7.3.1.13 The `location` definition

Definitions with the parameter value `usage=location` accept the following directives:

location
type
source
exit **(MUST be the last directive in the series)**

They are used to set up and modify directories of topographical coordinates or to allocate graphical attributes to topographical objects used in the graphical representation of maps.

`location` definitions are *dynamically* linked to the `location` data contained in a database. This means that you can change them even after the database has been set up, and these changes will immediately affect the way the system interprets the topographical identifiers contained in the database.

Regardless of the individual directives the following applies: once the system has finished processing the `location` definition as a whole it holds all the values necessary to exploit the available drawing area as fully as possible. The smallest value for y entered corresponds to the bottom edge of the drawing area, and the largest value for y entered corresponds to the top edge of the drawing area. The same principle applies to the values for x. However, the system retains the proportions of both coordinates: if they do not correspond to those of the actual drawing area, it leaves an area on the map blank in one of the two dimensions. If the resulting map does *not* match up to your expectations, we suggest you define some “invisible” points by selecting `type=` and/or `sign=` parameters as appropriate.

7.3.1.13.1 The `location` directive

This directive has the general form:

FIRSt=<name of object>	**(default: none)**
ORDEr=<number>	**(default: none)**
SECOnd=<name of object>	**(default: none)**
COLOur=Contrast | Red | Blue | Green | Without	**(default: Contrast)**
LINEs=Simple | Double | Treble	**(default: Simple)**
USAGe=Solid | Halftone | Tenthtone | Empty	**(default: Solid)**
SYMBol=Circle | Square | Triangle	**(default: Circle)**
SIZE=<number>	**(default: device dependent)**
WRITe=<character string>	**(default: none)**
SIGN=Yes | No	**(default: Yes)**
CONNected=Yes | No	**(default: Yes)**
FORM=Digipad | Arcinfo	**(default: Digipad)**
ALWAys=No | Yes	**(default: No)**
NAME=<name of group>	**(default: none)**

It is used to declare a single topographical object. This directive differs from other $\kappa\lambda\epsilon\iota\omega$ directives in that it always appears on one (or more) lines in front of a string of coordinates, the precise construction of which depends on the `form=` parameter in force. These strings of coordinates are normally obtained by mechanically digitizing a map.

7.3.1.13.1.1 Declaring a topographical object

Every topographical object is defined by a `location` directive.

Topographical objects can be related to one another, as in the example of a building with an inner courtyard. In this case, two topographical objects are present: the building itself and the inner courtyard, both of which are described by a contour.

These two topographical objects differ in that the inner courtyard is inseparably linked to the building which contains it. For this reason, $\kappa\lambda\epsilon\iota\omega$ combines two topographical objects of this kind into one logical object: because of its object name, the building itself can be addressed from the database, which can also contain it as a value for the core information of any number of `location` entries. The inner courtyard is always displayed if the building itself is being displayed, but cannot be processed independently.

Topographical objects can, furthermore, consist of single points: a symbol, centered around that point, will be displayed whensoever the object becomes eligible for display.

7.3.1.13.1.1.1 The `first=` parameter

This parameter expects the object name of a topographical object as a parameter value. This is a standard $\kappa\lambda\epsilon\iota\omega$ name: it may therefore consist of up to twelve characters (including letters, numbers, full stops (periods) ("."), hyphens ("-"), and underscores ("_")). The first character must be an alphabetic character (a letter).

The parameter tells the system that the `location` directive is being used to introduce a definition for a new topographical object. The adjoining string of coordinates describes the outermost contour of a closed topographical object (cf. section 7.3.1.13.1.2.5 below for more information on this), the contour of an open topographical object or a single point.

If this parameter is missing the system assumes that the `location` directive in question is describing a topographical object which is inseparably linked to the last topographical object defined, as described in the example above. The adjoining string of coordinates describes a line which is completely enclosed by the previously encountered object. This is only possible if the latter was a closed topographical object.

Thus the following example describes a building `no.12` with one inner courtyard, and a building `no.11` which is simply defined by its external contour.

```
location first=no.11
...  string of coordinates for the outer contour
location first=no.12
...  string of coordinates for the outer contour
location
...  contained string of coordinates
```

7.3.1.13.1.1.2 The `order=` parameter

This parameter expects a value consisting of a whole number greater than or equal to one. The meaning of this parameter can best be explained with the help of an example. Let us assume that we want to define a house with two inner courtyards, the second of which contains two small well-houses. The directives:

```
location first=house24
...  string of coordinates for the outer contour
location
...  string of coordinates for the first inner courtyard
location
...  string of coordinates for the second inner courtyard
location
...  string of coordinates for the first well-house
location
...  string of coordinates for the second well-house
```

do *not* have the desired effect because according to the standard definition, the system interprets every `location` directive without a `first=` parameter as an inner contour of the previous `location` directive. Hence the system would interpret the directives in our example as follows:

```
location first=house24
...  string of coordinates for the outer contour
location
...  string of coordinates for the first inner courtyard
location
...  string of coordinates for 'x' inside the first inner courtyard
location
...  string of coordinates for 'y' inside 'x'
location
...  string of coordinates for 'z' inside 'y'
```

In order to avoid this, you can explicitly describe the subordination in `order=`. Here the following applies: every `location` directive with a `first=` parameter is classified as "1". If you specify `order=` the corresponding `location` directive is interpreted as the inner contour of the last `location` directive, and its `order=` number is decremented. This logic precisely corresponds to the logic whereby the order in which a series of groups appears determines their logical relationships. So we could represent the faulty set of assumptions described above in the following way; as a series of subordination relationships which $\kappa\lambda\epsilon\iota\omega$ assumes to exist:

```
location first=house24;order=1
...  string of coordinates for outer contour
location ...;order=2
...  string of coordinates for first inner courtyard
location ...;order=3
...  string of coordinates for 'x' inside the first inner courtyard
location ...;order=4
...  string of coordinates for 'y' inside 'x'
location ...;order=5
...  string of coordinates for 'z' inside 'y'
```

To put the objects into their correct order, you would describe them as follows:

```
location first=house24;order=1
...  string of coordinates for the outer contour
location ...;order=2
...  string of coordinates for the first inner courtyard
location ...;order=2
...  string of coordinates for the second inner courtyard
location ...;order=3
...  string of coordinates for the first well-house
location ...;order=3
...  string of coordinates for the second well-house
```

or, if we omit all the parameters which are equivalent to default settings:

```
location first=house24
...  string of coordinates for the outer contour
location
...  string of coordinates for the first inner courtyard
location ...;order=2
...  string of coordinates for the second inner courtyard
location
...  string of coordinates for the first well-house
location ...;order=3
...  string of coordinates for the second well-house
```

7.3.1.13.1.1.3 The `second=` parameter

Like the `first=`, this parameter expects the name of a topographical object as value. It tells the system that the adjoining string of coordinates is contained in the string previously defined by `first=`. The parameter value must exactly match that of the previous `first=` parameter. This parameter is rearely used, as the same results can be obtained by omitting `first=`. However, we recommend that you include it if there is any risk of accidentally rearranging directives while preparing a file containing strings of coordinates.

7.3.1.13.1.2 Displaying a topographical object

The following parameters define the attributes acquired by a topographical object when it is displayed by means of a `mapping` command (cf. section 8.3.7) unless you explicitly selected other attributes in this command.

7.3.1.13.1.2.1 The `colour=` parameter

This parameter accepts the keywords `contrast`, `red`, `blue` and `green` as values, all of which can be shortened to their respective first character.

By default the contours of all topographical objects are drawn with "maximum contrast": i.e. black when printed on paper, white on a screen with a dark background; where the string of coordinates is closed, the enclosed area remains the same colour as the background.

You can use the `colour=` parameter to change this colour palette. You can specify this parameter twice in every `location` directive.

The first `colour=` parameter defines the colour of the topographical object's contours. The second parameter, which you may only specify if the topographical object is closed, defines the colour of the enclosed area. In the case of objects containing inner contours, the area between the outer contour and the contained enclosed inner contour is filled with colour: the colour of the area enclosed within the inner contour is in turn defined by the `colour=` parameter of the `location` directive which defined that particular contour.

If you do not specify this parameter but it appeared in the last `type=` directive you defined, the system replaces its default settings with the value(s) in that directive.

The keywords listed above are normally defined as follows. They should be understood as the lowest common denominators of the graphical hardware in your computer; this is an area in which we can expect radical improvements in the near future.

- Contours:
  - `contrast`: black on white background, white on dark background.
  - `red`: dark red.
  - `blue`: dark blue.
  - `green`: dark green.
- Colours for enclosed areas (if available on your graphics card; otherwise identical to contour colours)
  - `without` or parameter omitted: background colour.
  - `contrast`: black on white background, white on dark background.
  - `red`: light red.
  - `blue`: light blue.
  - `green`: light green.

7.3.1.13.1.2.2 The `lines=` parameter

This parameter accepts the keywords `simple`, `double` or `treble` as values, all of which can be shortened to their respective first character.

It defines the line width of the contours.

- `simple` instructs the system to draw each line with just one pass of the plotter pen, or display the line on-screen as one pixel in width. (This is the system's default setting.)
- `double` instructs the system to draw every contour as two parallel lines of this type.
- `treble` instructs the system to draw every contour as three parallel lines of this type.

7.3.1.13.1.2.3 The `usage=` parameter

This parameter accepts the keywords `solid`, `halftone`, `tenthtone` or `empty` as parameter values. All of them can be shortened to their respective first character.

By default the contours of topographical objects are drawn as simple continuous lines; where strings of coordinates are closed, the entire enclosed area is filled with the colour defined in the relevant `colour=` parameter, if you defined one.

This default can be modified this model by specifying the `usage=` parameter. You can specify this parameter twice for every `location` directive.

The first `usage=` parameter you specify defines the pattern you want to use for the topographical object's contours. The second parameter, which you can only specify if the topographical object is closed, defines the area fill pattern. The way the inside area in objects with enclosed lines is defined correponsds to the definition used in the `colour=` parameter.

If you do not specify this parameter, but it did appear in the last `type=` directive the system will replace its normal default settings with the value of the parameter in that last directive.

The keywords listed above are defined as follows for the time being. You should regard them as the "lowest common" denominators for your graphical hardware; this is an area where we can expect some dramatic improvements in the near future.

- Contours:
  - `solid`: continuous contour.
  - `halftone`: "dashed", i.e. c.50 % of the points on the contour are described.
  - `tenthtone`: "dotted", i.e. c.10 % of the points on the contour are described.
  - `empty`: no line is drawn at all.
- Area fill pattern:
  - `solid`: entire area filled.
  - `halftone`: "heavy shading", i.e. c.50 % of the points in the area are coloured.
  - `tenthtone`: "light shading", i.e. c.10 % of the points in the area are coloured.
  - `empty`: the area is left blank.

7.3.1.13.1.2.4 The `sign=` parameter

This parameter expects one of the keywords `no` or `yes` as a value. Both of these can be shortened to their respective first character.

- `yes` means that the topographical object is labelled on every map derived from this `location` definition. This is the system's default setting.
- `no` suppresses this labelling feature.

If you do not specify this parameter, but it appeared in your last `type=` directive, the system will use the value there instead of its default settings.

7.3.1.13.1.2.5 The `write=` parameter

This parameter expects a constant as a parameter value.

By default, a topographical object on a map is accompanied by a label which the system generates from the topographical object's object name: thus the label HOUSE23 appears next to the topographical object `house23`. This may result in problems if you want to identify topographical objects by written texts which are not suitable for use as $\kappa\lambda\epsilon\iota\omega$ names: e.g. identifiers including blank spaces, such as RIVER DANUBE or other identifiers which are more than twelve characters in length, such as ST. PETER AND ST. PAUL. In such cases, you can use the `write=` parameter to declare a label which deviates from the object's standard identifier:

```
location write="River Danube"
...  string of coordinates
location write="St. Peter and St. Paul"
...  string of coordinates
```

7.3.1.13.1.2.6 The `symbol=` parameter

This parameter accepts the keywords `circle`, `square` or `triangle` as parameter values. All of them can be shortened to their respective first character.

When this parameter is specified for a topographical object which consists of a single point, a symbol of the shape indicated by the keyword will be displayed centered around that point. If this parameter is specified, when more than one point is defined, the symbol will be displayed centered around the first point of the polygon describing the topographical object.

If a topographical object consists of only one point and this parameter is not specified, a circle will be displayed in its position.

7.3.1.13.1.2.7 The `size=` parameter

This parameter expects as value a number. It describes the default size of the symbol displayed for the topographical object. The number is interpreted according to the granularity of the particular graphical device on which the map is displayed. At the moment we recommend experimentation to find out about the most appropriate one.

