7. The environment of a $\kappa\lambda\epsilon\iota\omega$ program

When you activate $\kappa\lambda\epsilon\iota\omega$ a series of settings come into force. Some of these settings depend on your particular installation. However, most of these settings can be changed while you are working with $\kappa\lambda\epsilon\iota\omega$. Whatever you can change once you can change as often as you like.

The `options` command is used to make these changes.

7.1 The `options` command

The general form of the `options` command is:

EXPLain=Yes | No	**(default: No)**
USAGe=Production | Training	**(default: Production)**
SOURce=<filename on your computer> | Stop	**(default: none)**
SEQUence=Numbers | Letters | Roman	**(default: Numbers)**
FIRST=<sequence identifier>	**(default: none)**
LIMIt=<sequence identifier>	**(default: none)**
PART=<filetype on your computer>	**(default: none)**
NAME=Survey | Detailed	**(default: Survey)**
TARGet=<filename on your computer> | Stop	**(default: none)**
SUFFix=<filetype on your computer>	**(default: none)**
OVERwrite=No | Yes	**(default: No)**
LINEs=<number> (0 to 4)	**(default: 1)**
SIGNs=<number>	**(default: installation-dependent)**
RESUlt=No | Yes	**(default: No)**
MINImum=<number>	**(default: 0.95)**
MAXImum=<number>	**(default: 1.00)**
ALSO=<number>	**(default: none)**
WITHout=<number>	**(default: none)**
IDENtification=Yes | No	**(default: Yes)**
ALWAys=<number>	**(default: 0)**
ONLY=<number>	**(default: 0)**
CONTinue=Yes | No	**(default: Yes)**
MORE=No | Yes	**(default: No)**
SECOnd=Yes | No	**(default: Yes)**
CUMUlate=No | Yes	**(default: No)**
SELF=Yes | No	**(default: Yes)**
POSItion=Yes | No	**(default: Yes)**
STARt=Yes | No	**(default: Yes)**
IMAGe=Yes | No	**(default: Yes)**
BRIDge=Yes | No	**(default: Yes)**
RELAtion=Yes | No	**(default: Yes)**

This command defines the general conditions in which a $\kappa\lambda\epsilon\iota\omega$ program is executed.

7.1.1 General system behaviour

$\kappa\lambda\epsilon\iota\omega$'s behaviour in interactive mode, as well as certain other aspects of the system's behaviour, depend on the way you invoke the program. You can modify these modes of behaviour.

7.1.1.1 The `type=` parameter

This parameter, used in earlier versions for control of dialogue behaviour, has been removed. It is currently undefined.

7.1.1.2 The `explain=` parameter

The `explain=` parameter accepts the keywords `yes` or `no` as values. Both of them can be abbreviated to their respective first character.

- `yes` causes the system to issue explanations for its individual error messages (but only when the system is operating in interactive mode),
- `no` suppresses these explanations.

7.1.1.3 The `usage=` parameter

The `usage=` parameter accepts the keywords `production` and `training` as values. Both of them can be abbreviated to their respective first character.

- `production` instructs the system to execute every correctly formulated task immediately (this is how the system behaves by default),
- `training` instructs the system to check tasks for errors, but not to execute them. (However it does *not* affect the execution of definitions.)

7.1.1.4 The `form=` parameter

This parameter, used in earlier versions for debugging purposes, has been removed. It is currently undefined.

7.1.2 Managing input

By default

- $\kappa\lambda\epsilon\iota\omega$ reads input data from a standard input medium and
- ignores the distinction between upper-case and lower-case characters when processing data names.

7.1.2.1 Managing data names

By default $\kappa\lambda\epsilon\iota\omega$ converts all the names it manages into lower-case characters before processing them. This is mandatory where definition names are involved; in the case of names of groups or elements you may cancel this feature.

7.1.2.1.1 The `name=` parameter

The `name=` parameter accepts the keywords `survey` or `detailed` as values. Both of them can be abbreviated to their respective first character.

- `survey` instructs the system to convert all names under its control to lower-case before processing them,
- `detailed` prevents this from happening.

7.1.2.2 Rerouting input

You can instruct $\kappa\lambda\epsilon\iota\omega$ to read from any file or group of files on the computer after the end of an `options` command. This capability can be nested to any depth: files to which input has been redirected by an `options` command may themselves contain commands redirecting the input to other files.

However, $\kappa\lambda\epsilon\iota\omega$ will not redirect input in a circle. This means that the system will reject a command in file "b" instructing it to read in file "a", after it has been directed to file "b" from file "a" in the first place regardless of any other files which may be addressed on the way by either "a" or "b".

