



Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Graphics for ABC

J. Zwaan, R. Zwart

Computer Science/Department of Algorithmics and Architecture

CS-R9255 1992

Graphics for ABC

Jaap Zwaan and Rolf Zwart

CWI

P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

&

IHBO De Maere

P.O. Box 1075, 7500 BB Enschede, The Netherlands

Abstract

This report sets the first steps towards a graphical facility for the programming language ABC. It discusses which features are to be included as primitives in a graphical extension to the language, the way pictures could be represented and gives directions towards an implementation.

1991 Mathematics Subject Classification: 68N15, 68Q50, 68U05.

1991 CR Categories: I.3.0, I.3.4, I.3.6, H.5.2, H.1.2.

Keywords and Phrases: Computer graphics, graphics utilities, programming languages, user interfaces, ergonomics.

Note: Work performed during a stay at the CWI, Amsterdam.

1 Introduction

This report discusses a graphical extension to the programming language ABC the final version of a sequence of languages called *B*. An informal introduction to ABC can be found in [2] and a more technical introduction in [8].

ABC is a programming language designed for personal computing and suited for the non-professional programmer, although the professional will also find ABC worth using. It is a simple language that supports structured programming. The design objectives for ABC were:

- ◆ simplicity;
- ◆ suitability for conversational use;
- ◆ inclusion of structured-programming tools.

From the start of the design, ABC was not envisaged as an isolated programming language, but as part of an environment dedicated to the ABC programmer. Such an environment supports tools to assist the user in making the most out of their computer in an easy and powerful way. One of these tools will be a graphical package, integrated with the language and the environment.

1.1 Design objectives

The same design objectives underlying ABC should apply to its graphical package. Graphics for ABC is designed as an extension to the programming language ABC. This makes it possible to create and manipulate 'pictures' with an ABC program in an easy way. The features discussed here only deal with non-interactive use. Like any other ABC object, it should be possible also to create or modify pictures interactively with the standard editor of the ABC environment, similar to MacPaint [5] and Cip [11]. Clearly, this requires some extensions to the capabilities of the currently existing editor. This, however, falls outside the scope of this report. It should also be possible to receive interactive data input in ABC programs via a graphical interface (e.g., to write menu-driven programs). A proposal for a set of primitives for such a facility is given in [1].

One of the design objectives is simplicity: the creation of pictures must be easy. It is only possible to judge the merits of alternatives in the design process if one has a clear idea of the kind of pictures the graphics facility is aiming to support. The assumption is here that these pictures have mainly the nature of diagrams: two-dimensional charts, or schemas, built from simple geometrical figures that could be drawn by an artist equipped with a line-drawing pen, possibly adorned with text. In this respect, the scope of the facility described here is the same as that of Ideal [12] and Pic [7]. Thus no attention is paid to the problem of representing and rendering three-dimensional objects; although many of the primitives proposed can be given an obvious meaning in a 3D context, the philosophy here has been that the inevitable complications caused by possible excursions into higher dimensions should not burden in any way the user who is only interested in planar pictures. Before the present 2D proposal is finalised, it may nevertheless be worth spending some effort in considering the design of a 3D extension, since it is conceivable that a future addition in that direction could be facilitated by modifications to the current 2D primitives without adding complexity. It would be a pity, then, if possible future extensions, even if not envisaged now, would be thwarted unduly by earlier commitments.

Another matter entirely out of reach of the proposal in this report is that of animation. The envisioned properties of the full ABC environment, in which representations on the screen of objects modified by a running program will be continually updated, should supply a limited form of animation. Real-time control of animation is currently out of the question, and nothing like a video arcade game could therefore be programmed.

1.2 Operational versus object-centred approach

The traditional approach to graphics in programming languages is *operational*: pen down, move to here, pen up, move to there, and so on. The conceptual entities from which a picture is composed are hidden in this approach, and simple modifications to a picture may require a major rewrite of the program. These quite low-level primitives may be replaced by somewhat higher ones (draw a line from here to there), or still higher ones, leading to extensive packages such as GKS [4].

Still, the operational approach tends to force the user to be overly concerned with coordinates, and is in general rather inflexible. For example, suppose a user has defined a procedure for drawing bar charts. To draw two charts next to each other, the user has to exert extreme care in positioning the charts to prevent overlapping. In some way, the information about the width of the charts is present in their procedural descriptions, but it is not available as such to the program. The procedure for bar chart drawing can be amended to return the width of the chart just drawn, which may depend on the number of entries in the table represented in the chart, and also on the length of the texts representing the table keys. However, this information is then available only *after* a chart has been drawn. This means that this amendment still does not make it possible to let the program centre the picture composed of the two charts.

These problems can be overcome with a *object-centred* approach. Here the act of *creating* a picture and that of *drawing* it on the screen are separated: a picture is a value that can be created and modified by a program. This makes it possible to define, for example, a picture as consisting of two sub-pictures next to each other, or of another picture with a box around it.

This philosophy of pictures has been adopted from ILP [3]. The primitives of ILP are rather primitive compared to the conceptual entities comprising the kind of diagrams for which the ABC graphics facility is intended. A better model in this respect is provided by Ideal and Pic, for example; these packages largely obviate the necessity of undue user concern with coordinates. However, in contrast to an ILP picture, which is a structured object, an Ideal or Pic 'picture' is a sequence of symbols, a linearized representation of a structured description of a drawing that has to be parsed to retrieve the structure; building a picture from components entails low level textual operations. Thus, Ideal and Pic, although more felicitous in the level of their primitives, are less suitable if it comes to the algorithmic creation or manipulation of pictures (nor were they designed with such aims in mind).