7.3.1.13.1.3 A topographical object's topological properties

7.3.1.13.1.3.1 The `connected=` parameter

This parameter expects one of the keywords `no` or `yes` as a value. Both of these can be shortened to their respective first character.

- `yes` instructs the system to treat the string of coordinates as a closed contour, in which the last coordinate point is linked to the first one. This is the system's default setting.
- `no` instructs the system to interpret the string of coordinates as an open contour.

Open strings of coordinates are always interpreted as lines, never as areas: this means you cannot assign area fill colours to them.

7.3.1.13.1.4 Inputting a topographical object

7.3.1.13.1.4.1 The `form=` parameter

This parameter accepts the keywords `digipad` and `arcinfo`[1] as values. Both of these can be shortened to their first character.

This parameter defines the format in which a string of coordinates can be entered following a `location` directive. The parameter values for this parameter correspond to the manufacturers' names of various digitizing devices and/or Graphical Information Systems which provide the relevant formats.

If this parameter is missing the system uses the value of the `form=` parameter in the last `type=` directive.

7.3.1.13.1.4.1.1 The `digipad` format

With this format all lines following a `location` directive and starting with a number in the first column are regarded as part of a definition of a contiguous string of coordinates. Each line is constructed as follows:

- In the first column, the line begins with a number, which tells the system how many points in the string of coordinates are defined in this line.
- This is followed by the coordinates for each individual point, separated from the initial number and each other by single spaces.
- Each of these coordinates begins with an x coordinate expressed as an integer, immediately followed by a comma and the y coordinate, also expressed as an integer.
- The coordinates measure the distance from a coordinate origin in the bottom left corner of the drawing area. Negative values are illegal.

Hence the `location` directive:

```
location first=h123;form=digipad
6 2683, 3976 2717, 4067 2812, 4050 2816, 4062 2887, 4051 2887, 4041
2 2904, 4036 2884, 3946
```

defines the topographical object `h123`, which consists of eight coordinate pairs.

If this parameter is missing, the system uses the value of the `form=` parameter in the last `type=` directive.

[1] In the case of ARC/INFO $\kappa\lambda\epsilon\iota\omega$ assumes a specific setting of this system to be used. More specifically, coordinates must be generated with the use of ARC/INFO's UNGENERATE command, ensuring that arcs have only one node.

7.3.1.13.1.4.1.2 The `arcinfo` format

With this format each line following a `location` directive contains the coordinates of exactly one point of the topographical object,

- Starting in the first column with the x coordinate, followed by one or more space characters, and
- the y coordinate.
- Both coordinates are expressed as real numbers, measuring inches or centimetre on the tablet of the digitizer.
- A topographical objects ends with a line containing the keyword `end` in the first three columns.

In this format the example given above would therefore read:

```
location first=h123;form=arcinfo
26.83 39.76
27.17 40.67
28.12 40.50
28.16 40.62
28.87 40.51
28.87 40.41
29.04 40.36
28.84 39.46
end
```

7.3.1.13.1.5 The ownership of a topographical object

When the system creates a map on the basis of information contained in a database, it only displays those topographical objects which have you have selected explicitly in a `mapping` command. Quite independently of any topographical objects you have deliberately selected in this way, you can also define how the general background of a map should be formed.

7.3.1.13.1.5.1 The `always=` parameter

This parameter expects one of the keywords `no` or `yes` as a value. Both of these can be shortened to their respective first character.

- `no` explicitly instructs the system to behave as it would by default, i.e. only to draw this object if the latter is explicitly included in a map based on this `location` definition.
- `yes`, on the other hand, instructs the system to include this topographical object in any and every map based on the `location` definition in question.

7.3.1.13.1.5.2 The `name=` parameter

3.3.5) The `name=` parameter.

This parameter expects a standard $\kappa\lambda\epsilon\iota\omega$ name as a parameter value.

It combines all topographical objects for which the same name has been specified to form a single group. You can address the group by this name whenever you are producing maps.

7.3.1.13.2 The `type` directive

This directive has the general form:

TYPE=Insertion | Creation | Substitution | Destruction	**(default: Insertion)**
COLOUr=Contrast | Red | Blue | Green | Without	**(default: Contrast)**
LINEs=Simple | Double | Treble	**(default: Simple)**
USAGe=Solid | Halftone | Tenthtone | Empty	**(default: Solid)**
SIGN=Yes | No	**(default: Yes)**
FORM=Digipad | Arcinfo	**(default: Digipad)**

It determines whether the following `location` directives should be interpreted as instructions for redefining, replacing or deleting topographical objects, and defines the default attributes for any subsequent `location` directives.

7.3.1.13.2.1 The `type=` parameter

The `type=` parameter accepts the keywords `creation`, `insertion`, `substitution` or `destruction` as values All of these can be shortened to their respective first character. They instruct the system how to interpret any following `location` directives.

- `creation` and `insertion` are synonymous. They tell the system that the following `location` directives define new topographical objects. If a `location` directive in this mode addresses the name of an object which has already appeared in the definition, the system refuses to execute this directive (or any of the following directives up to the next `location` directive with a `first=` parameter) and displays an error message.
- `substitution` causes the system to behave as it does in `insertion` mode, except that if you address a topographical object with an object name which has already appeared in the `location` definition, the system deletes this object and then includes the newly defined object into the `location` definition under the same name as the old object. At present, you cannot change a topographical object's individual properties: you must always completely redefine the object in question.
- `destruction` instructs the system to delete the topographical object mentioned in any following `location` directives' `first=` parameter. In this mode, it is mandatory to specify a `first=` parameter for all following `location` directives; you should not specify any other parameters. The object defined in the `first=` parameter is deleted in this mode.

7.3.1.13.2.2 The `colour=` parameter

This parameter accepts the keywords `contrast`, `red`, `blue` and `green` as values, all of which can be shortened to their respective first character.

If a `location` directive does not contain the parameter of the same name the system uses the parameter value which was assigned to this parameter in the preceding `type` directive.

The system interprets the parameter values as described in section 7.3.1.13.1.2.1 above. Similarly, you can define this parameter twice in every `type` directive.

7.3.1.13.2.3 The `lines=` parameter

This parameter accepts the keywords `simple`, `double` and `treble` as values, all of which can be shortened to their first character.

If a `location` directive does not contain the parameter of the same name, the system uses the parameter value which was assigned to this parameter in the last `type` directive.

The system interprets the parameter values as described in section 7.3.1.13.1.2.2 above.

7.3.1.13.2.4 The `usage=` parameter

This parameter accepts one of the keywords `solid`, `halftone`, `tenthtone` or `empty`. All of these can be shortened to their respective first character.

If a `location` directive does not contain the parameter of the same name the system uses the parameter value which was assigned to this parameter in the last `type` directive.

The system interprets the parameter values in the manner described above in section 7.3.1.13.1.2.3. Similarly, you can define this parameter twice in every `type` directive.

7.3.1.13.2.5 The `sign=` parameter

This parameter expects a parameter value consisting of one of the keywords `no` or `yes`. Both of these can be shortened to their respective first character.

If a `location` directive does not contain the parameter of the same name the system uses the parameter value which was assigned to this parameter in the last `type` directive.

The system interprets the parameter values in the manner described above in section 7.3.1.13.1.2.4.

7.3.1.13.2.6 The `form=` parameter

This parameter accepts the keywords `digipad` and `arcinfo` as values. Both of these can be shortened them to their first character.

If a `location` directive does not contain the parameter of the same name, the system uses the parameter value which was assigned to this parameter in the last `type` directive.

The system interprets this parameter value in the manner described above in section 7.3.1.13.1.4.1.

7.3.1.13.3 The `source` directive

This directive has the general form:

FIRSt=<number>	**(default: none)**
SECOnd=<number>	**(default: none)**
STARt=No | Yes	**(default: No)**
DISTance=<number>	**(default: none)**
ANGLe=<number>	**(default: none)**
SCALe=<number>	**(default: none)**
FORM=Digipad | Arcinfo	**(default: Digipad)**

This directive is used to link up parts of a map which have been digitized separately.

It provides a number of methods to do so. Which method is choosen, depends on the subset of parameters specified.

7.3.1.13.3.1 Simple offset method

This method is selected, if the `first=` and the `second=` parameter are defined and `start=` is either undefined or explicitly set to `no` (which is the default). In that case all other parameters are considered erroneous.

With this method a simple numerical offset is added to the coordinates of the points defined. Both parameters mentioned accept any numbers (i.e. including negative numbers) as parameter values:

- `first=` defines the value used to adjust x-axis coordinates, and
- `second=` defines the value by which the y-axis coordinates should be adjusted.

7.3.1.13.3.2 Shifting static origin

This method is selected, if the `first=` and the `second=` parameter are defined and `start=` is set to `yes`. In that case all other parameters are legal (see below).

When this configuration of parameters is encountered for the first time, the values of `first=` and `second=` are stored. Both, `first=` and `second=`, have been specified as numbers which belong to the same system of coordinates used for the definition of subsequent topographical objects.

- `first=` defines the x coordinate of an arbitrary point.
- `second=` defines the y coordinate of an arbitrary point.

If later another `source` directive is encountered with the same parameter set being specified it is assumed, that the `first=` and `second=` parameter on the two directives specify excatly the same arbitrary point. All coordinates encountered after the second `source` directive are modified as is necessary to relate them to the same coordinate system as used for the coordinates following the first `source` directive. (And so *mutatis mutandis* for all further `source` directives encountered.)

7.3.1.13.3.3 Shifting variable origin

This method is selected, if a static origin has been defined on an initial `source` directive and on a subsequent `source` directive `first=` and `second=` are encountered again, `start=` is set to `yes` and `distance=` and `angle=` are defined as well.

With this configuration of parameters it is assumed, that `distance=` describes the direct geometrical distance between the coordinates specified on the `source` directive specifying the initial static origin and the origin specified on the current `source` directive. (*For the time being no allowance is made for the three-dimensionality of the Earth.*) If no further parameter is specified, it is assumed that the distance is described in the scale used for specifying all the coordinates. `angle=` describes the angle in degrees between the x-line going through the first static origin and the direct line connecting this static origin with the currently defined one.

7.3.1.13.3.4 Shifting scales

Both the parameter sets used for a static as well as for a dynamic origin can be combined with a `scale=` parameter. If that parameter is specified, the system expects that immediately after the `source` directive there follows the definition of a topographical object defined by two points, as described in section 7.3.1.13.1.4. (If necessary the format for the digitised points can be selected with the `form=` parameter.) This information is used as follows:

It is assumed that the `scale=` parameter of the first defined origin (i.e., on the first `source` directive) and the one on the current `source` directive give the length of a distance on the underlying map, which is given in common units. (Most usually kilometres or miles.) It is further assumed, that the two coordinate pairs following the `source` directive give the start and end point of a line of exactly that length on the underlying maps.

When the coordinates following the last `source` directive will be modified, the difference in scale expressed by these two magnitudes will be taken into account.

7.3.1.14 Defining sort sequences

Every computer has a given sorting sequence which is determined by the operating system. This is usually rather limiting when processing linguistic material. For instance, it will often cause alphabetical characters to be sorted in the order a, b, ... z, A, B, ... Z. To avoid this kind of problem, κλειω applies the following logic:

- Sort sequences are managed as logical objects with the class name `order`.
- When you invoke κλειω, the system generates a logical object of the `order` type.
- The commands `index`, `catalogue` and `create` sort data according to the sorting sequence defined by this object. The `index` command (or the individual logical "columns" in an index) can access sorting sequences with other names.
- You can make any changes you like to the logical object called `order`.
- When you set up a database, the logical object `order` is stored in the database in exactly the same way as objects called `text`, `number`, `date`, `category`, `relation`, `location` or `image`.
- Whenever you access this database, κλειω automatically reactivates the sorting sequence associated with the database.

7.3.1.14.1 The default sort sequence

Because of differences between the character sets available on different computers, κλειω does not use the same sorting sequence on every machine. To find out which sort sequence κλειω uses on your computer, enter the κλειω command

```
describe name=order;usage=order
```

Regardless of your computer, however, κλειω always constructs its *default sort sequence* according to the same rules. These are as follows:

1) The first character in the sort sequence is a blank space.
2) This is followed by the letters of the Roman alphabet. In this instance
   2.1) upper-case and lower-case letters are classified as equivalent,
   2.2) letters with diacritical marks (ä, á, à, â ...) are classified as equivalent to the basic letters.
3) If the computer you are using has access to other national alphabets *as a standard feature*, the relevant alphabetic characters now follow, in the appropriate alphabetical order.
4) Finally, numbers are included, as well as
5) any non-graphical special characters, in the sorting order specified by your particular operating system and finally
6) any graphical characters, in the sorting order specified by your operating system.

