enddef;

```

`drawoutputbox` Draws an output box. The box is `l` by `h` and the `str` is put at the center. The bottom of the box is a wavy line.

This code was supplied by Guy Worthington (see `comp.text.tex` newsgroup thread *Trial, ignore*, January 2004, and in particuar Guy's message of 2004/01/27).

```
def drawoutputbox(suffix $)(expr l, h)(text str) =
  rectpoints($, l, h);
  begingroup
    save c;
    pair c[];
    c1 = (x$br, y$br+1/8h); % right side of box is shorter
    c2 = (x$bm, y$bm+1/16h);
    % draw the tear
    draw z$bl..c2{dir 45}..c1{dir -15}--z$tr--z$tl--cycle;
  endgroup;
  z$c = (x$bm, 1/2(y$bm+y$tm));
  label(str, z$c);
enddef;

```

`drawstickman` Draws a full frontal genderless stick figure, inside a rectangle `l` by `h`.

```
\def drawstickman(suffix $)(expr l, h) =
  rectpoints($,l,h);
  begingroup
    save c;
    pair c[];
    c1 = 8/24[z$bm,z$tm];
    c2 = 15/24[z$bm,z$tm];
    c3 = 18/24[z$bm,z$tm];
    c4 = 1/2[c3,z$tm];
    c6 = (x$bl, ypart(c2));
    c7 = (x$br, ypart(c2));
    draw z$bl--c1--z$br;             % legs
    draw c1--c3;                     % body
    draw c6--c7;                     % arms
    draw c3{right}..z$tm{left}..cycle; % head
  endgroup;
enddef;

```

The end of the package

```
⟨/up⟩
```

References

[EN89] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin Cummings Publishing Co. Inc., 1989.

[GMS94] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley Publishing Company, 1994.

[GRM97] Michel Goossens, Sebastian Rahtz, and Frank Mittelbach. *The LaTeX Graphics Companion*. Addison-Wesley Publishing Company, 1997.

[Hob92] John D. Hobby. *A user's manual for MetaPost*. Computing Science Technical Report no. 162, AT&T Bell Laboratories, Murray Hill, NJ, April 1992. (Available from CTAN with the MetaPost distribution in `.../graphics/metapost`).

[Hoe98] Alan Hoenig. *TeX Unbound*. Oxford University Press, 1998.

[IDE85] AFWAL/MLTC, Wright-Patterson AFB, OH. *Integrated Informatiuon Support Systems (IISS), Vol. V: Common Data Model Subsystem, Part 4: Information Modeling Manual — IDEF1X*. Report Number: AFWAL–TR–86–4006, Volume V, 1985.

[ISO87] ISO TR9007. *Information processing systems — Concepts and terminology for the conceptual schema and the information base*, 1987.

[ISO94] ISO 10303-11:1994. *Industrial automation systems and integration — Product data representation and exchange — Part 11: Description methods: The EXPRESS language reference manual*, 1994.

[Knu86] Donald E Knuth. *The METAFONTbook*. Addison-Wesley Publishing Company, 1986.

[Lam94] Leslie Lamport. *LaTeX: A Document Preperation System.* Second edition. Addison-Wesley Publishing Company, 1994.

[NH89] G. M. Nijssen and T. A. Halpin. *Conceptual Schema and Relational Database Design*. Prentice Hall, 1989.

[Rec97] Keith Reckdahl. *Using EPS Graphics in LaTeX2e Documents*. February 1997. (Available from CTAN as `../info/epslatex.ps`).

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. *Object-Oriented Modeling and Design* Prentice Hall, 1991.

[SW94] Douglas A. Schenck and Peter R. Wilson. *Information Modeling the EXPRESS Way*. Oxford University Press, 1994. (ISBN 0-19-508714-3)

[SM88] S. Shlaer and S. J. Mellor. *Object-Oriented System Analysis*. Yourdon Press, 1988.

[Wil99] Peter Wilson. *Some Experiences in Running METAFONT and MetaPost*. November, 1999. (Available from CTAN as `../info/metafp.ps`).

Note: See `http://www.tug.org` for information on accessing CTAN — the Comprehensive TeX Archive Network.

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.