7.1.2.2.1 The `source=` parameter

The `source=` parameter accepts as a value any constant corresponding to the name – or part of a name as defined more precisely by one of the following parameters – of a file which has been correctly named according to your operating system's filename conventions, or the keyword `stop` which can be abbreviated to its first character.

- If you specify a filename, the system interrupts its reading of the current input file and continues reading in the named file. If the computer considers the filename to be invalid, the system will not switch over to the second file. The filename can be as long as you like (up to a total of 4095 characters in $\kappa\lambda\epsilon\iota\omega$ as delivered). Hence you can define the longest of paths on systems running MS-DOS, UNIX, VMS and other similar operating systems. However, we would advise you to define *absolute* paths because future versions of $\kappa\lambda\epsilon\iota\omega$ will manage their own directories on these systems. Once the system reaches the end of the file to which it was redirected it returns to the original file at the point at which it encountered the `options` command and continues to import the data from this file. We are unable to predict what the consequences would be if you edited or deleted this file in the meantime.
- If you specify `stop` the system stops processing the input file which is currently in use and resumes processing input data at the point in a previous input file at which it last encountered an `options` command for redirecting input.

If you specify `stop`, but the system has not been redirected to any other file than the default input file, it ignores this parameter.

7.1.2.2.2 The `first=` parameter

The `first=` parameter expects a value which can be converted into a number, following the same rules as those described for the `sequence=` parameter.

It instructs the system to supplement the filename specified in the `source=` parameter by a count beginning with the parameter's numerical value.

If you specify this parameter you must also specify a `limit=` parameter.

$\kappa\lambda\epsilon\iota\omega$ attempts to read in sequential order from all files with names beginning with the `source=` parameter's value and supplemented by the `first=` parameter's numerical value, and increments the last-named number until it equals the value of the `limit=` parameter.

7.1.2.2.3 The `limit=` parameter

The `limit=` parameter expects a value which can be converted into a number following the rules described for the `sequence=` parameter.

It only makes sense if it is accompanied by the `first=` parameter, which describes how it should be interpreted.

7.1.2.2.4 The `sequence=` parameter

The `sequence=` parameter expects one of the keywords `number`, `letters` or `roman` as a value. All of them can be abbreviated to their respective first character.

- `number` instructs the system to interpret the data in the `first=` and `limit=` parameters as whole Arabic numerals.
- `letters` instructs the system to interpret the data in the `first=` and `limit=` parameters as an algebraic numbering system. An algebraic number consists of a sequence of alphabetic characters (no distinction being made between upper-case and lower-case letters), which the system interprets as a number in a number system on base 27, where "a" represents the number "1", "z" represents the number "26" and zero cannot be expressed as a value. (So "z" is immediately followed by "aa".)
- `roman` instructs the system to interpret the data in the `first=` and `limit=` parameters as Roman numerals. These have to satisfy the following conditions:
  - They consist solely of the "digits" i, v, x, l, c, d and m. (upper-case input is acceptable).
  - They do not include any spaces.
  - Not more than one of the digits i, x or c can be positioned in front of a digit with a higher value.
  - Only m can be used more than four times.
  - A number cannot contain more than one hundred digits.

So that you can control `first=` and `limit=` properly, you must define the `sequence=` parameter *in front of* these two parameters.

7.1.2.2.5 The `part=` parameter

The `part=` parameter expects as a value a constant representing the *type* of input files which should be used. The way the system interprets this parameter depends on the logic underlying your computer's file management: on MS-DOS systems, the filetype is indicated by an extension consisting of a full stop (period) followed by three characters, appended to a filename constructed from the values of the `source=` and `first=` parameters.

7.1.3 Managing output

By default, κλειω writes the resulting output to the output medium defined as the default output medium for the particular computer in question. Output is produced as a series of lines, the length of which is defined by your particular installation.

7.1.3.1 Describing the output format

By default, κλειω prints

- lines of a length determined by your installation and
- inserts one blank line between each pair of lines if the two lines in question are not so closely joined that one is regarded as the continuation of the other.

7.1.3.1.1 The `lines=` parameter

The `lines=` parameter expects a value consisting of an integer between zero and four. This number specifies how many blank lines should be inserted between two consecutive lines during while κλειω is printing out.

7.1.3.1.2 The `signs=` parameter

The `signs=` parameter expects a value consisting of a positive whole number. This value tells κλειω how many characters to print on each line.

7.1.3.1.3 The `result=` parameter

The `result=` parameter accepts the keywords `no` or `yes` as values. Both of them can be abbreviated to their respective first character.

- `no` causes the system to indicate the start of a new 'page' to be indicated in the output file by an empty line. This is the default behaviour of the system.
- `yes` results in page breaks being represented by form feed characters. (Which lead to page ejects on a printer.)