The order in which characters are sorted by default depends on the *configuration* of κλειω on your machine. Please consult appendix B in this manual on the *κλειω-configuration program* on how to use that configuration program to adapt the defaults, for example to a non-standard keyboard.

7.3.1.14.2 The `order` definition

The following directives are available to definitions with the parameter value `usage=order`:

sign
conversion
training
exit **(MUST be the last directive in the series)**

This class of definition is used to declare sort sequences. It differs from other definitions in that *every* newly declared sort sequence is first initialised against the default sort sequence on the computer you are using. Hence *every* definition of this class modifies an already existing sort sequence: because most of the sorting sequences defined by users mean that any changes to the default sort sequence are likely to be minimal, most definitions of this type can be kept very short.

7.3.1.14.2.1 The `sign` directive

This directive has the general form:

SIGN=<character>	**(MUST be specified)**
USAGe=<Character>	**(default: none)**
BEFOre=<character>	**(default: none)**
AFTEr=<character>	**(default: none)**
ORDEr=<number>	**(default: none)**

Note: you must specify one and only one of the parameters usage=, after=, before= or order=.

This directive assigns a particular position in the sort sequence to each character.

7.3.1.14.2.1.1 The `sign=` parameter

The `sign=` parameter expects a single character (any character) as a value. The remaining parameters in the directive define the position of this character in the sorting order.

7.3.1.14.2.1.2 The `usage=` parameter

This parameter expects a parameter value consisting of a single character (any character). It instructs the system to treat this character and the character included in `sign=` as identical for sorting purposes.

Thus in order to make `i` and `j`, or `u` and `v` identical for sorting purposes, you would use the four directives below:

```
sign sign=j;usage=i
sign sign=J;usage=i
sign sign=v;usage=u
sign sign=V;usage=u
```

7.3.1.14.2.1.3 The `after=` parameter

This parameter expects a single character (any character) as its value. It instructs the system to include the character specified in `sign=` *after* the named character in the sorting order. So in order to sort ä *after* `a`, you would use the following directive:

```
sign sign=ä;after=a
```

7.3.1.14.2.1.4 The `before=` parameter

This parameter expects a parameter value consisting of a single character (any character). It instructs the system to include the character specified in `sign=` *before* the named character in the sorting order. Thus in order to sort `a` *before* `ä`, you would use the following directive:

```
sign sign=a;before=ä
```

7.3.1.14.2.1.5 The `order=` parameter

This parameter expects a parameter value consisting of a number between 1 and 255. It tells the system where in the sorting order to position the character assigned to `sign=`. On the one hand, you can use this option to establish totally different sorting orders from the alphabetical one – e.g. for texts originally written in non-Roman alphabets and subsequently transcribed in the Roman alphabet. The other important application of this parameter is in defining the order in which non-alphabetical special characters should be sorted. We advise you to assign high numerical values to such characters in order to avoid conflicts with letters and numbers. Thus in order to sort the characters `&`, `%` and `$` in that order, you would use the following directives (for example):

```
sign sign=&;order=101
sign sign=%;order=102
sign sign=$;order=103
```

7.3.1.14.2.1.6 Interaction between the parameters

As a general rule, within the definition of one sort order, you should not combine the two options for defining a relative sorting order (using `usage=`, `after=` and `before=`) and the option for defining an absolute sorting order (using `order=`). If you use `describe` to display the currently active sort sequence, it causes the numerical value for every group of characters in the sorting order to be displayed, but this numerical value is constantly changing while the sorting order is being processed by `after=` and `before=` parameters.

7.3.1.14.2.2 The `conversion` directive

This directive has the general form:

PREParation=<name of definition>	**(default: none)**
WITHout=<character string>	**(default: none)**
FORM=<character>	**(default: none)**
SIGN=<character>	**(default: none)**

Before any character string is compared to other character strings for sorting purposes, you can use this directive to convert it into a sort key.

7.3.1.14.2.2.1 The `before=` parameter

This parameter expects the name of a `conversion` definition as a parameter value. It is used to convert character strings which are to be sorted into sort keys before the system compares them in order to determine their sorting order.

7.3.1.14.2.2.2 The `without=` parameter

This parameter expects a constant as a parameter value. It is used to define a list of characters which should be ignored during sorting *after* any `conversion` operation.

So if we assume the existence of a `conversion` definition with the name `texsort`, designed to remove various TEX-names from a text which is to be sorted, we might use the following directive to ensure it does so before the sort is performed as well as to ensure that the system ignores any remaining braces ({ and }):

```
conversion preparation=texsort;without="{}"
```

7.3.1.14.2.2.3 The `form=` parameter

This parameter expects a single character as a parameter value. You apply it to the text to be sorted in order to instruct the system to differentiate between a value to be used for sorting and a value to be used for display purposes.

If we assume that `form=` has been assigned a value of `+`, this results in the following behaviour. Let us assume we want to sort the following text:

```
+1792+seventeenhundredandninetytwo+
```

In this case, $\kappa\lambda\epsilon\iota\omega$ will use the value `seventeenhundredandninetytwo` for sorting purposes: but the system component which initiated the sort operation receives the character string `1792` for further processing. Normally used for generating indices, this feature means that character strings can be placed in other positions in the sequence than the positions in which they would appear if the sort operation was purely alphabetical.

7.3.1.14.2.2.4 Sorting older character conventions

The parameters described in the next two paragraphs were introduced in order to allow the user to surmount particular sorting problems arising with respect to older stocks of data. In this case the data was incorporated into the database using what is known as the *ZMD convention*. This convention specifies the following format for entering special characters and diacritical marks which cannot be displayed on a given input device (e.g. a terminal):

$$< escape\ code >< basic\ letter >< modifier >$$

where the $< modifier >$ is normally numeric.

The following parameters are intended to allow you to ignore such codes and remove them from input data intended for sorting as economically and efficiently as possible.

You should only use these parameters if you are obliged to deal with a database containing such codes and organisational reasons mean that you do not wish to convert them into more modern conventions.

The fact that these parameters are available does not imply that we recommend the convention we have just described.

Do *not* use this convention when you are entering new data.

7.3.1.14.2.2.4.1 The `sign=` parameter

The `sign=` parameter expects a value consisting of a special character which has not been reserved as a data signal character.

It is used to define the escape code in a ZMD convention. This escape code, together with the first character after the basic letter, are ignored in sort operations.

7.3.1.14.2.2.4.2 The `usage=` parameter

The `usage=` parameter expects a positive number as a parameter value.

It tells the system how many characters to expect in a ZMD convention's record code, i.e. how many of the characters immediately after the basic letter should be ignored in sort operations.

7.3.1.14.2.3 The `training` directive

This directive is used to test a sorting order. To date, no parameters have been defined for this directive. you can only use it in interactive mode.

After this directive κλειω expects individual lines to contain excatly one character string each. Pairs of successive lines are compared. The system shows the order in which they will appear according to the sorting order you have just defined. A line starting with an asterisk (`‘‘*’’`) ends this training mode.

7.3.1.15 The `catalogue` definition

Catalogues are always set up using the `catalogue` command; hence definitions with the parameter value `usage=catalogue` are used to process existing catalogues. The following directives are available for such operations:

keyword
exit **(MUST be the last directive in the series)**

They allow you to insert lists of references into a library of such lists which has been assigned to a catalogue. In the current version of $\kappa\lambda\epsilon\iota\omega$ a reference is always defined in terms of the catalogue's standard search context, that is, of an entry in the database.

7.3.1.15.1 The `keyword` directive

This directive has the general form:

NAME=<name>	**(MUST be specified)**
FORM=<pattern definition>	**(MUST be specified)**
BRIDge=No | Yes	**(default: No)**
OVERwrite=No | Yes	**(default: No)**

It determines how a given list of references is defined. It may be repeated as many times as you like within a definition.

7.3.1.15.1.1 The `name=` parameter

This parameter expects the name of a list of references as a value. Thereafter, the list of references defined in the `form=` parameter can be accessed under this name.

7.3.1.15.1.2 The `form=` parameter

This parameter expects a value consisting of a pattern of linguistic terms, which by either appearing or not appearing together define the list of references. Patterns of this kind consist of:

- *Terms*, which must appear together in a contextual unit. If a pattern consists of just one term, the latter need not be delimited: otherwise terms should be enclosed either in quotation marks or inverted commas.
- *Built-in functions*, which control access to information items in a catalogue which have been defined differently.
- *Linkage operators*, which control the joint appearance or non-appearance of the terms.
- Logical brackets, which control the order in which the linkage operators are analysed.

Two terms and/or built-in functions must be separated by a linkage operator; two linkage operators must be separated by a term or a built-in function. If the system cannot find a term (or a built-in function's argument) in the relevant catalogue, the pattern as a whole is held to be invalid and no reference is entered in the reference library.

The meaning of the individual components is illustrated in the examples below, which are explained in greater detail in the following sections.

example no.	example
1	`"school"`
2	`school&`
3	`"school&" and 'church&'`
4	`"school&" and church&`
5	`"&abimus" or "&ibimus"`
6	`name[test1] not "caretaker"`
7	`lemma[Winiger] and lemma[Adalwin]`
8	`"pregnant" or "impregnate" not "months" or "weeks"`
9	`"pregnant" or "impregnate" not ("months" or "weeks")`

7.3.1.15.1.2.1 Terms

As a rule, terms in a pattern consist of constants, i.e. character strings, which appear either between two quotation marks or two inverted commas. If a pattern does not contain quotation marks or inverted commas or a built-in function at the point at which the system expects to find a term the system interprets the remainder of the pattern as *one* term. Thus in the preceding examples, examples 1 and 2 consist of just one term each, examples 3, 4 and 5 of two terms each, connected by a linkage operator. Example 6 finishes with a term, after a linkage operator. Examples 8 and 9 contain four terms each.

In normal circumstances every term is understood as a word form which must be included in the relevant catalogue in exactly the same spelling.

If the term ends with the special character "`&`", it is interpreted as the beginning of a word. In the preceding examples 2, 3 and 4, the system searches for words beginning with `school`, and in examples 3 and 4 (quite independently) for words beginning with `church`. If you try to use a pattern to search for the beginnings of words in a catalogue which does not contain the access path "beginnings of words" (cf. section 8.3.2.8), the system treats the entire pattern as invalid.

If the term starts with the special character "`&`", it is interpreted as an end of word. Thus in example 5, the system is searching for words ending with `abimus` and, quite independently, `ibimus`. If you try to use a pattern to search for word endings in a catalogue which does not contain the access path "ends of words" (cf. section 8.3.2.8), the system treats the entire pattern as invalid.

7.3.1.15.1.2.2 Built-in functions

Built-in functions start with one of the two keywords `name` or `lemma` and contain a character string between two square brackets as an argument.

7.3.1.15.1.2.2.1 The `name[]` built-in function

This built-in function interprets its argument as the name of a list of references, stored previously with the help of the interactive system or another `keyword` directive. Thus in example 6 above, the system searches for a previously stored list of references with the name `test1`.

If a catalogue's library does not contain a list of references with the specified name the system treats the entire pattern as invalid.

7.3.1.15.1.2.2.2 The `lemma[]` built-in function

This built-in function converts its argument into a derived form, using the procedure defined for this catalogue, and searches for this form in the catalogue which is being processed. If the currently processed catalogue does not contain an access path covering derived forms (cf. section 8.3.2.8.3) the system treats the entire pattern as invalid.

Thus in example 7 above, the system searches for names which can be traced back to the same basic form using the procedure defined for this catalogue, such as the names `Winiger` and `Adalwin`, for example.

7.3.1.15.1.2.3 Linkage operators

Patterns are used to represent an abbreviated form of dialogue, with the aim of setting up a list of references. Thus each pattern links the results of steps taken to date, i.e. all parts of the pattern appearing on their left, to the part of the total pattern which appears immediately to their right, in a particular way.

The operator `and` stipulates that the part of the pattern immediately to its right must also be contained in the references defined by the preceding pattern if these references are to remain in the list. Thus in examples 3 and 4 above, the system searches for contexts containing word forms beginning with both `school` and `church`; in example 7 above, it searches for contexts containing a word form which can be traced back to the same form as `Winiger` as well as a form which can be traced back to as `Adalwin`.

The operator `or` causes all instances of the part of the pattern appearing immediately to its right to be appended to the list of references built up to date. Thus in example 5 above, the system searches for contexts containing at least one word form ending with `abimus` or `ibimus`.

The operator `not` instructs the system to retain only those references in the list built up to date which do not contain the part of the pattern appearing immediately to the right of the operator. Thus in example 6 above, the system searches for those references in the list named `test1` which do not contain the word form `caretaker`.