The present proposal attempts to combine the best of these two worlds: the object-creation approach of ILP with the user-friendly high level of Ideal and Pic.

2 Other graphical languages and packages

2.1 An overview

MacPaint

MacPaint [5] is a high-level graphic package for the Macintosh. Nearly all the drawing is done with a mouse and a little help from the keyboard. The largest positive difference between MacPaint and the other graphical packages described in this section, is that MacPaint makes use of a mouse and is interactive. Some disadvantages of MacPaint are:

- ◆ Its absence of exact positioning, which makes it hard to position a drawing at a precise position (the user has to do that with a mouse).
- ◆ The user cannot undo something she did two or more steps before. There is no data structure but a pixel map, so structured editing is impossible.

4 Graphics for ABC

- ◆ Parts of pictures are only accessible to other programs as long as they are not printed under or over something else.
- ◆ Using input from a program for painting a picture is impossible.

It is the only package that has no programming language at all.

Pic

Pic [7, 6] is a language for specifying pictures so that they can be typeset as an integral part of a document preparation system: it is meant as a preprocessor for Troff [9]. The basic objects in Pic are boxes, lines, arrows, ellipses, arcs and splines, which may be placed anywhere and labeled with arbitrary text. An advantage of Pic is that it is possible to specify the sizes and positions of objects with minimal use of absolute coordinates. But there are also some negative points: Pic is useful for drawing flowcharts, but it takes a long description to specify a more complex drawing, e.g. a directed graph. An interactive program Cip [11] exists, that facilitates drawing in a way similar to MacPaint and generates a — quite readable — Pic-file.

Ideal

Like Pic, Ideal [13] is a preprocessor for Troff. The greatest disadvantage of Ideal is its principal theorem: *complex numbers are good*. This fact excludes nearly all ABC users. Studying the language takes a lot of time before one can describe even a simple picture. But as soon as the language is mastered, it is powerful. A program written in Ideal includes a system of constraints that declares the relative positions of its significant points and requests for actions to be performed at those points. Two commands embody the idea of sketching several pictures on different parallel planes, then merging them into a single picture. This is used when otherwise creating one part of a picture would destroy another part.

Graphical Kernel System (GKS)

GKS [4] is a graphics system that allows programs to support a wide variety of graphic devices. It is defined independently of programming languages. Before it can be used from a particular language, a *language binding* must be defined for that language. GKS does not make use of relative positioning within a segment, so it is difficult for non-experienced people to deal with the language. Its low level primitives are powerful; in particular its primitives for text are elaborate.

Intermediate Language for Pictures (ILP)

ILP [3] is a special purpose data description language: it is intended solely for the description of pictures and to fill the gap between instructions for drawing on the one hand and a picture description as part of a more sophisticated language or data structure for an application area on the other. ILP is a low-level language in the sense that for each feature required the simplest construction is chosen.

2.2 Capabilities in other languages

What should be included in the graphical package for ABC? In this section is discussed which primitives other packages use. We have distributed all these primitives over three categories:

- ◆ **Basic primitives** are the fundamental building blocks of a picture.
- ◆ **Modification primitives** alter the picture in one way or another:
 - extend the picture with an enclosing object, e.g. a box (*container* primitives);

- alter aspects like position, drawing mode, shading and colour (*appearance* primitives);
 - change the whole picture by rotating, mirroring or scaling it.
- ♦ **Composition primitives** compose a picture out of other pictures.

Basic primitives

These primitives take a value as an argument and make a picture out of it.

Point

Points seem to be the most obvious drawing primitive, but only MacPaint has one; it will probably not be used frequently.

Line

Drawing a line is one of the simplest primitives. The user only has to specify two points in some way:

- ♦ In MacPaint the user marks the two endpoints of the line with the mouse and it draws a line between them.
- ♦ In GKS is it harder to draw a line, because GKS only features polylines; so the user has to specify an array consisting of two coordinates. These coordinates need to be absolute, because GKS has no facility for relative positioning.
- ♦ Pic does not supply absolute positioning; it only supplies directions and compass points of a picture. This is fine to work with:
 - the user can specify, e.g., `line from C.north to C2.south` where `C` and `C2` are pictures. This has the advantage that the user need not guess what the exact point is.
 - Another way of specifying a line in Pic is: `line up 1 inch line right 2 inch.`
 - The third way is related to the first, but one can reference a picture just by its name and Pic will calculate the best looking point on that picture for one of the ends of that line.
- ♦ ILP caters for absolute points (called FIXED) and moving in a certain direction (called FREE, this is like `line up 1inch right 2inch` in Pic), but there is no relative positioning.
- ♦ Ideal draws a line between two points, a point being described by its coordinates.

Arrow

It is difficult to specify arrows by means of other primitives, because the head of the arrow needs to be properly oriented about the shaft of the arrow, no matter what angle the arrow points.

- ♦ MacPaint does not cater for arrows at all, but a small head of an arrow is easily entered in 'fatbits' mode.
- ♦ GKS and ILP have no arrows. Pic does supply arrows. These have a default length and direction which can be changed easily, like lines can.
- ♦ Ideal has no facility for arrows, but they can be defined by means of the equations given in [12].

Text

This is a common primitive. All the packages supply a way to include text in a picture and some have a rather elaborate set of these primitives.

- ◆ MacPaint supplies left, right and middle alignment as well as various point sizes and fonts. A difficulty with MacPaint is its positioning: the user needs to position the pointer carefully and has to undo and try again many times. MacPaint has no primitive for centring several lines one under the other, except when the user types them in at one time.
- ◆ GKS supplies left, middle, top and bottom alignment, writing from right to left, angled text, multiple fonts, multiple sizes and multiple styles.
- ◆ Since Pic and Ideal are both preprocessors for Troff, they need not supply text primitives, because Troff itself can take care of that.
- ◆ ILP has low level text primitives: it only supplies standard characters in standard sizes.