7.1.3.2 Rerouting output

Output can be redirected to any number of files in succession. Files can be cascaded, i.e. parts of the output can be redirected from one output file to other output files, whereby you have the option of continuing to output to a previous file once you have finished directing output to later files in the series.

7.1.3.2.1 The `target=` parameter

The `target=` parameter accepts as a value a constant corresponding to the name – or part of the name, as defined more precisely by the following parameter – of a file which has been correctly named according to your operating system's filename conventions, or else the keyword `stop`, which can be abbreviated to its first character.

- If you specify a filename the system interrupts its output to the current output file and redirects it to the file you specified. If the computer considers the filename to be invalid the system does not redirect its output. The filename can be as long as you like (up to a total of 4095 characters in κλειω as delivered). Hence you can define the longest of paths on systems running MS-DOS, UNIX, VMS and similar operating systems. However, we would advise you to define *absolute* paths, because future versions of κλειω will be able to manage their own directories on these systems.
- If you specify `stop`, the system stops processing the output file which is currently in use and resumes its output at the point in a previous output file at which it last encountered an `options` command for redirecting output.

If you specify `stop`, but had not redirected the system to any other file than the default output file, it ignores this parameter.

7.1.3.2.2 The `suffix=` parameter

The `suffix=` parameter expects as a value a constant representing the *type* of output files which should be used.

The way the system interprets this parameter depends on the logic underlying your computer's file management system: on MS-DOS systems, the filetype is indicated by an extension consisting of a full stop (period) followed by three characters, appended to a filename derived from the value of the `target=` parameter. The full stop is automatically supplied by the system; so the parameter should read `suffix="lis"`, *not* `suffix=".lis"`.

7.1.3.2.3 The `overwrite=` parameter

The `overwrite=` parameter expects the keywords `yes` or `no` as values. Both of them can be abbreviated to their respective first character.

- `no` means that the system does not reroute its output if the output file you specified already exists (this is the default setting).
- `yes` means that the system overwrites an existing file with the same name as the output file you specified.

7.1.4 Choosing visibility and view

When $\kappa\lambda\epsilon\iota\omega$ is processing a database it checks every element and group it encounters to see whether it satisfies general conditions regarding visibility and view before accepting it for further processing. By default the system specifies the following criteria:

- Every item of information to be processed must have a visibility of at least 0.95. (This corresponds to the special visibility code "!".)
- No item of information intended for processing should have a visibility of more than 1.00. (In view of the fact that such a level of visibility is impossible, this condition is not particularly restrictive.)
- Every item of information to be processed must exist in the database's main view. (This means it is not enough for the information item to exist in a view with an explicit level code.)
- It makes no difference whether or not an item of information exists in other views in addition to the main view.

7.1.4.1 The `minimum=` parameter

The `minimum=` parameter expects a value consisting of

- a question mark, which has a value of 0.5
- an exclamation mark, which has a value of 0.95,
- a number between zero and one, which is accepted as it stands, or
- a number greater than one, which is repeatedly divided by one hundred until the resulting value lies between zero and one.

Until you change this value again, the system accepts all items of information with a visibility greater than or equal to the value of `minimum=` for processing.

If as the result of an `options` command the value assigned to `minimum=` value becomes greater than the value assigned to `maximum=` both values are reset to the appropriate system defaults.

7.1.4.2 The `maximum=` parameter

The `maximum=` parameter expects a value consisting of

- a question mark, which has a value of 0.5,
- an exclamation mark, which has a value of 0.95,
- a number between zero and one, which is accepted as it stands, or
- a number greater than one, which is repeatedly divided by one hundred until the result lies between zero and one.

Until you change this value again, the system accepts all items of information with a visibility less than or equal to the value of `maximum=` for processing.

If as a result of an `options` command, the value assigned to `minimum=` value becomes greater than the value assigned to `maximum=`, both values are reset to the appropriate system defaults.

7.1.4.3 The `also=` parameter

The `also=` parameter expects a value consisting of a whole number between one and the highest alternate level code valid on your computer. (Typical code values include 8, 16, 32.)

This parameter instructs $\kappa\lambda\epsilon\iota\omega$ to process all information items existing in the alternate view with the level code you specified until you cancel the instruction again.

7.1.4.4 The `without=` parameter

The `without=` parameter expects a value consisting of a whole number between one and the highest alternate level code valid on your computer. (Typical code values include 8, 16, 32.)

This parameter instructs $\kappa\lambda\epsilon\iota\omega$ not to process any information items existing in the alternate view with the level code you specified until further notice.

7.1.5 Identifying selected data

Whenever the system displays selected information for the user's attention the information is accompanied by an identification line in which the system attempts to show which data in the database the information was derived from.