7.3.1.15.1.2.4 Logical brackets

As we have already established in the preceding section, patterns which simulate a dialogue are always processed from left to right. Hence example 8 above, would be interpreted as follows: *search for contexts containing the word form* `pregnant`, *to this list add any contexts containing the word form* `impregnate`, *to the resulting list remove any contexts containing the word form* `months`; *finally, append to the resulting list any contexts containing the word form* `weeks`.

You can modify this mechanical left-to-right order of processing by inserting pairs of brackets nested to any depth; the only condition which applies to their insertion is that a closing (right) bracket must be separated from the next opening (left) bracket by a linkage operator. This means we can interpret example 9 above, as follows: *search for contexts containing the word form* `pregnant`, *add any contexts containing the word form* `impregnate` *to the resulting list; retain this list. Now create a new list of all those contexts containing the word form* `months`, *add any contexts containing the word form* `weeks` *to the resulting list. Finally remove any contexts appearing in the second list from the first list you retained.*

7.3.1.15.1.3 The `bridge=` parameter

This parameter expects a parameter value consisting of one of the two keywords `yes` or `no`. Both of them can be shortened to their respective first character.

By default the system stores a list of references as if it was created using the menu option "`Save`" on the Full text menu (cf. "Halbgraue Reihe", volume B10, section 2.2.3). This means that when you are working with commands you can use the *group function* `keyword[]` to access the entries contained in the list of references (cf. section 8.1.1.2.2.4) but you *cannot* use the *element function* `keyword[]` (cf. section 8.1.1.4.2.46) to demonstrate that a particular entry appears in this list of references.

- `no` confirms that the system should behave in this way by default.
- `yes` instructs the system to manage the list of references as if it were a list of key words, i.e. a list of references defined by selecting `Link` from the Full text menu. The *element function* `keyword[]` can also reproduce lists of references of this kind.

7.3.1.15.1.4 The `overwrite=` parameter

This parameter expects a parameter value consisting of one of the two keywords `yes` or `no`. Both of these can be shortened to their respective first character.

The function of the `overwrite=` parameter in the `item` command introducing the definition is to specify whether the system should overwrite a list of references which has already been defined in the `name=` parameter with the newly defined list. You can change this behaviour selectively for each separate reference using the parameter of the same name for each separate `keyword` directive.

- `no` prevents the system from overwriting an existing list of references.
- `yes` allows it to do so.

7.3.1.16 The `connection` definition

Definitions with the parameter value `usage=connection` accept the following directives:

usage
exit (MUST be the last directive in the series)

They are used to define conversion rules for input data where a name encountered in the data is to be transformed into another name before it is processed.

7.3.1.16.1 The `usage` directive

This directive has the general form:

READ=<group identifier> (MUST be specified)
NAME=<group identifier> (MUST be specified)

This directive defines how one particular name should be converted. You can repeat this directive as often as you like in a definition.

7.3.1.16.1.1 The `read=` parameter

This parameter expects a parameter value consisting of a valid $\kappa\lambda\epsilon\iota\omega$ name. As soon as the relevant `connection` definition is activated, the system watches for the appearance of this name in the input data.

7.3.1.16.1.2 The `name=` parameter

This parameter also expects a parameter value consisting of a valid $\kappa\lambda\epsilon\iota\omega$ name. This name is used to replace the name you specified in `read=` (i.e. the name in the input data) before the data is passed on to other parts of the system for further processing.

7.3.1.17 The `image` definition

Definitions with the parameter value `usage=image` accept the following directives:

part
device
volume
image
visibility
exit (MUST be the last directive in the series)

You use this definition in order to instruct the system how to proceed when setting up a link with a digitally stored image.

Images stored in digital form take up more storage space – by many times – than any other type of data managed by $\kappa\lambda\epsilon\iota\omega$. For this reason, they are not stored in the database itself. The database only holds a symbolic name for each image known as an *image file name*.

Whenever the system is instructed to display the image, it attempts to find a file containing this image on your computer.

These linkages are managed in the form of `image` definitions.

To administer images, $\kappa\lambda\epsilon\iota\omega$ uses a two-level abstraction of the equipment which is actually available on a given machine.[2]

- A *device* describes a specific base directory, from which the directory holding the actual images is dependent. In small image databases such a directory could be an actual directory on your hard disk. When removable disks or CD-ROMs are used, this directory would be the *mount point* for this device.
- On each device, there exists at least one *volume* which contains the actual images. This volume is – to provide a minimal amount of protection against accidental deletion – an *invisible* directory. On UNIX systems this means, that the name of the directory starts with a period ("`.`"). By default the first twelve characters of the directory name after the period are used within all `image` directives as the name of the volume.
  - If a device is realized on a non-removable disk, it contains exactly one volume, which is required to be called `harddisk` (and which has to exist as a directory `.harddisk` therefore).
  - On a removable device it is expected, that each exchangeable medium (each magnetooptical cartridge, each CD-ROM), contains a directory with the name of the device after a leading period, in which the actual images are stored.
  - By a slight modification of this logic, a volume can also be assigned a completely arbitrary directory name, which therefore does not have to be invisible. This is primarily provided for the direct import of photo CDs and similar devices into the system, where the user has no control over the directory names created.

[2] All image processing makes sense only with version 6 of $\kappa\lambda\epsilon\iota\omega$, which currently runs only on various versions of UNIX. The following description describes all concepts therefore in the terminology of this operating system.

When $\kappa\lambda\epsilon\iota\omega$ discovers that an image which is needed resides on another volume of a device than the one currently mounted on it, it will open an operating system window for you and prompt you to mount the appropriate volume. As for that purpose the system has to use the native colour scheme of your console, this may imply temporarily to change the colours on the image processing screens providing a strange picture.

In order to speed up the linear processing of large stocks of images (e.g. in order to restrict the processing of images scattered throughout an entire database to the contents of the last data storage medium you set up for storing visible images, without constantly having to load other storage media in the meantime), you may decide you want to exclude individual storage media from the processing, albeit temporarily. Images excluded in this way are said to be *invisible*, i.e. the system will suppress any attempt to display them.

7.3.1.17.1 The `part` directive

This directive has the general form:

TYPE=Insertion | Creation | Substitution | Destruction	**(default: Insertion)**
DEVIce=<device name>	**(default: none)**
VOLUme=<volume name>	**(default: none)**

It defines the conditions for all subsequent directives.

This directive may appear any number of times in a definition.

7.3.1.17.1.1 The `type=` parameter

This parameter expects as a parameter value one of the keywords `insertion`, `creation`, `substitution` or `destruction`. All of them can be shortened to their respective first character.

The other directives of an `image` definition are executed differently according to the value selected for this parameter.

7.3.1.17.1.2 The `device=` parameter

This parameter expects as a parameter value the name of a logical device defined *before this directive is encountered* on a `device` directive. This parameter value will be used on all successive directives, where a `device=` parameter is legal, has, however, not been specified. This is known as the *current default device.*

7.3.1.17.1.3 The `volume=` parameter

This parameter expects as a parameter value the name of a logical volume on the current default device defined *before this directive is encountered* on a `volume` directive. This parameter value will be used on all successive directives, where a `volume=` parameter is legal, has, however, not been specified. This is known as the *current default volume.*

7.3.1.17.2 The `device` directive

This directive has the general form:

NAME=<device name>	**(MUST be specified!)**
USAGe=Permanent | Temporary	**(default: Permanent)**
TYPe=Temporary | Permanent	**(default: Temporary)**
TARGet=<directory name>	**(default: none)**

It defines, modifies or deletes a logical device. The operations performed depend on the current mode of the definition, defined by the `type=` parameter of the `part` directive.

- `part type=creation`:
  The directory specified by `target=` is created, if it does not exist already. If `usage=permanent` a directory `.harddisk` is created in that directory and an appropriate entry for this logical volume is made in the `image` definition.
  Further execution is aborted, if a device with that name has already been defined within the `image` definition.
- `part type=insertion`:
  The directory specified by `target=` has to exist. If `usage=permanent` it has to contain a subdirectory `.harddisk`.
  Further execution is aborted, if a device with that name has already been defined within the `image` definition.
- `part type=substitution`:
  If no device with that name has yet been created within the `image` definition, this mode behaves as `type=insertion`.
  Otherwise the previous value of the `target=` parameter is replaced by the current one. All volumes defined for this device are left as they are: this mode is primarily provided to support changing the mount points as may be required.
- `part type=destruction`:
  The logical device is deleted within the `image` definition. Further actions depend on the value of the `type=` parameter of the `device` directive. If
  - `type=temporary` all volumes on the device are left as they are, allowing a later insertion into another `image` definition.
  - `type=permanent` asks the system to delete all images in all volumes. With exchangable media this will require mounting all volumes defined for this device.

7.3.1.17.2.1 The `name=` parameter

This parameter expects as value a standard $\kappa\lambda\epsilon\iota\omega$ name. It will be used to address this device in all following directives.

7.3.1.17.2.2 The `target=` parameter

This parameter expects as value a constant. It represents a valid directory on the host computer or reachable over a network. It is strongly recommended to specify all directories absolutely, that is starting with a slash to indicate the root directory. For removable devices this directory has to be a valid mount point for the host computer.

7.3.1.17.2.3 The `usage=` parameter

This parameter expects the keywords `permanent` or `temporary` as values. Both of these can be abbreviated to their respective first character.

- `permanent` instructs the system to consider this device as a hard disk which will be accessible at all times. This is the default assumption.
- `temporary` instructs the system to consider the device as removable.

7.3.1.17.2.4 The `type=` parameter

This parameter expects the keywords `temporary` or `permanent` as values. Both of these can be abbreviated to their respective first character.

- `temporary` instructs the system to leave all volumes defined on this device untouched in physical storage. Volumes and images defined for this device are deleted logically only. This is the default assumption.
- `permanent` instructs the system to delete all images and volumes defined for this device physically, that is remove them from the media on which they reside.

7.3.1.17.3 The `volume` directive

This directive has the general form:

NAME=<volume name>	**(MUST be specified!)**
TYPe=Temporary | Permanent	**(default: Temporary)**
TARGet=<directory name>	**(default: none)**
DEVIce=<device name>	**(default: none)**

It defines, modifies or deletes a logical volume. The operations performed depend on the current status of the definition, defined by the `type=` parameter of the `part` directive. The name of the directory for the volume is arrived at in the following way: (a) if a `target=` parameter has been specified, that name is used unchanged. (b) Otherwise the internal name of the logical volume is used as directory name, made "invisible" by prefixing it with a period "`.`".

- `part type=creation`:
  The directory for the volume is created on the current device, if it does not exist already. An appropriate entry for this logical volume is created.
  Further execution is aborted, if a volume with that name has already been defined for the current device within the `image` definition.
- `part type=insertion`:
  The directory for the volume `target=` has to exist on the current device.
  Further execution is aborted, if a volume with that name has already been defined within the `image` definition.
- `part type=substitution`:
  If no volume with that name has yet been created within the `image` definition, this mode behaves as `type=insertion`.
  Otherwise the previous name of the directory for the volume is replaced by the current one. All images defined for this volume are left as they are: this mode is primarily provided to support changing the organisation of removable volumes. As a consequence the user is responsible for performing all renaming operations on the volumes changed, *keeping in mind that a directory derived from the* `name=` *parameter must be invisible*.
- `part type=destruction`:
  The logical volume is deleted within the `image` definition. Further actions depend on the value of the `type=` parameter of the `volume` directive. If
  - `type=temporary` all images on the volume are left as they are, allowing a later insertion into another `image` definition.
  - `type=permanent` asks the system to delete all images in the volume. With exchangable media this will require mounting the volume on its device.

7.3.1.17.3.1 The `name=` parameter

This parameter expects as value a standard κλειω name. It will be used to address this volume in all following directives.

7.3.1.17.3.2 The `target=` parameter

This parameter expects as value a constant. It represents a valid directory name. This name should *not* start with a slash, as it is supposed to be relative to the directory representing the device.

7.3.1.17.3.3 The `type=` parameter

This parameter expects the keywords `temporary` or `permanent` as values. Both of these can be abbreviated to their respective first character.

- `temporary` instructs the system to leave all files defined on this volume untouched in physical storage. Images defined for this volume are deleted logically only. This is the default assumption.
- `permanent` instructs the system to delete all images defined for this volume physically, that is remove them from the media on which they reside.

7.3.1.17.3.4 The `device=` parameter

This parameter expects as value the name of a device which has been defined before this directive is encountered. It specifies on which device the volume has to be mounted. If this parameter is missing, the current default device is used.