Rectangle or box

There are two different primitives for making a rectangle. One creates a rectangle of a certain width and length (sometimes there is a default) and the other places a rectangle around some existing picture (this is usually called box ing).

- ◆ In MacPaint you draw rectangles by positioning the mouse in one corner, dragging it to another corner and then releasing the button.
- ◆ Pic can make a good fitting rectangle around text or other picture, called BOX. It can also draw rectangles of a default size or any size the user wants.
- ◆ Ideal supplies a primitive called box , but the sizes must always be specified .

Circle and ellipse

Circles and ellipses are hard to construct by means of other primitives, and therefore need to be a primitive.

- ◆ MacPaint supplies circles and ellipses; the user only has to specify two points and MacPaint draws the largest possible ellipse or circle that will fit in a rectangle between the two points; the circle will be drawn adjusted to the first point.
- ◆ Pic draws ellipses (circles) in the same way as it draws rectangles (squares): either drawing a circle around something or by specifying its dimensions.
- ◆ A circle is a primitive in Ideal (the description of a circle is given in [13]).

Arc

- ◆ Only Pic and Ideal support arcs. In Ideal the user has to specify a centre, a radius and starting and ending angles, whereas in Pic she has to specify the direction (clockwise or counterclockwise), starting and ending points and a radius (if the radius is omitted, a radius of 1 inch is assumed).

Polygon and polyline

A polyline is a figure consisting of one or more lines, each line starting at the end point of the previous one. A polygon is a polyline having an extra line from the last to the first point.

- ◆ In MacPaint the user only has to click on the desired positions and MacPaint will draw a polyline or a polygon (depending on what was specified).

- ◆ GKS draws a polyline or polygon with the aid of an array consisting of positions.

Spline

Splines are smooth curves passing through specified points. Only Pic and Ideal supply splines.

Modification primitives

These primitives take a picture as their argument and result in a modified picture.

Box

A box is a rectangle around some other picture; boxes were discussed in the previous section under rectangle.

Line mode

A line can be drawn in several ways, e.g. with dots, dashes, solid or some predefined pattern.

- ◆ All graphical packages supply at least two drawing modes. MacPaint also supplies thin and fat.

Colouring and shading

Colouring can only be done if an output device provides colours. Shading is easy, but the big problem is how to specify a pattern and the object to be shaded.

- ◆ Colouring is not implemented in MacPaint, because its output devices (a printer or terminal) do not feature it, whereas shading is easy in MacPaint. The user only has to be sure that there are no gaps or holes in the outline of the wanted area, otherwise the pattern will 'leak through'. The user cannot shade in a certain painted object, the user can only shade an outlined area, because MacPaint does not cater for the underlying data structure; it only knows of pixels being white or black.
- ◆ GKS features colouring and shading in and does this in a nice way (it recognises the data structure).
- ◆ Pic and Ideal do shading as the user expects it to be done.

Rotation, mirroring and scaling

Rotation means turning a picture around some point, mirror means exchanging left and right or top and bottom, and scaling means increasing or decreasing the sizes horizontally or vertically or both.

- ◆ MacPaint can only rotate by 90°. Scaling can be done for some specified field (only a rectangle) horizontally and vertically independently.
- ◆ GKS can rotate pictures by any angle and scale horizontally and vertically independently.

Composition primitives

Once the user has created various pictures, she might like to create a new picture existing of the previously created pictures. For example, she has described a tree and a duck and now wants the duck to sit under the tree.

Combine

Here, combining (merging) means to draw one picture upon the other without leaving out certain parts.

- ♦ With MacPaint this is very complicated, but it is possible.
- ♦ In GKS this can be done by specifying the new position of the object to be dropped in the picture.

Overlay

Overlaying means drawing a picture over another and removing the parts that would be under the picture: underlying parts of previously painted pictures will disappear.

- ♦ MacPaint can do this, but it is not easy, mainly because MacPaint does not recognise the structure of the drawing. The picture to be painted over the other must be selected with a surrounding rectangle, so everything outside the wanted picture, but inside the rectangle will be considered part of the picture. Therefore more may disappear than wanted. This can be avoided by selecting the lasso (for selecting nonrectangular things), but the user has to be precise: it is hard to move the lasso around the outline.

3 Graphics primitives for ABC

In this section, possible graphics primitives for ABC are examined. The first section contains a comprehensive list of candidate primitives, all enumerated for discussion. In the second section the candidates list is pruned: a selection of these primitives is made and it is explained why others are left out or are transformed into a different shape; where necessary, the semantics of a primitive are refined.

3.1 Candidate primitives

Basic primitives

The position of the origin of the pictures returned by these primitives will be discussed in the next section.

Dot

dot pos

A dot at position **pos**.

dot

This is a zeroadic function that returns a dot at (0, 0). To position this dot use **at(pos, dot)**, which has the same effect as **dot pos**.

Line and arrow

line (pos1, pos2)

Return a line from **pos1** to **pos2**.

line pos

Return a line from (0,0) to **pos**.

line (pic1, pic2)

Return a line from **pic1** to **pic2** in a neat way (from border to border).

arrow (pos1, pos2)

Return an arrow from **pos1** to **pos2**.

arrow pos

Return an arrow from (0,0) to **pos**.

arrow (pic1, pic2)

Return an arrow from **pic1** to **pic2** in a neat way (from border to border).