7.3.1.17.4 The `image` directive

This directive has the general form:

NAME=<image file name>	**(MUST be specified!)**
TYPe=Temporary | Permanent	**(default: Temporary)**
SOURce=<directory name>	**(default: none)**
DEVIce=<device name>	**(default: none)**
VOLUme=<volume name>	**(default: none)**
WRITe=<character string>	**(default: the image name)**

It defines, modifies or deletes an image file. The operations performed depend on the current status of the definition, defined by the `type=` parameter of the `part` directive.

- `part type=creation`:
  A file with the file name specified by the `name=` parameter is created on the current volume. The contents of the file specified by the `source=` parameter are copied into it. An appropriate entry for the image is created.
  If an image with that name has already been defined for the current device within the `image` definitio, an error message is displayed.
- `part type=insertion`:
  The image file has to exist on the current volume. The `source=` parameter is illegal in this mode.
  If an image with that name has already been defined for the current device within the `image` definitio, an error message is displayed.
- `part type=substitution`:
  If no image file with that name has yet been created within the `image` definition, this mode behaves as `type=insertion`.
  Otherwise the existing image file is deleted and replaced by the contents of the file specified by `source=`.
- `part type=destruction`:
  The image file is deleted within the `image` definition. Further actions depend on the value of the `type=` parameter of the `volume` directive. If
  - `type=temporary` the image is left as it is.
  - `type=permanent` asks the system to delete the image from storage. With exchangable media this will require mounting the relevant volume.

7.3.1.17.4.1 The `name=` parameter

This parameter expects as value a file name for the image. The same name identifies the image within the database. At the time of printing of this manual it is therefore restricted to a length of twelve characters.

7.3.1.17.4.2 The `source=` parameter

This parameter expects as value a constant. It represents a valid file name on the host computer. The contents of this file are copied into the file describe by the `name=` parameter. *No tests are performed, whether this file actually does contain a recognisable digital image.*

7.3.1.17.4.3 The `type=` parameter

This parameter expects the keywords `temporary` or `permanent` as values. Both of these can be abbreviated to their respective first character.

- When `part type=create` or `substitution`:
  - `temporary` instructs the system to leave the file defined by the `source=` parameter as it is. This is the default assumption.
  - `permanent` instructs the system to delete this file after its contents have been successfully copied unto the appropriate volume.
- When `part type=destruction`:
  - `temporary` instructs the system to delete the definition of the image, leave the physical file on its volume, though. This is the default assumption.
  - `permanent` instructs the system to delete the file from its volume.

7.3.1.17.4.4 The `device=` parameter

This parameter expects as value the name of a device which has been defined before this directive is encountered. It specifies on which device the volume containing the image has to be mounted. If this parameter is missing, the current default device is used.

7.3.1.17.4.5 The `volume=` parameter

This parameter expects as value the name of a volume which has been defined before this directive is encountered. It specifies on which volume the image will be kept. If this parameter is missing, the current default volume is used.

7.3.1.17.4.6 The `write=` parameter

It expects as value a constant. This constant will be displayed by *κλειω* instead of the image file name, whenever a representation of the image in the data type `text` is needed.

7.3.1.17.5 The `visibility` directive

This directive has the general form:

NAME=<image file name>	**(MUST be specified!)**
DEVIce=<device name>	**(default: none)**
VOLUme=<volume name>	**(default: none)**
VISIbility=Yes | No	**(default: toggle current status)**

This directive changes the visibility of

a) all the images on all the volumes on a device, if the name of that device is given with the `device=` parameter.

b) all the images on a volume, if the name of that volume is given with the `volume=` parameter.

c) of an individual image, if the name of that image is given with the `name=` parameter.

At least one of these three parameters has to be specified.

The visibility assigned to the set of images selected is arrived at in the following way:

- If `visibility=yes` all selected images are declared visible.
- If `visibility=no` all selected images are declared invisible.
- If `visibility=` is missing the visibility of all selected images is toggled, that is, visible images become invisible and viceversa.

7.3.1.18 The `chronology` definition

Definitions with the parameter value `usage=chronology` accept the following directives:

name
conversion
training
exit **(MUST be the last directive in the series)**

They are used to define a calendar of saints' days, i.e. to define the rules required to "deal with" references to a locally prevailing saints' day calendar.

7.3.1.18.1 Syntax required for saints' day dates

7.3.1.18.1.1 Simple calendar terms

For this purpose you define *calendar terms* which can be linked to a valid expression by a set of limited rules.

Calendar terms are either numbers or fragments of text. Numbers and fragments of text should always be separated from one another by blank spaces.

Fragments of text may themselves contain spaces: for this reason, you can string as many of them together as you like. The system interprets several spaces in succession as *one* blank space. No distinction is made between upper-case and lower-case characters in fragments of text.

Numbers may be closed by full stops (periods), but this is not obligatory. If you do terminate a number with a full stop (period) it should appear immediately after the number, i.e. there should be no intervening space.

You can use the following calendar terms of the "number" type in order to formulate the dates of religious festivals (holy feast days):

- $< Year >$
  A number which can be interpreted as a year in the Christian calendar according to the rules applying to `date` data types.
  Example: `Wednesday after Oculi` **`1567`**
- $< Number >$
  Any number representing a count of feast days with the same name or describing an interval of time before or after a feast day.
  Example: **`3rd`** `Sunday after Pentecost 1567`
  `2` `days after All Saints' Day 1567`

The user should declare calendar terms of the "textual fragment" type in one of the directives described below. The system differentiates the following types of textual fragment:

- $< Feast\ Day >$
  A textual fragment designating a feast day (religious festival) in the ecclesiastical year.
  Examples: `Wednesday after` **`Oculi`** `1567`
  `Johannis apost.` `1567`
- $< Weekday >$
  A textual fragment designating a day of the week.

Examples: **Wednesday** `after Oculi 1567`
The Lord's Day `after Chrysostomus 1567`

- $< Day >$
  A textual fragment representing the term "day".
  Example: `2` **days** `after All Saints' Day 1567`
- $< Octave >$
  A fragment of text representing the term "octave".
  Examples: **octave** `before Walpurgis 1567`
  octave `after Walpurgis 1567`
- $< Eve >$
  A textual fragment representing the term "eve".
  Examples: **eve of** `Walpurgis 1567`
  `Walpurgis`**eve** `1567`
- $< Proximity >$
  A fragment of text representing the terms "before(hand)" or "after(wards)".
  Example: `Wednesday` **after** `1567`

7.3.1.18.1.2 Composite calendar terms

The system combines these simple calendar terms to form valid dates. To make it easier to write the rules whereby this is done, we can combine a series of these simple calendar terms to form two *composite calendar terms*, $< Day\,Count >$ and $< Feast >$.

$< Day\,Count >$ is defined as:

$$< Day\,Count > \quad = \quad [< number >] \left\{ \begin{array}{c} < Weekday > \\ < Day > \end{array} \right\}$$

This composite calendar term corresponds to expressions such as:

```
3 days
Wednesday
3rd Sunday
```

$< Feast >$ is defined as:

$$< Feast > \quad = \quad \left[\left\{ \begin{array}{c} < Octave > \\ < Eve > \end{array} \right\} \right] \quad < Feastday > \left[\left\{ \begin{array}{c} < Octave > \\ < Eve > \end{array} \right\} \right]$$

Semantic restriction: both $< Octave >$ *and* $< Eve >$ *may only appear* either *in front of* or *after* $< Feast\,Day >$.

This composite calendar term corresponds to expressions like:

```
Walpurgis octave
Walpurgis Eve
Walpurgis
```

7.3.1.18.1.3 Dates of religious festivals (feast days)

A *feast day date* is an expression composed of calendar terms, constructed according to the following rule:

$$\Big[[< Day\,Count >< Proximity >]\ < Day\,Count >\ [< Proximity >]\Big]\ < Feast >< Year >$$

Hence a feast day date corresponds to expressions like:

```
Pentecost 1567
Whitsun 1567
Sunday after Pentecost 1567
2nd Sunday after Pentecost 1567
3 days before the 2nd Sunday after Pentecost 1567
```

7.3.1.18.1.4 Periods of time

According to the rules for the `date` data type, every $\kappa\lambda\epsilon\iota\omega$ date potentially consists of a *terminus ante quem* and a *terminus post quem*. Expressions written like this:

`Walpurgis 1567 - Johannis 1567`

are *not* supported by the current implementation of the feast day dating function, however.

Chronology definitions, on the other hand, do support a special calendar term of the textual fragment type, $< Period >$. A textual fragment of this type describes a period of time relating to a single date.

Example: **week after** `Pentecost 1567`

Depending on how it is defined, this calendar term is converted into a pair of dates representing a period of time by the appropriate directives in the chronology definition. So we could convert the above example into:

`Monday after Pentecost 1567`
or `Sunday after Pentecost 1567`.

Both dates must satisfy the specified rules for the formulation of feast day dates: hence their validity clearly depends on the conversion rules you declare. Consequently, it is not possible to specify an appropriate syntax for these calendar term independently of the rules you defined. Expressions written as follows, however:

$$< Period > \quad < Feast\,Day > \quad < Year >$$

are always transformed into correctly interpreted periods of time unless the rules for interpreting them are themselves incorrect.

7.3.1.18.2 Individual directives for the `chronology` definition.

7.3.1.18.2.1 The `name` directive

The general form of the `name` directive is:

NAME=<character string>	**(MUST be specified)**
DATE=<day specification>	**(default: none)**
EASTer=<number>	**(default: none)**
WEEKday=SUNDay | MONDay | TUESday | WEDNesday | THURsday | FRIDay | SATUrday | ABSOlute | OCTAve | PREVious	**(default: none)**
AFTEr=Inclusive | Exclusive	**(default: none)**
BEFOre=Inclusive | Exclusive	**(default: none)**
FORM=<character string>	**(default: none)**
FIRSt=<character string>	**(default: none)**
SECOnd=<character string>	**(default: none)**
NULL=No | Yes	**(default: No)**

It allows you to define a calendar term of the textual fragment type. This directive may appear as often as you like in a chronology definition.

The way you combine your parameters determines the kind of calendar term defined by this directive. This means that there is not much point in discussing each parameter separately. Instead all possible parameter combinations will be discussed in the following sections. Any combination of parameters which is not discussed below is invalid.

7.3.1.18.2.1.1 Calendar terms of the $<$ *Feast Day* $>$ type

You can define a $<$ *Feast Day* $>$ as a particular day in the year, a period of time in days from Easter Sunday, or an expression relating to another day.

7.3.1.18.2.1.1.1 Defining fixed feast day dates

You attach a church feast day to a particular day of the year by assigning values to a `name` directive's `name=` and `date=` parameters. The `name=` parameter's value is the name of the feast day, and the `date=` parameter's value consists of a pair of numbers separated by a full stop (period). The first number defines the day, the second the month.

Example:

```
name name="Walpurgis";date=1.5
```

7.3.1.18.2.1.1.2 Defining feast days in relation to Easter

You can define a church feast day in relation to Easter by assigning values to a `name` directive's `name=` and `easter=` parameters. If the feast day corresponding to the identifier you assigned to the `name=` parameter happens *before* Easter, you should assign a negative numerical value to the `easter=` parameter; if the feast day occurs *after* Easter, you should assign a positive value to the `easter=` parameter.

Examples:

```
name name="Oculi";easter=-28
name name="Easter";easter=0
name name="Pentecost";easter=49
```

7.3.1.18.2.1.1.3 Defining movable feast day dates

You can define a church feast day in relation to another feast day by assigning values to a `name` directive's `name=` and `form=` parameters. If the system encounters the `name=` parameter's value in the text, it replaces it with the value of the `form=` parameter and then interprets the latter with the help of the remaining chronology definition. This process is recursive: thus you can assign the value which you assigned to `form=` in this directive to the `date=` parameter in another directive of this type.

Examples:

```
name name="festival";form="Sunday after St. Thomas the Apostle"
name name="beginning of Advent";form="5th Sunday before Christmas"
name name="Advent";form="Sunday after the beginning of Advent"
```

7.3.1.18.2.1.1.4 Defining an unknown < *Feast Day* >

If you do not know where to allocate a feast day in your calendar, you can declare it by assigning the relevant identifer to the `name=` parameter in a `name` directive, any day of the month to the directive's `date=` parameter and the keyword `yes` to its `null=` parameter.

A feast day date which includes this calendar term as < *Feast Day* > is interpreted as "0.0" in the relevant year.

Example:

```
name name="St. Noman";date=1.1;null=yes
```

7.3.1.18.2.1.2 Calendar terms of the < *Weekday* > type

You can define calendar terms of this type by assigning values to a `name` directive's `name=` and `weekday=` parameters. For the purposes of this definition, the `weekday=` parameter may assume one of the following values: `sunday` ("Sunday"), `monday` ("Monday"), `tuesday` ("Tuesday"), `wednesday` ("Wednesday"), `thursday` ("Thursday"), `friday` ("Friday") or `saturday` ("Saturday").

Examples:

```
name name="Sunday";weekday=sunday
name name="Lord's day";weekday=sunday
```

7.3.1.18.2.1.2.1 Defining an unknown < *Weekday* >

If a particular weekday is illegible or incomprehensible, you can still declare it by assigning any weekday to the relevant fragment of text using the `weekday=` parameter and then assigning the keyword `yes` to the `null=` parameter.

The system interprets a feast day date which includes this textual fragment as "0.x" in the year in question, where "x" represents a month which can be derived from the remaining specifications.

Example:

```
name name="Scribbleday";weekday=saturday;null=yes
```

7.3.1.18.2.1.3 Calendar terms of the $< Proximity >$ type

You can define calendar terms of this type by assigning a value to a `name` directive's `name=` parameter and *either* to its `after=` *or* `before=` parameters. `after=` and `before=` expect the keywords `exclusive` or `inclusive` as parameter values. These keywords specify whether the relevant fragment of text should describe the period starting on the first day after/before the next $< Feast >$ (`exclusive`), or starting on the $< Feast >$ itself (`inclusive`). In normal applications you should *always* specify `exclusive`. As a rule, you should only use the parameter value `inclusive` to define periods of the type discussed in section 7.3.1.18.2.1.7.

Example:

```
name name="before";before=exclusive
```

7.3.1.18.2.1.3.1 Defining an unknown $< Proximity >$

If you want to declare an illegible or incomprehensible $< Proximity >$, you can do so by first assigning any meaning to a corresponding fragment of text using the `after=` or `before=` parameters, then assigning the keyword `yes` to the `null=` parameter. In this context, `after=` and `before=` are interchangeable. The system interprets a feast day containing this fragment of text as "0.x" of year in question, where "x" represents a month which can be derived from the remaining specifications.

Example:

```
name name="p.";before=exclusive;null=yes
```

7.3.1.18.2.1.4 Calendar terms of the $< Day >$ type

You can define calendar terms of this type by assigning the textual fragment to a `name` directive's `name=` parameter and assigning the keyword `absolute` to its `weekday=` parameter.

This system then calls in this textual fragment in order to specify intervals of time in days. Example:

```
name name="days";weekday=absolute
```

7.3.1.18.2.1.5 Calendar terms of the $< Octave >$ type

You can define calendar terms of this type by assigning the fragment of text to a `name` directive's `name=` parameter and assigning the keyword `octave` to its `weekday=` parameter.

If it appears immediately before or after a $< Feast\,Day >$, the system interprets it as a modifier for the specified $< Feast\,Day >$'s octave.

Example:

```
name name="octave";weekday=octave
```

7.3.1.18.2.1.6 Calendar terms of the < *Eve* > type

You can define calendar terms of this type by assigning the fragment of text to a `name` directive's `name=` parameter and assigning the keyword `previous` to its `weekday=` parameter.

If it appears immediately before or after a < *Feast Day* > the system interprets this fragment of text as a modifier for the eve of the specified < *Feast Day* >.

Example:

```
name name="eve";weekday=previous
```

7.3.1.18.2.1.7 Calendar terms of the < *Period* > type

You can define calendar terms of this type by assigning values to a `name` directive's `name=`, `first=` and `second=` parameters. As a rule, the character strings assigned to `first=` and `second=` should each contain one asterisk. The value you assign to the `name=` parameter corresponds to the identifier of the period you want to analyse.

The system proceeds as follows when it is interpreting feast day dates:

- The part of the date following < *Period* > is inserted in place of the asterisk in the fragment of text assigned to `first=`. The system then evaluates the resulting feast day date and treats it as the period's *terminus post quem*.
- Similarly, `second=` provides the period's *terminus ante quem*.

The mechanism discussed in section 7.3.1.18.2.1.3 allows you to define calendar terms which are treated as "inclusive" before and after terms.

Example:

```
name name="week after";first="1 day after *";
      second="Sunday after *"
```

If you issued this directive, the following date in the input data:

```
week after Martinmas
```

would be interpreted as:

`1 day after Martinmas` *until* `Sunday after Martinmas`

7.3.1.18.2.2 The `conversion` directive

The general form of the `conversion` directive is:

PREParation=<name of definition> (MUST be specified!)

It allows you to arrange for a holy feast day date to be simplified by means of a `conversion` definition before it is analysed in accordance with the rules specified by the `name` directive.

7.3.1.18.2.2.1 The `preparation=` parameter

This parameter expects the name of a `conversion` definition as a value. The associated logical object does not have to exist at the very moment it is referred to but should exist by the time the system starts to execute the `chronology` definition at the very latest. If the `conversion` definition still does not exist even at this very last possible moment the system refuses to execute the program and issues an error message. Furthermore, you can only use the `training` directive if the logical object already exists.

If you specify this parameter the system uses the named `conversion` definition to pre-process the dates of feast days which are awaiting further processing. *Only when this procedure is complete* will the system interpret them.

7.3.1.18.2.3 The `training` directive

The general form of the `training` directive is:

EXPLain=No | Yes (default: No)

You can specify this directive as a way of testing the chronology definition. $\kappa\lambda\epsilon\iota\omega$ expects every subsequent line to contain exactly one feast day date, which is then converted into a date written in the conventional way. The system terminates this mode as soon as you enter a line starting with an asterisk: $\kappa\lambda\epsilon\iota\omega$ then expects another directive from a chronology definition (as a rule, an `exit` directive).

7.3.1.18.2.3.1 The `explain=` parameter

The `explain=` parameter accepts the keywords `yes` or `no` as values, both of which can be shortened to their respective first character.

- `no` explicitly requests the default setting: the specified holy feast day date is printed out followed by any error messages. The result of the conversion is then printed out.
- `yes` instructs the system to provide a considerably more precise report on the conversion. It therefore informs you which calendar terms were applied.

7.3.1.19 The `classification` definition

Definitions with the parameter value `usage=classification` accept the following directives:

part
sign
exit (**MUST be the last directive in the series**)

7.3.1.19.1 The purpose of embedded classifications

The purpose of this definition is to provide a system of *embedded classifications* within a text that is administered by a $\kappa\lambda\epsilon\iota\omega$ database within elements of the data type `text`. Such classifications are transparent for most $\kappa\lambda\epsilon\iota\omega$ modules: that is, the embedded symbols are treated as any other character string. The `catalogue` command allows the user, however, to activate the embedded classifications by specifying a `classification` definition to be applied to the text when a catalogue is created.

Embedded classifications can be used for two purposes: on the one hand to integrate linguistic knowledge into a full-text database, and on the other to apply overlapping hierarchies to a text which is administered within a $\kappa\lambda\epsilon\iota\omega$ database. An example for the later case might be a database of medieval charters, where the structure of the text is realised as a $\kappa\lambda\epsilon\iota\omega$ data structure, while topographical names, proper names and legally significant phrases are marked up by embedded classifications.

Classification symbols start with data signal character 10 (defaulting to the backslash - "\"). They are followed by a user-defined character string and end with a pair of curly brackets, which may contain additional information.

Classification symbols form *subclassifications* within a classification, which group together symbols for mutually exclusive concepts. A subclassification can be binary – that is, express the absence or presence of a certain quality – or consist of arbitrarily many different classes.

An example for a binary subclassification to be applied to a text might be "illegible"; an example for a multiple-valued one "role of a person". Subclassifications can overlap each other: each word in a text can belong to different classes in arbitrarily many subclassifications.

Let us assume the following user defined symbols:

- A binary subclassification *illegible* indicated by `IL`.
- A multiple-valued subclassification *role* with the classes
  - *husband* indicated by `HUS`,
  - *wife* indicated by `WIFE` and
  - *child* indicated by `CH`.

These classes could be used to mark up a short piece of text as follows:

```
\HUS{} John Smith and \WIFE{} Martha \IL{+}Fow??ler
scribble \IL{-} the birth of \CH{} their son Michael.
```

While this example is not complete, it shows the basic mechanism:

Rule 1: for a multiple-valued subclassification we announce which of the classes is applicable by specifying the indicator for the appropriate class.

Rule 2: for a binary subclassification we announce that the indicated property is present by a plus sign in the pair of curly brackets and that it ceases to be present by a minus sign.

When applied to a catalogue, the above example would be classified as follows:

Term	illegible	role
`John`	unknown	husband
`Smith`	unknown	husband
`and`	unknown	husband
`Martha`	unknown	wife
`Fow??ler`	yes	wife
`scribble`	yes	wife
`the`	no	wife
`birth`	no	wife
`of`	no	wife
`their`	no	child
`son`	no	child
`Michael`	no	child

This classification is obviously faulty in two respects: (a) the multiple-valued classification should be reset to "unknown" between portions of the text which are not clearly refering to any one individual and (b) we should either reset the legibility indicator to "unknown" as well, or start already with it being set to "no".

This can be done by the following markup:

```
\HUS{} John Smith \HUS{-} and \WIFE{} Martha \IL{+}Fow??ler
scribble \{} the birth of \CH{} their son Michael.
```

which applies the following rules:

Rule 3: when a class of a multiple-valued subclassification is invoked with a minus sign within the curly brackets (\{−}), that subclassification is reset to "unknown".

Rule 4: the symbol \{} resets *all* subclassifications to "unknown".

As a result we would get:

Term	illegible	role
`John`	unknown	husband
`Smith`	unknown	husband
`and`	unknown	unknown
`Martha`	unknown	wife
`Fow??ler`	yes	wife
`scribble`	yes	wife
`the`	unknown	unknown
`birth`	unknown	unknown
`of`	unknown	unknown
`their`	unknown	child
`son`	unknown	child
`Michael`	unknown	child

A final rule should be mentioned:
Rule 5: all classifications transcend database entities: If additional text from later elements get into the catalogue, the classifications at the end of our example would remain in force.

7.3.1.19.2 Individual directives

7.3.1.19.2.1 The `part` directive

The general form of the `part` directive is:

NAME=<character string>	**(MUST be specified)**
MORE=No | Yes	**(default: No)**
WITHout=<character string>	**(MUST be specified)**
WRITe=Yes | No	**(default: Yes)**

This directive is used to start the definition of a new subclassification. A `part` directive has to be specified before the first `sign` directive is encountered.

7.3.1.19.2.1.1 The `name=` parameter

This parameter expects a constant as value. This constant is displayed whenever the value of this subclassification is reported. (Usually by the full-text system.)

7.3.1.19.2.1.2 The `more=` parameter

The `more=` parameter accepts the keywords `no` or `yes` as values, both of which can be shortened to their respective first character.

- `no` explicitly requests the default setting: the subclassification is assumed to be binary.
- `yes` declares the subclassification to be a multiple-valued one.

7.3.1.19.2.1.3 The `without=` parameter

This parameter expects a constant as its value. This constant will always be displayed, when the value of this subclassification is required for a word where it is unknown. If this parameter is not specified the text "`*(unknown)`" will be displayed in such a situation.

7.3.1.19.2.1.4 The `write=` parameter

The `write=` parameter accepts the keywords `yes` or `no` as values, both of which can be shortened to their respective first character.

- `yes` instructs $\kappa\lambda\epsilon\iota\omega$ to display this subclassification, even if no class has been selected: which value will be displayed depends on the use made of the `without=` parameter above. This is the default behaviour of the system.
- `no` instructs the system to suppress the subclassification if no value has been specified for it.

7.3.1.19.2.2 The `sign` directive

The general form of the `sign` directive is:

SIGN=<character string> **(MUST be specified)**
WRITe=<character string> **(default: the value of SIGN=)**

This directive is used to define a specific class within a subclassification. A series of `sign` directives, following a single `part` directive, define a subclassification between them.

7.3.1.19.2.2.1 The `sign=` parameter

This parameter expects a constant as value. This constant is expected within the input data as indicator for the class to be defined.

7.3.1.19.2.2.2 The `write=` parameter

This parameter expects a constant as value. This constant is displayed on output whenever this class has to be displayed.

7.3.1.19.2.3 Example

The classification discussed within the introductory section 7.3.1.19.1 would be defined by the following definition:

```
item name=example;usage=classification

part name="illegible"
sign sign="IL"

part name="role";more=yes
sign sign="HUS";write="Husband"
sign sign="WIFE";write="Wife"
sign sign="CH";write="Child"

exit name=example
```

7.3.2 The `delete` command

The general form of the `delete` command is:

NAME=<Name of object>	**(MUST be specified!)**
USAGe=TEXT | NUMBer | DATE | CATEgory | RELAtion | CONVersion | SOUNdex | SKELeton | GUTH | SUBStitution | CODEbook | CATAlogue | SOURce | KEYWord | ORDEr | CONNection | LOCAtion | IMAGe | BRIDge | CHROnology	**(MUST be specified!)**
TYPE=Temporary | Permanent	**(default: Temporary)**
SOURce=<name of database>	**(default: the database being processed currently)**
CATAlogue=<name of catalogue>	**(default: the catalogue being processed currently)**
EACH=No | Yes	**(default: No)**

It is used to remove (= delete) logical objects from your local and/or permanent environment. In addition, it offers you the option of deleting databases painlessly at operating system level; an option which we would strongly recommend as the best way of performing deletions.