Text

| | |
|-----------------|---|
| text | Return the text at the given position. |
| text expression | Return the expression, converted to text with its origin at (0, 0). |

Rectangle and square

| | |
|------------------------|---|
| rectangle (pos1, pos2) | Return a rectangle with corners at the specified points. |
| rectangle (x,y) | Return a rectangle with width and height with its origin at (0, 0). |
| square size | Return a square of the specified size. |
| square (pos, size) | Return a square of size with origin positioned at pos . |

Circle and ellipse

| | |
|----------------------|---|
| circle radius | Return a circle with radius and middle point at (radius , radius). |
| circle (pos, radius) | Return a circle with radius with centre pos . |
| ellipse (pos1, pos2) | Return the largest ellipse that would fit in rectangle (pos1, pos2) . |
| ellipse (x, y) | Return an ellipse with width x and height y with its origin at (0, 0). |

Arc

| | |
|---------------------------|---|
| arc (centre, pos1, pos2) | Return a counterclockwise circular arc with centre at centre , starting point at pos1 and end point on the line from centre through pos2 . |
| arc (radius, pos1, angle) | Return a counterclockwise circular arc with centre at (0, 0), starting point at pos1 and the specified angle. |
| arc(centre, pic1, pic2) | Return a counterclockwise circular arc with centre at centre , starting point at pic1 and end point at pic2 determined in the same way as with line and arrow. |

Polygon and polyline

| | |
|--------------|--|
| polyline seq | Return lines between all successive points in seq (absolute coordinates). |
| polygon seq | Return lines between all points in seq and one extra line from the first to the last point. |

Curve and loop

| | |
|-----------|---|
| curve seq | Return a smooth curve (often called spline), connecting successive points in seq . |
| loop seq | Return a smooth curve through successive points and through the first and last point. |

Modification primitives**Container primitives**

| | |
|---------|---|
| box pic | Return pic with the smallest possible box around it. |
|---------|---|

| | |
|---------------------------------------|---|
| circle pic | Return pic with the smallest possible circle around it. |
| ellipse pic | Return pic with the smallest possible ellipse around it. |
| square pic | Return pic with the smallest possible square around it. |
| hull pic | Return pic in a convex hull. |
| padded pic | Return pic in a nice, not too thick border (margin). |
| padded (pic , margin) | Return pic in a border with thickness margin . |

Positioning

| | |
|-------------------------------|--|
| at(pos , pic) | Give pic a new origin at pos . |
|-------------------------------|--|

Line mode

| | |
|------------------------------------|---|
| bold pic | Return pic drawn in boldface instead of the default solid. |
| dotted pic | Return pic drawn dotted instead of solid. |
| dashed pic | Return pic drawn dashed instead of solid. |
| drawn (pic , mode) | Return pic drawn in the specified drawing mode. |
| hidden pic | Return pic invisibly instead of solid. |

Colouring and shading

| | |
|--------|--|
| colour | Return pic painted in the specified colour col . |
| shade | Return pic shaded (filled) with pattern. |

Rotate, mirror, scale

| | |
|---|---|
| rotate(pic , angle) | Rotate pic around its origin through angle . |
| mirror pic | Mirror pic left to right. |
| mirror (pic , line) | Mirror pic around line . |
| scale (pic , factor) | Multiply the dimensions of pic with factor . |
| scale (pic , (xfac , yfac)) | Multiply the dimensions of pic with a factor xfac and yfac in x and y directions respectively. |

Remarks

- ◆ There are other conceivable modes of drawing e.g.: thin, dotdashed, longdashed, shortdashed, dotdotdashed, but it is not convenient to predefine functions for all possible modes. A problem is, how to specify a non-built-in pattern or drawing mode.
- ◆ In **hull** and **padded** should the enclosing figure be visible or invisible? Invisible will probably be desired most often, but that will give these primitives a special status: users have to remember which containers are visible and which are not. Also, there would have to be a **visible** function to make the hull or padding conspicuous.
- ◆ Some adjusting could be necessary if the obtained container has strange dimensions (a long thin box containing one line of text), but the user should also be able to turn off this automatic adjusting.
- ◆ What should the following do?

```
"purple" coloured box circle "yellow" coloured text "tie"
```

Should the circle be purple too, or should the circle have the default colour? And the text: yellow, purple or the standard colour? The same situation applies to drawing modes. There are several solutions:

- The modification always works through the entire picture.
- It works through on all deeper levels until the mode or colour is explicitly mentioned (otherwise they have the default value); a function **throughout** causes even an explicit shade, colour or mode to be overridden.
- It works through on all deeper levels until the mode or colour is explicitly mentioned; composite pictures (see next section) cause even an explicit colour or mode to be overridden, because the 'composition' can't have a colour on its own.
- It works only one level deep, but works through on default values if the picture is a composite one.
- It works only one level deep; when the effect is wanted throughout the picture, one has to repeat the action for each level, or special functions **throughout** and **default** (undoes the last **throughout**) would have to be introduced.

Ideally, you want a method that distinguishes between when the picture is already an object (like `"grey" colour mouse: entire mouse painted grey`) or is just being created (like `"grey" colour circle text "peep": only a grey circle`), but that is unfortunately not possible. The second, third and fourth alternatives look acceptable.