7.3.2.1 The `name=` parameter

This parameter expects you to specify the name of a logical object or database. The command destroys this logical object or database.

You must specify this parameter; as a safeguard, the system does not allow you to delete an entire class of logical objects.

7.3.2.2 The `usage=` parameter

This parameter expects a parameter value consisting of one of the keywords `text`, `number`, `date`, `category`, `relation`, `conversion`, `soundex`, `skeleton`, `guth`, `substitution`, `codebook`, `catalogue`, `keyword`, `source`, `order`, `connection`, `location`, `image`, `bridge` or `chronology`, all of which can be shortened to their respective first four characters.

It specifies which of a logical object's classes should be deleted. You can also use this parameter to delete up to four classes of logical object which cannot be introduced by the `item` command, or not by this command alone.

- `codebook` deletes a codebook, regardless of whether it was generated by `item` or `create`.
- `catalogue` deletes a catalogue generated by the `catalogue` command. If you want to do this you must also specify `type=permanent`, as a safeguard.
- `keyword` removes a list of references stored in a catalogue's reference library from the latter. If you had not yet used the `catalogue[]` built-in function to allocate a catalogue, you must explicitly specify a catalogue in the `catalogue=` parameter.[3]

[3] At the present time – and for some time to come, because the feature in question is deeply embedded in the system – deleting a list of references may result in inconsistent behaviour on the

- `source` deletes a complete database, together with any physical files which are dependent on that database. For this purpose and as a safeguard, you must also specify `type=permanent`.
- `bridge` removes the linkage between the database specified in `source=` and the database with the name specified in `name=`. Removing this linkage safeguards the consistency of both databases. That is to say, κλειω will only delete the database if the system is also able to remove the two-way linkage between the database specified in `name=` and the database specified in `source=`.
  The system can only do this if both databases are accessible on your hard disk.

If you do not specify this parameter, the system deletes all logical objects with the given name, regardless of their class. *However, the system will only delete a database, if you explicitly specify the latter's name.*

This parameter accepts a series of additional keywords. However they do cause the system to display the following error message: `System feature not yet implemented: item`.

7.3.2.3 The `type=` parameter

The `type=` parameter expects you to specify one of the keywords `permanent` or `temporary`. Both of them can be shortened to their respective first character.

- `temporary` means that the logical object is only deleted from the local environment, i.e. instructs the system to behave as it would by default.
- `permanent` means that the logical object is also deleted from your database's permanent environment. If you have not yet addressed a particular database using `query`, you must specify the database's name using `source=`.

7.3.2.4 The `source=` parameter

The `source=` parameter expects the name of a database as a parameter value. If you also specify `type=permanent`, the system deletes the designated logical objects from this database's permanent environment.

Once the system has processed this command, the linkage with this database is broken again: so anything you allocated in a preceding `query` command remains unchanged.

7.3.2.5 The `catalogue=` parameter

The `catalogue=` parameter expects the name of a catalogue as a parameter value. If you specify the `usage=keyword` parameter, the indicated reference list is deleted from this catalogue's reference library.

Once the system has processed this command, it breaks the link with this catalogue: so anything you may have allocated previously to another catalogue remains unchanged.

7.3.2.6 The `each=` parameter

The `each=` parameter accepts the keywords `no` or `yes` as values, both of which can be shortened to their respective first character.

---

part of the system in one particular situation: even if you delete the last of your previously saved lists of references, the main menu will still display a list of all the lists of references you saved earlier. If you select this menu item the system will then inform you that there are no stored lists of references.

This parameter is meaningful only, if the `usage=` parameter has the value `image`.

`no` means that the `image` definition is deleted; all the physical image files contained within it will be preserved, though. This is the default behaviour of the system. `yes` means that all physical image files described within the `image` definition will be deleted, before the definition is finally destroyed. This may imply the mounting of all removable media described within the definition.

7.3.3 The `describe` command

The general form of the `describe` command is:

NAME=<name of object>	**(default: none)**
USAGe=TEXT | NUMBer | DATE | CATEgory | RELAtion | CONVersion | SOUNdex | SKELeton | GUTH | SUBStitution | CODEbook | CATAlogue | SOURce | KEYWord | ORDEr | CONNection | LOCAtion | IMAGe | BRIDge | CHROnology	**(default: none)**
TYPE=Temporary | Permanent	**(default: Temporary)**
SOURce=<name of database>	**(default: the database currently being processed)**
WRITe=Names | Parts | Structure | Generic | Text | Mapping | Device | Volume | Image	**(default: none)**
CATAlogue=<name of catalogue>	**(default: the catalogue currently being processed)**

It is used to display logical objects from the local and/or permanent environment.

If you invoke it without a parameter, it will instruct the system to display all the logical objects in the local environment.

7.3.3.1 The `name=` parameter

This parameter expects you to specify the name of a logical object or a database. The command will then display the logical object or the database's structure.

If you do not specify this parameter all definitions of the class specified in `usage=` are displayed instead.

7.3.3.2 The `usage=` parameter

This parameter expects a parameter value consisting of one of the keywords `text`, `number`, `date`, `category`, `relation`, `conversion`, `soundex`, `skeleton`, `guth`, `substitution`, `codebook`, `catalogue`, `keyword`, `source`, `order`, `connection`, `location`, `image`, `bridge` or `chronology`, all of which can be shortened to their respective first four characters.

It specifies which class of a given logical object should be displayed. You can use this parameter to display up to four classes of logical objects which cannot be introduced by the `item` command, or not by the `item` command alone:

- `codebook` displays a codebook, regardless of whether it was generated by `item` or `create`.
- `catalogue` displays a catalogue generated by the `catalogue` command. If you want to do this, you should also specify `type=permanent`.
- `keyword` displays a list of references stored in a catalogue's reference library. If you have not yet used the `catalogue[]` built-in function to allocate a catalogue, you must explicitly specify a catalogue in the `catalogue=` parameter.

  The list of references is reproduced in the syntax of the `form=` parameter of the `catalogue` definition's `keyword` directive (cf. section 7.3.1.15.1.2). If you redirect the output from this command to a file it can be used as a quick and efficient way of restoring earlier lists of references should you decide to recreate a catalogue and/or a database.

- `source` displays a database's structure. If you want to do this you should also specify `type=permanent`.
- `bridge` instructs the system to print out a list of names of all databases to which the database specified in `source=` is linked.

This parameter accepts a series of additional keywords, but they will cause the system to display the following error message: `System feature not yet implemented: item`.

7.3.3.3 The `type=` parameter

The `type=` parameter expects you to specify one of the keywords `permanent` or `temporary`. Both of them can be shortened to their respective first character.

- `temporary` instructs the system to display logical objects from the local environment only, i.e. instructs the system to behave as it would by default.
- `permanent` instructs the system to load the logical object from the database's permanent environment before displaying it. If another object of the same name and class already exists in the local environment, it remains unchanged and is not replaced by the new logical object. If you have not yet addressed a particular database using `query`, you should specify the database's name using `source=`.

7.3.3.4 The `source=` parameter

The `source=` parameter expects the name of a database as a parameter value. If you also specify `type=permanent`, the system loads the designated logical objects from this database's permanent environment.

Once the system has processed this command, the linkage with this database is broken again: so anything you allocated in a previous `query` command remains unchanged.

7.3.3.5 The `write=` parameter

The `write=` parameter accepts the keywords `names`, `parts`, `structure`, `generic`, `text` and `mapping` as values, all of which can be shortened to their respective first character.

It is only legal if you use it with certain of the `usage=` parameter's values. This means that the system interprets it as follows:

- `usage=source`

  l Here `write=` defines the amount of information about the database to be displayed. You can specify this parameter as often as you like, using a different keyword each time in order to obtain the printout you require.

  - `names` instructs the system to print out a list of descriptions of all valid element names in the database,
  - `parts` instructs the system to print out a list of descriptions of all valid group names in the database,
  - `structure` instructs the system to print out a description of the structural relationships between all non-generic identifiers in the database, specifying any structural restrictions in force (e.g. minimum/maximum permissible frequencies) and

   - `generic` instructs the system to print out a list of descriptions of all names valid as generic names (for elements *and* groups) in the database.
- `usage=location`
  Here `write=` defines the way in which the contents of a `location` definition should be reproduced.
   - `text` describes the prevailing settings for each individual topographical object.
   - `mapping` instructs the system to produce a map, containing all the topographical objects defined in the `location` definition, together with their standard attributes.
- `usage=image`
  Here `write=` defines the way in which the various types of items within a `image` definition are listed.
- `image` instructs the system to give a complete list of all the images described by the definition. If no other `write` parameter has been specified all the images are listed in one list. If any of the following parameter values are specified also, the images are listed under the device and/or the volume on which the reside.
- `volume` instructs the system to give a complete list of all the volumes in the definition. If `write=device` is not specified all volumes are described in one list, otherwise the volumes are listed under the devices on which the can be mounted.
- `device` instructs the system to give a complete list of all the devices in the definition.

7.3.3.6 The `catalogue=` parameter

The `catalogue=` parameter expects the name of a catalogue as a parameter value. If you specify the parameter `usage=keyword`, the system loads the indicated reference list from the catalogue's reference library.

Once the system has processed this command, it breaks the link with this catalogue; so anything you may have allocated previously to another catalogue remains unchanged.

7.3.4 Defining environment-related default settings

Every time the system needs to perform an operation which requires it to convert input characters into one of $\kappa\lambda\epsilon\iota\omega$'s own data types the system checks to see whether the user has explicitly stated which logical object it should use to perform this conversion. If the user has not done so the system checks to see whether the user referred to an element to which a logical object has been assigned, so that it can use the latter instead. Only if neither of these two situations applies does the system activate the default settings associated with this particular data type (cf. sections 7.3.1.2 and 7.3.1.6). These are stored in logical objects with the same names as the data type in question (text, date etc.): the user can delete or redefine them just like any other logical object.

Thanks to the commands listed below, the user has the option of

- ensuring that other logical objects are invoked by default and
- controlling the way the system behaves in general.

All the commands listed below are also directives in a database's structure definition, so you can also issue them between a `database` command and its associated `exit` command.

7.3.4.0 The durability of logical objects

All logical objects which control the system's default behaviour are integrated into a given database's logical environment in a particularly lasting manner. These objects include:

Name of object	class of object	name modifiable by the command
Text	Text	Text
Date	Date	Date
Date	Number	—
Number	Number	Number
Category	Category	Category
Relation	Relation	Relation
Location	Location	Location
Image	Image	Image
Order	Order	—

Without these logical objects, you cannot work with the database. This has a number of consequences, which will not be obvious to the occasional user without some additional explanation.

a) If you delete one of these logical objects, but do not define a new object with the same name, you will be unable to use the database until you have done so. This means that if you specify the database's name in the `name=` parameter of the `query` or `read` commands, you will be confronted by an error message. Before you can start to use your database again, you will have to reintegrate the missing logical object into the database's permanent logical environment, using an appropriate `item` definition.

b) If you use `query` to address a database, the system starts by restoring the logical objects listed above in exactly the same condition they were in when you last used the database.

Thus in the following κλειω task:

```
item name=order;usage=order;overwrite=yes
sign sign=ä;after=a
sign sign=ö;after=o
sign sign=ü;after=u
exit name=order
query name=example;part=person
index part=:name
stop
```

the system will appear to behave incorrectly: it will continue to sort all the characters together, making no distinction between characters with and without umlauts. *This happens because when the system processes the* `query` *command, the first thing it does is re-establish the sorting order which existed when you first created the* `source` *database.*

To avoid this problem, you can either place an `order` definition in front of the `database` command for the database in question, or retrospectively arrange for a permanent change to be stored in the database. The following variation would produce the required result:

```
item name=order;usage=order;overwrite=yes;source=example;type=permanent
sign sign=ä;after=a
sign sign=ö;after=o
sign sign=ü;after=u
exit name=order
query name=example;part=person
index part=:name
stop
```

If you define the required sorting order as shown above, it will be preserved even after you exit from the κλειω program in question. The system will reuse it every time it analyses this database, until further notice.