Composition primitives

| | |
|-------------------|---|
| pile (seq, shift) | Return a picture, containing all the pictures in seq on top of each other. Shift is a number denoting how far the 'next' picture will be displaced horizontally; -1 and +1 probably meaning the left and right border of the 'previous' picture (other numbers interpolate or extrapolate). |
| row (seq, shift) | Similar to pile , but the pictures are juxtaposed; shift denotes the vertical displacement. |
| splittable pic | Allow the program to split a picture with type row, pile or text automatically in multiple parts if it does not fit on the screen, to make it more compact. |
| combine list | Return all the pictures contained in list combined. |
| overlay seq | A special form of combine ; underlying parts of previous pictures disappear. |

3.2 Suitable primitives for ABC

Some objectives for the selection process are:

- ◆ minimize the number of functions;
- ◆ use monadic and dyadic versions wherever useful;
- ◆ choose high level functions;
- ◆ combinations of operations should be meaningful;
- ◆ there should be no unnecessary user concern with coordinates;
- ◆ appropriate defaults should be chosen;

- ♦ creating simple pictures must be simple, yet creating extensive pictures not too complicated;
- ♦ it must be user friendly, no unexpected effects should occur;
- ♦ the functions should be powerful and versatile;
- ♦ the functions should be logical and should match fair expectation;
- ♦ the functions should be easily extensible;
- ♦ the functions should have mnemonic names.

So, summarising: in the spirit of ABC.

Naming conventions

What form should the names of the functions have? One can either view the function as an action to be performed:

```
mirror that.picture
```

or consider it in the context of its occurrence in an expression:

```
DISPLAY mirrored picture
```

hence a name in the form of a past participle. The latter interpretation is attractive because in many cases it resembles natural English rather well. This usage is reflected by the following syntax:

```
expr: [specification] past-participle object
specification: prefix | number | adverb | adjective
(...etc)
```

Some functions take the form of a noun, e.g. `dot`, `ellipse`: the object to be returned. In some cases different forms are indistinguishable. One problem is, that equivalent forms of, for instance, `encircled` do not exist (`enboxed` or `boxed` or `boxed.in` could be acceptable, but `[en-]squared`? And what about `enellipsed`?). Also, long 'sentences' written in this fashion do not look like natural language any more than the alternative. For the composite primitives, it also would give ugly names: `rowed`, `overlaid` (only combined and piled seem to be acceptable), so they will keep their original form. Otherwise, wherever it does not look too far-fetched, the participle form has been chosen.

Basic primitives

| | |
|------------------------------|---|
| <code>at pos</code> | Return an 'empty picture' at pos ; enables use of positions where pictures are required. |
| <code>dot</code> | Return a dot at the origin. This zeroadic version of <code>dot</code> can easily be given a position with <code>dot at pos</code> (see modification primitives). |
| <code>pic1 line pic2</code> | Return a line from the border of pic1 to the border of pic2 . This dyadic function avoids all the extra specifications about where to start or end a line; the other versions are easily simulated using <code>at</code> |
| <code>line pic</code> | Return a line from (0,0) to pic . This line can be given a new position by <code>(line at pos2) at pos1</code> , which has the same effect as the dyadic primitive and is useful if only a direction and a starting point is specified, as in: <code>(line at (-2,-3)) at east.of some.picture.</code> Alternatively, one can use <code>polyline {pos1; pos2}</code> . |
| <code>pic1 arrow pic2</code> | Similar to <code>line</code> ; the size of the arrowhead should depend on the length of the arrow. |

| | |
|-------------------------|--|
| arrow pic | Similar to monadic <code>line</code> . |
| text expr | Where expr is an expression of any type; expr will first be converted to a text. The text is transformed into a (special sort of) picture and positioned at (0, 0). ABC already has functions on text: <code>x<<n</code> , <code>x>>n</code> and <code>x><n</code> . More elaborate functions on text (-pictures) fall outside the scope of this report, but should support font families, sizes and styles. |
| centre arc (start, end) | Return a counterclockwise arc with centre at centre , starting point at start and end point on the line from centre to end . The other arc primitives were left out because this one will be used more often, especially in combination with the additional functions as in: <code>(x, y) arc (north.of box1, south.of circle2)</code> and it is easy to simulate the other arc primitives with this one. |
| arc (start, end) | Equivalent to <code>(0, 0) arc (start, end)</code> . |
| polyline seq: | Return lines between successive points in seq . |
| polygon seq | Return lines between successive points in seq and one extra line from the first to the last point. Though it can be simulated by <code>polyline</code> , it is included for convenience because it will be used more frequently. |
| curve seq | Return a smooth curve, connecting successive points. |
| loop seq | Similar to <code>curve</code> , but connects the last to the first point in the same way as the other points. |

The `circle`, `ellipse`, `rectangle` and `square`-primitives are dealt with using modification primitives, below.

Modification primitives

| | |
|---------------------|--|
| margins box pic | Return pic with a rectangle around it; how tight it will fit is determined by margins (the margins between the smallest possible box around pic in x- and y-direction, and the present box). The box has a horizontal edge. |
| margin square pic | Return the pic with a margin in the smallest possible square. |
| margin circle pic | Return the pic with a margin in the smallest possible circle. |
| margins ellipse pic | Return the pic with margins in the smallest possible ellipse. The margins will have to be compressed at certain places to force a real ellipse to be returned. |
| margin hull pic | Return the pic with a margin in a convex polygon with rounded corners. |
| margin padded pic | Return the pic with a padding of thickness margin . |
| box pic | Equivalent to <code>nice.margins box pic</code> : a box around a picture with 'nice' looking margins. What is 'nice' depends on the [smaller dimension of the] enclosed picture and differs between the various 'containers' (it will need to be determined experimentally). |
| square pic | Equivalent to <code>nice.margin square pic</code> (margin is equal for the x- and y-direction). |

| | |
|--|--|
| <code>circle pic</code> | Equivalent to <code>nice.margin circle pic</code> . |
| <code>ellipse pic</code> | Equivalent to <code>nice.margins ellipse pic</code> . |
| <code>hull pic</code> | Equivalent to <code>nice.margin hull pic</code> . |
| <code>padded pic</code> | Equivalent to <code>nice.margin padded pic</code> . |
| <code>pic at pos</code> | Give pic a new origin at pos . |
| <code>mode drawn pic</code> | Return pic in a different drawing mode instead of (the default) solid. The mode is a text identifying the drawing mode in a, hopefully, mnemonic way: there should be a simple, intuitive meaning associated to texts defining a pattern e.g. <code>"."</code> drawn pic (dotted) or <code>"-.-"</code> drawn pic (dash-dot-dashed). Forms such as <code>"<.>"</code> drawn line (double-headed dotted arrow) and <code>"->"</code> drawn curve (curved arrow) have been suggested, but these might be prone to be generalised and certain choices would be rather arbitrary, e.g. the position of the arrowhead in <code>"->"</code> drawn (circle at anywhere). |
| <code>angle rotated pic</code> | Return pic rotated around its origin through an angle specified in radians. |
| <code>mirrored pic</code> | Return pic mirrored left to right; the other primitive will seldom be needed and can be expressed, if necessary, using <code>rotated</code> . |
| <code>(fac.x, fac.y) scaled pic</code> | Return pic with its coordinates multiplied by factors fac.x and fac.y respectively. This primitive is more general than the other one; a user can simply write a function <code>factor zoom pic</code> . |
| <code>col coloured pic</code> | Return pic painted in the specified colour col . |
| <code>pattern shaded pic</code> | Return pic shaded with a pattern . Specifying a pattern could be done in a way similar to specification of drawing modes; in fact, the pattern can be seen as the drawing mode of the 'shade'. |

Problems and remarks

- ◆ `pic1 around pic2` would be nice as a generalisation of the container primitives, but it is hard to find a reasonable algorithm.
- ◆ If a plain circle, rectangle, ellipse or square is desired, it can be specified by something like:
`radius circle (at anywhere)`.
The monadic use
`box at anywhere`
for instance, would yield a box with 'nice' sizes.
- ◆ If a single layer of padding (margin) is specified, plain rectangles, squares etc will have 'half' sizes as arguments; otherwise, the total padding must be specified. It is not clear what the most desirable property is.
- ◆ A few functions could be either predefined or user-defined:

HOW TO RETURN dotted pic: `RETURN "." drawn pic`


```

HOW TO RETURN drawn pic: RETURN "solid" drawn pic
HOW TO RETURN hidden pic: RETURN " " drawn pic
HOW TO RETURN shaded pic: RETURN "/" shaded pic
HOW TO RETURN yellow pic: RETURN "yellow" coloured pic
HOW TO RETURN coloured pic: RETURN "" coloured pic

```

but this again raises the question, where we should draw the line. It is probably best to leave it to the user, since this gives a smaller and therefore more manageable set of primitives.

- ♦ Drawing mode, colour and shading work through on all deeper levels until the mode or colour is explicitly mentioned (otherwise they have the 'weak' default value); the function `throughout pic` will cause even an *explicit* colour or mode to be overridden. In

```
PUT purple box circle yellow text 'Hallo!' IN hallo
```

the circle and the box should be purple and the text yellow, whereas in

```
"orange" coloured throughout hallo
```

all will be painted orange, no matter what colours `hallo` happens to contain.

```
"" coloured pic
```

gives `pic` the default colour; similarly for `drawn` and `shaded`. Perhaps these primitives should *always* at least affect the top level; specifying `throughout` on a one-levelled picture looks somewhat silly. Some experiments in this field need to be done.

Composition primitives

| | |
|----------------|---|
| shift pile seq | Return a picture, containing all the pictures in seq on top of each other. shift is a number denoting how far the 'next' picture will be displaced horizontally; -1 and +1 meaning the left and right border of the 'previous' picture (other numbers interpolate or extrapolate, so 0 stands for the middle and +2 denotes a position to the right of the previous picture). |
| shift row seq | Similar to <code>pile</code> , but the pictures are juxtaposed; shift denotes the vertical displacement. |
| pile seq | Equivalent to 0 <code>pile seq</code> , i.e. centred. |
| row seq | Equivalent to 0 <code>row seq</code> . |
| split pic | Allow the <code>DISPLAY</code> -command and the container primitives to split a picture with type <code>row</code> , <code>pile</code> or <code>text</code> automatically in multiple parts, in order to make it look better. <code>split</code> should do something reasonable with other types of picture. |
| combine seq | Return all the pictures contained in seq combined (the order is irrelevant). |
| overlay seq | Special form of <code>combine</code> ; underlaying parts of previous pictures disappear. |

Remarks

- ♦ For `seq` lists, tables or (in the future) sequences of pictures should be allowed; only in a `combine` is the order unimportant.

Additional functions

To make the process of drawing easier, there are also several extra functions proposed, all returning the (absolute) coordinates of a point:

| | |
|-------------------|--|
| vector exit pic | Return the position of a point on the border of pic , where the half line through the centre of pic , and in the direction of vector crosses the border of pic . |
| centre.of pic | Return the position of the centre of pic . |
| north.of pic | Return the position of the 'north' point of pic . |
| dist north.of pic | Return the position at a distance dist north of pic . |
| south.of pic | |
| dist south.of pic | Similar to <code>north.of</code> ; analogous forms exist for: <code>west.of</code> , <code>east.of</code> , <code>nw.of</code> , <code>ne.of</code> , <code>se.of</code> , <code>sw.of</code> . |

Problems and remarks

- ◆ Absolute positioning should be avoided, but this is not possible when the user wants to specify a polygon or polyline. However, the picture can always be moved and scaled, so this is not likely to cause problems.
- ◆ The user will be given as much space as needed; the screen acts as a window on the picture, but the question of how the screen should be divided arises. Reasonable choices for the position of the origin and the coordinate system are:
 - centre of screen $\rightarrow (v, h)$ or (x, y) coordinates
 - bottom left $\rightarrow (x, y)$ coordinates
 - top left $\rightarrow (v, h)$ coordinates

(v, h) means: first coordinate = 'down' (vertical), second = 'across' (horizontal). Preferably, the origin of the screen is put in the middle, because most primitives return a picture with default position $(0, 0)$. Unfortunately, this necessitates the use of negative coordinates which may not be convenient for novices. Putting the origin in one of the corners, on the other hand, has the drawback that the occurrence of negative coordinates is invisible but nevertheless valid (there are no limits on the size of a picture; it can be shifted and scaled down to fit on the screen). Moreover the choice between the bottom left and the top left is difficult: it depends on the coordinate system chosen.