More or less the same thing applies to all the other logical objects named above.

c) A similar problem arises when you instruct the system to perform the following task:

```
item name=order;usage=order;overwrite=yes
sign sign=ä;after=a
sign sign=ö;after=o
sign sign=ü;after=u
exit name=order
database name=example;overwrite=yes;...
```

The system also appears to behave incorrectly in this case: it continues to sort characters with umlauts in the newly created database in the last sorting order you defined for `source`. This is because the system restores "privileged" logical objects after deleting an existing database to make it easier for new databases to "inherit" the user's preferred default settings.

If you want to avoid this, we would advise you explicitly to link the relevant logical object to the newly created database using `type=permanent` and the `source=` parameter, once the system has executed the structure definition.

As we mentioned, this procedure applies to all the logical objects listed above, not just to sort sequences.

7.3.4.1 The `text` command

The general form of the `text` command is:

NAME=<name of object>   (default: none)

You use it to modify the system's default behaviour when processing data of the `text` data type.

By default, the system assumes that:

- unless you have made other arrangements it should use the logical object `text` to perform the conversion.

7.3.4.1.1 The `name=` parameter

This parameter expects a parameter value consisting of the name of a `text`-class logical object. From then on the system uses this object to process `text` data, unless you explicitly instruct it to use another object.

7.3.4.2 The `date` command

The general form of the `date` command is:

NAME=<name of object>	**(default: none)**
TEXT=No | Yes	**(default: No)**
MINImum=<calendar constant>	**(default: system-dependent)**
MAXImum=<calendar constant>	**(default: system-dependent)**
FIRSt=<calendar constant>	**(default: system-dependent)**

You use it to change the system's default behaviour when processing data of the `date` data type.

By default, the system assumes that:

- unless you have made other arrangements it should use the logical object `date` to perform the conversion.
- calendar dates are stored in a form which is independent of the input format; the system *cannot* reproduce the original date as it was entered, letter for letter.
- All dates prior to 1/1/1500 will be rejected as too early.*
- All dates after 1/1/1994 will be rejected as too late.*
- The setting which specifies when the Gregorian calendar was introduced is set to 15/10/1582.* The system treats all dates prior to this date as dates in the Julian calendar; it treats all other dates – including 15/10/1582 – as dates in the Gregorian calendar.

7.3.4.2.1 The `name=` parameter

This parameter expects as a parameter value the name of a logical object of the `date` class. Henceforth, the system will use it to process `date` data, unless you explicitly instruct it to use another object of this type.

7.3.4.2.2 The `text=` parameter

This parameter expects the keywords `yes` or `no` as parameter values. Both of them can be shortened to their respective first character.

- `no` instructs the system to store the entered data in its own format as well, i.e. instructs the system to behave as it would by default.
- `yes` instructs the system to store the input exactly as it is entered and to reproduce it in this format should the user require the system to display the entered data in visual form.

7.3.4.2.3 The `minimum=` parameter

This parameter expects as a value a calendar date in a notational style which the system can transform using the existing set of standard rules.

This date becomes the earliest valid date in the database.

---

* These defaults can be changed with the configuration program.

7.3.4.2.4 The `maximum=` parameter

This parameter expects as a value a calendar date in a notational style which the system can transform using the existing set of standard rules.

This date becomes the latest valid date in the database.

7.3.4.2.5 The `first=` parameter

This parameter expects as a value a calendar date in a style of notation which the system can transform using the existing set of standard rules.

Henceforth, the system treats this as the date on which the Gregorian calendar was introduced.

7.3.4.3 The `number` command

The general form of the `number` command is:

NAME=<name of object>	**(default: none)**
TEXT=No | Yes	**(default: No)**
MINImum=<number>	**(default: system-dependent)**
MAXImum=<number>	**(default: system-dependent)**

You use it to change the system's default behaviour when processing data of the `number` data type.

By default, the system assumes that:

- Unless you have made other arrangements it should use the logical object `number` to make the conversion.
- Numerical data should be stored in the system's internal format, which may be quite different from the original input. The system *cannot* reproduce the input data as it originally appeared.
- Numbers which are less than the minimum specified in your installation are rejected as incorrect; in the system as delivered, this minimum value is set to 0.0. Because negative numbers are not a common feature of historical documents, the system is only capable of processing them if it is explicitly empowered to do so.
- Numbers which are greater than the maximum specified in your installation are rejected as incorrect; in the system as delivered, this maximum limit is set to 1000000.0

7.3.4.3.1 The `name=` parameter

This parameter expects as a parameter value the name of a logical object in the `number` class. Henceforth, the system uses this object to process `number` data, unless you explicitly instruct it to use another object of this type.

7.3.4.3.2 The `text=` parameter

This parameter expects the keywords `yes` or `no` as values. Both of these can be abbreviated to their respective first character.

- `no` instructs the system to store input data in its own format only; i.e. to behave as it would by default.
- `yes` instructs the system to store input data in its original format as well, and to use it in this format whenever you wish to display the input data in visual form.

Cf. section 7.3.1.4.1.7 on how to differentiate in this respect between individual `number` definitions.

7.3.4.3.3 The `minimum=` parameter

This parameter expects as a value a number in a format which the system can convert using the existing set of standard rules.

This number becomes the lowest valid number in the database.

7.3.4.3.4 The `maximum=` parameter

This parameter expects as a value a number in a format which the system can convert using the existing set of standard rules.

This number becomes the highest valid number in the database.

7.3.4.4 The `category` command

The general form of the `category` command is:

NAME=<name of object> (default: none)

You use it to modify the system's default behaviour when processing data of the `category` data type.

By default, the system assumes that:

- Unless you have made other arrangements it should use the logical object `category` to make the conversion.
- Every character in an entry of this data type may only appear once.
- It should not preserve the order of the characters.

7.3.4.4.1 The `name=` parameter

This parameter expects as a parameter value the name of a `category`-class logical object. Henceforth the system uses this object to process `category` data, unless you explicitly instruct it to use another object of this type.

7.3.4.5 The `relation` command

The general form of the `relation` command is:

NAME=<name of object>	**(default: none)**
TEXT=No | Yes	**(default: No)**
CUMUlate=No | Yes	**(default: No)**
SOURce=<name of database>	**(default: none)**
MINImum=<number>	**(default: 0.95)**
MAXImum=<number>	**(default: 1.00)**
ALSO=<number>	**(default: none)**
WITHout=<number>	**(default: none)**

You use it to modify the system's default behaviour when processing data of the `relation` data type.

By default, the system assumes that:

- Unless you make other arrangements it should use the logical object `relation` to make the conversion.
- It should not attempt to activate subnetworks which have not yet been activated and which are linked to an object of the `relation` type.
- When you specify ranges for visibility and/or view of to limit the amount of data processed, this ranges are valid only for the database which is specified with the `name=` parameter of the `query` command. All databases which are reached as a result of `relation` data being interpreted are accessed with the default visibility and view.

7.3.4.5.1 The `name=` parameter

This parameter expects as a parameter value the name of a `relation`-class logical object. Henceforth, the system uses this object to process `relation` data, unless you explicitly instruct it to use another object of this type.

7.3.4.5.2 The `text=` parameter

This parameter expects one of the keywords `yes` or `no` as a value. Both of these can be shortened to their respective first character.

- `no` instructs the system to store the input data in its own format only; i.e. to behave as it would by default.
- `yes` instructs the system to store the input data in its original format as well, and to use it in this format whenever you wish to display the input data in visual form.

7.3.4.5.3 The `cumulate=` parameter

This parameter expects the keywords `yes` or `no` as values. Both of these can be shortened to their respective first character.

- `no` instructs the system not to include any more network identifiers from the network specified in `name=` in the database; i.e. to behave as it would by default.
- `yes` instructs the system to incorporate all identifiers still available in this network into the database which is currently being processed, before performing any other action.

Note: *The system executes this command immediately. In the case of complex networks, this activation may last for a considerable length of time.*

Once the system has executed this command, it deletes both files with the name of the network in question and any files with the suffixes (known on MS-DOS systems as file extensions) `rni` and `rnd`.

7.3.4.5.4 The `source=` parameter

The `source=` parameter expects the name of a database as a parameter value. If you specify this parameter the system attempts to link the indicated network to this database. This parameter only makes sense if you combine it with the `cumulate=` parameter. Once the system has processed this command, it breaks the linkage with this database: hence anything which you may have allocated in a previous `query` command remains unchanged.

7.3.4.5.5 Choosing visibility and view

As visibility and views can be interpreted according to completely different models, dependent on the purposes of the creator of the database, it will not usually be sensible to keep ranges defined for them when navigating through the databases one is left and another one entered. Therefore $\kappa\lambda\epsilon\iota\omega$ assumes, that the visibility ranges and the selection of views specified with the parameters `minimum=`, `maximum=`, `also=` and `without=` of the `options` command are valid only for the database which is specified by the `query` command. That is, as soon as another database is entered in consequence of a user command, all these settings are ignored.

The four parameters just mentioned can also be specified with the `relation` command. They expect exactly the same kind of values described in section 7.1.4 of this manual for their namesakes. Visibility and view described in this way are applied to all databases which are reached after leaving the database specified by the `query` command.

There are no possibilities so far to define different ranges of visibility and view for individual databases: the definitions of the `options` command apply to the one specified by `query`, the ones of the `relation` command to all other databases accessed.

7.3.4.6 The `location` command

The general form of the `location` command is:

NAME=<name of object> **(default: none)**
USAGe=DOS-screen | POSTscript | COLOrpostscript | DISSpla | AIX-screen **(default: see below)**

You use it to modify the system's default behaviour when processing data of the `location` data type.

By default, the system assumes that:

- Unless you have made other arrangements it should use the logical object `location` to perform the conversion. *In fact, a logical object of this type does exist. But in the current version of κλειω, it is always empty: so if the system does use it, you will not obtain any cartographical printouts.*
- it should attempt to display all cartographical output requested on your computer's default output device.

7.3.4.6.1 The `name=` parameter

This parameter expects as a parameter value the name of a `location`-class logical object. Henceforth, the system will use this object to process `location` data, unless you explicitly instruct it to use another object of this type.

7.3.4.6.2 The `usage=` parameter

This parameter expects a parameter value describing a logical or physical output device of your computer. All these keywords can be shortened to their first four characters only.

This parameter instructs the system henceforth to redirect all cartographical output to the output device specified by the keyword.

- `dos-screen` This keyword directs the output to the screen of your PC operating under the DOS operating system. The screen will contain the map until you press any key on your keyboard. On DOS systems this is the default; on all other systems this option produces an error message.
- `aix-screen` This keyword directs the output to the screen of your RS 6000 operating under the AIX operating system. The screen will contain the map until you press any key on your keyboard. On AIX systems this is the default; on all other systems this option produces an error message.
- `postscript` This keyword directs the output to the name of the output file specified by the `mapping` command. This file can than be sent to any *Postscript* printer. Please note that on many sites a printer which allows the printing of Postscript as well as non-Postscript files requires the usage of a specific command to act as a Postscript device. This is the default on all but DOS and AIX systems. All directives referring to colour will be supressed.
- `colorpostscript` This keyword directs the output to the name of the output file specified by the `mapping` command. This file can than be sent to any Colour Postscript printer.
- `disspla` This keyword directs the output to the name of the output file specified by the `mapping` command. This file contains plotting instructions for the DISSPLA

plotting package. To convert its content into a plotted map, you have to access a special module converting it into a plotfile according to the local conventions. At the time of printing such a program exists only for the VM/CMS operating system. A version for UNIX may become available; as this still requires that your site has a DISSPLA license, the use of this option is discouraged.

7.3.4.7 The `image` command

The general form of the `image` command is:

NAME=<name of object> **(default: none)**
USAGe=No | Yes **(default: No)**

You use this command to modify the system's default behaviour when processing data of the `image` data type.

By default, the system assumes that:

- Unless you have made other arrangements it should use the logical object `image` to make the conversion. *This logical object does exist. In the current version, however, it is always empty: if the system uses it, it will not be able to process any images.*
- It should attempt to output all graphical representations to the default graphics output device attached to your computer.

7.3.4.7.1 The `name=` parameter

This parameter expects as a parameter value the name of an `image`-class logical object. Henceforth, the system uses this object to process `image` data, unless you explicitly instruct it to use another object of this type.

7.3.4.7.2 The `usage=` parameter

This parameter expects as a parameter value one of a series of keywords specific to your computer: all keywords can be abbreviated to their first four characters.

This parameter instructs the system henceforth to redirect all graphical output generated by image processing operations to the device specified by the keyword.

At the time of printing of this manual, this parameter is not supported. All images are always displayed within the windows of the Image Analysis System of $\kappa\lambda\epsilon\iota\omega$, which is automatically started on the current terminal. If that terminal is non-graphical or has less than 8 planes (is not able to display at least 256 grey levels or shades of colour), the session is terminated with an error message.