When the screen origin is in the middle, there still is the dilemma between (x, y) and (v, h) coordinates; the former suits mathematicians better, but a naive user would probably expect the latter, since it reflects the way one reads in a lot of languages. Polar coordinates (r, ϕ) are naturally out of the question. For the time being, (x, y) coordinates are chosen but later experience may change this decision. Users will probably quickly get used to either system.

Allowing the user to specify their own preferences as to the position of the origin and the coordinates is another possibility; but then, the system can be in different states, the current one likely to be forgotten. Another possibility is, to provide for functions that convert from one system to the other.

- ◆ Units can be chosen in two ways:
 - absolute ('real world') units, e.g. physical screen pixels, meters, cm, 1/61 feet, character positions (*ens*), points, mm etc.
 - units relative to the dimensions of the device; the maximum is something like 1, 10, 100, 1000, 1024 or whatever.

Relative units are preferable, because pictures can be displayed in their entirety on any device independent of its physical dimensions: what you see (on the screen) is what you get (on paper, on a large demo-monitor, television set or whatever), possibly scaled up or down. The *smaller* dimension of the device should determine the maximum; thus it can be guaranteed that any picture having dimensions smaller than twice the maximum in either direction will fit. Choosing 1 [or 0.5, assuming the origin in the centre] as the maximum is an obvious choice; it is the least arbitrary. It causes no rounding problems because exact (rational) numbers can be used. The nuisance of using fractions can be overcome by expressing coordinates in whatever units you like and scaling pictures down before `DISPLAYing`.

To use absolute units, the system has to know the physical sizes of all devices connected to it, which is not always possible. Of the absolute units, the most acceptable would be **1 mm** which has the advantage that fractions would hardly ever be used and that it is a standard unit in many fields of engineering, etc. Character positions is another possibility, but a problem arises with different fonts.

- ◆ Where will the origin be for each picture? It could be positioned at one of the corners of its borders, for instance the lower left corner, but then mirroring and rotating can cause the origin to change abruptly. Putting the origin of a picture at its centre is preferable; this position has to be calculated for other operations anyway.

4 Structure of pictures

4.1 Linking (connectivity) of sub-pictures

Any non-trivial picture consists of sub-pictures; these sub-pictures have an interrelation based on the specification of the picture. At first sight it seems desirable, in some cases, to maintain these structural relations in the resultant picture. The picture, then, has a kind of 'memory': it contains details about the way it was constructed and which elements of it are really the same entity (*clones* as opposed to *copies*). An example is a picture consisting of a box and a circle connected by an arrow:

```
combine {box1; circle1; box1 arrow circle1}
```

The user can make minor modifications to the picture with the editor without disturbing the internal structure, for instance: moving or scaling the circle causes the arrow to shift too (the structure is always maintained).

On the other hand, modifying the structure itself is quite complicated and may not yield the expected result. It has the inherent danger that a minor modification starts a 'chain reaction' of resulting changes to the picture, or even a circular (endless) chain, even more so when pictures are created by a program. An analogy can be made with a *parse-graph* for functions, in which variables of the same name conceptually contain pointers to one and the same string, so that renaming a variable causes it to change everywhere in the function. Though this might be useful sometimes, in many cases it is a nuisance. Also, it violates the rule in ABC that wherever you put an expression, it is evaluated and its 'original form' is unknown afterwards:

```
>>> PUT box keys IN keyboard
>>> PUT combine {keyboard; keyboard arrow screen; screen}
      IN terminal
```

would yield a structure for `terminal` different from this case:

```
>>> PUT combine {box keys; (box keys) arrow screen; screen}
      IN terminal
```

One of the questions to be answered is: in which cases should changes work through?

- ◆ Lines or arrows between objects: should moving or changing the size of the object affect the line or arrow? And the other way round?
- ◆ Circles, boxes etc around an object: should moving or changing the object affect the enclosing object? And what about changing the enclosing object itself?
- ◆ Composite pictures: are they always to be considered as a whole?
- ◆ Mirroring, scaling etc.: should they work on all clones of a picture?

Strongly linked pictures would give the user the impression that the system is 'not cooperating' and acts in a too 'pseudo-intelligent' way. Incorporating only a few connections leads to the problem that one forgets which functions work through and which don't.

Furthermore, every picture has its own structure (or lack of structure) and in order to generalise it one always will have to make assumptions on heuristic grounds (for example: "this line connects two objects", rather than "this line happens to go from one object to the other"). This also assumes discipline from a programmer to build pictures in a structured way.

4.2 A tree-model

If strong connectivity is not desired, a picture can be represented by a tree rather than a graph. A node of the tree should contain information about the **type**, the **coordinates** of relevant points of the picture and, dependent on the type, zero or more [pointers to] **subtrees**. There is no reason to limit the number of subtrees in this representation: it is a real n -ary tree with dynamic n . Until an ABC-editor for general data structures is available, a picture is most easily modified by editing the function that describes it, rather than editing the data structure itself. This fits the concept of permanently updated session records (see [10]). Assuming that that part of the environment has been implemented, one can modify the *description* of a picture and immediately see the result in another window. This will, in effect, amount to the same as the graph-like structure mentioned previously, but without the chaining problems: changes will work through in the rest of the picture, because the entire picture is completely recalculated.

4.3 Representation of pictures

Pictures can be unambiguously defined in a number of different ways. The ones considered are described below, with as little redundant information as possible:

Additionally, properties such as drawing mode and colour have to be incorporated in the structure (for all types of picture). It is desirable to minimize calculations while modifying pictures. Therefore, coordinates should be as independent from each other as possible.

The alternatives that are starred in Table 1, share some properties:

- ◆ The origin can be located in the centre of the picture; the coordinates are absolute, so the centre is quickly retrievable.
- ◆ All other coordinates are *relative* to the origin; this provides for easy shifting, rotation, scaling and mirroring of pictures.
- ◆ The `DISPLAY`-command has to calculate the absolute positions.

The relevant data have a number of common elements:

type, centre, list of vectors

| type | origin | vectors | size, radius | remarks | chosen | category |
|--------------------------|--------|---------|-----------------|--------------|--------|--------------|
| <i>dot</i> | 1 | | | | * | BASIC |
| <i>line, arrow</i> | 1 | 1 | 1 | <— length | * | |
| <i>text</i> | 1 | | | + the text | * | |
| <i>arc</i> | 1 | 2 | | | * | |
| | 1 | 1 | 1 | <— radius | * | |
| <i>polygon, polyline</i> | 1 | n-1 | | | * | MODIFICATION |
| <i>curve, loop</i> | 1 | n[+m] | | (m= ALL pts) | * | |
| <i>box</i> | 1 | 2 | | + enclosed | * | |
| <i>square</i> | 1 | 1 | | | * | |
| <i>circle</i> | 1 | | 1 | | * | |
| <i>ellipse</i> | 1 | 2 | | | * | COMPOSITION |
| | 2 | | 1 | (2 foci) | * | |
| <i>hull, padded</i> | 1 | n[+m] | | | * | |
| <i>row, pile</i> | 1 | 2 | | (borders) | * | |
| <i>overlay, combine</i> | 1 | 2 | | (borders) | * | |

Table 1. Relevant data of pictures (numbers denote the number of variables of that kind).

The precise interpretation of the vectors depends on the type of picture involved. These data have to be put in a ABC-data structure.

5 Examples of creating pictures

```

HOW TO RETURN ed's.oracle: \ van Thijn dus
  PUT choice {"1992"; "heeft 't"} IN slogan
  IF slogan = "1992":
    RETURN combine {olympic.rings; bottom.text}
  RETURN bottom.text
bottom.text:
  RETURN text ("Amsterdam "^slogan)

```

```

HOW TO RETURN olympic.rings:
  PUT hidden (20 padded (100 circle at (0, 0)) IN ring
  \make a padded ring
  PUT row {ring; ring; ring} IN top.row
  PUT (row {ring; ring}) at south.of top.row IN bottom.row
  RETURN combine {top.row; bottom.row}
  \rings can be coloured if output device supports it

```

```

HOW TO RETURN just.a.picture:
  PUT ellipse text "coffee" IN el
  PUT circle square box (el at (30, 70)) IN combi
  PUT combine {el; el arrow combi; combi} IN res
  RETURN (.01, .01) scaled res

```

6 References

- [1] M.J.A.C. Andreoli, *Taalprimitiva in B voor grafisch editen. Een verkenning*, Internal Report , CWI, Amsterdam (September 1985).
- [2] Steven Pemberton, An alternative simple language and environment for PCs, *IEEE Software* **4**, 1, January 1987, 56—64.
- [3] P.J.W. ten Hagen, T. Hagen, P. Klint, H. Noot, H.J. Sint and A.H. Veen, *Intermediate Language for Pictures*, Mathematical Centre Tracts 130, ISBN 90 6196 2048, Amsterdam (1980).
- [4] F.R.A. Hopgood, D.A. Duce, J.R. Gallop and D.C. Sutcliffe, *Introduction to the graphical kernel system GKS*, Academic Press, Rutherford Appleton Laboratory, Didcot, UK (1983).
- [5] Carol Kaehler, *MacPaint*, Apple, Cupertino, California 95014 (1983).
- [6] Brian W. Kernighan, *PIC — A Language for Typesetting Graphics*, Bell Laboratories, Murray Hill, New Jersey 07974 (March 1982).
- [7] Brian W. Kernighan, *PIC User manual*, Bell Laboratories, Murray Hill, New Jersey 07974 (March 1982).
- [8] L.G.L.T. Meertens, S. Pemberton, and L. Geurts, *The ABC Programmers Handbook*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990. ISBN 0-13-000027-2.
- [9] Joseph F. Ossanna, “Nroff/Troff User’s Manual,” *UNIX programmers Manual 2*, section 22, Bell Laboratories, Murray Hill, New Jersey 07974 (January 1977).
- [10] S. Pemberton, “A glimpse at the B-environment,” *The B Newsletter (issue 1)*, CWI, Amsterdam (August 1983).
- [11] “Cip User’s Manual: One Picture is worth a Thousand Words, *TM-82-11276-1* (1982).
- [12] Christopher J. Van Wyk, *IDEAL User’s Manual*, Bell Laboratories, Murray Hill, New Jersey (1979).
- [13] J. Van Wyk, *A graphics language for typesetting*, Bell laboratories, Murray Hill, New Jersey (1979).