By default this identification line contains details of the document from which the displayed information was extracted. It consists of the document's group identifier, the document's own identifier and its ordinal number within the database, these last two details being enclosed in a pair of parentheses (round brackets). This is followed by a colon, whereupon the same details are displayed for the group from which the elements displayed were taken. Thus for example:

```
house (14 = "123-c") :  son (3 = "son-3")
```

7.1.5.1 The `identification=` parameter

The `identification=` parameter accepts one of the keywords `yes` or `no` as a value. Both can be abbreviated to their respective first character.

- `yes` instructs the system to display the standard identification line, as described above. This is the system's default setting.
- `no` suppresses this output.

7.1.5.2 The `only=` parameter

The `only=` parameter expects a positive whole number as a value. This is interpreted as the maximum logical depth to which group identifiers should be displayed as constituents of the identification line. If you set this parameter to zero, the system displays the default identification line; if you set it to a higher value, the standard identifier is supplemented by as many identifying expressions as are necessary to express the logical distance from the "topmost" (document) level which corresponds to the numerical value you assigned to the `only=` parameter. So if we return to the example above, we see that a value of "5" might produce the following identification line:

```
house (14 = "123-c") flat (3 = "fla-3") family
    (1 = "fam-1"):  son (3 = "son-3")
```

7.1.5.3 The `always=` parameter

The `always=` parameter expects a positive whole number as a value.

The system interprets this as the logical distance from the document of the group identifier which should always be displayed as part of the identification line. If you set this parameter to zero, the system displays the default identification line; if you set it to a higher value, the document identifier is replaced by the identifier of the group located at a remove from the document corresponding to a number of levels equal to the number specified in the `always=` parameter. If the number of logical levels in the data is less than the number you specified, the highest available level is used. So in our introductory example, a value of "2" assigned to `always=` might result in the following identification line:

```
family (1 = "fam-1") :  son (3 = "son-3")
```

7.1.6 Defining the standard printout

The `options` command also allows you to define a series of parameters for controlling the scope of the printed output produced by all subsequent `write` commands, until you next change the parameters. The effects of these separate parameters on the scope of default printouts are discussed in detail in our description of the `write` command in section 8.3.1.2. These parameters include `continue=`, `more=`, `second=`, `cumulate=`, `self=`, `position=`, `start=`, `image=` and `bridge=`.

7.1.6.1 The `relation=` parameter

The `relation=` parameter accepts one of the keywords `yes` or `no` as a value. Both can be abbreviated to their respective first character.

- `yes` instructs the system to indicate in the output when the system uses an entry of the data type `relation` to navigate through the database. This is the system's default setting.
- `no` suppresses this output.

7.1.7 Examples

Below you will find some examples of correctly written `options` commands:

```
options usage=training;explain=no
```

```
options target="report";suffix="lis"
```

7.2 Comments: the `note` command and directive

One of the few commonplaces in data processing which applies as much to programming in assembler as to the programming languages used in Artificial Intelligence, is the saying that it is always worth taking the time and trouble to insert comments in the programs you write. $\kappa\lambda\epsilon\iota\omega$ provides you with the `note` command for precisely this purpose. `note` accepts any text as its specification. So that it is easier to use, the system also accepts this command as a valid directive for every kind of definition. So wherever the beginning of a new line is regarded as the legitimate beginning of a new command or directive you can also use the `note` command.

An example of a valid `note` command follows:

```
note This is a line of commentary
```

7.3 The logical environment

As soon as you activate $\kappa\lambda\epsilon\iota\omega$ a *logical environment* is created. This consists of *logical objects* which are quite independent of one another. Each logical object instructs the system how to interpret a particular kind of data. If you set up a new database, the logical environment existing at the time you set up the database is stored in the database – in part implicitly, in part by being explicitly and permanently associated with it.

Logical objects from your $\kappa\lambda\epsilon\iota\omega$ program's environment which are saved in the database in this way form the database's *permanent logical environment*; any logical objects known to a $\kappa\lambda\epsilon\iota\omega$ program at that time or defined by the user on an *ad hoc* basis or loaded from any number of databases, form the program's *local logical environment*.

You can modify both local and permanent logical environments at any time, by redefining, modifying or deleting as many logical objects as you like.

Logical objects come into being either by being declared in an `item` command or by being implicitly generated by certain commands described in sections 8.3.2 (`catalogue`) and 8.3.4 (`create`). You can modify them by means of definitions in an `item` command, delete them using a `delete` command, or make them visible by using a `describe` command.

Using the `text`, `date`, `number`, `category`, `relation`, `location` and `image` commands, you can also control which object is brought in from the logical environment as the default in given situations, and how the system uses it.

7.3.1 Defining logical objects

Logical objects are introduced by definitions, which are opened by the `item` command. As they are definitions, they are generally written as follows:

`item` command
|
Directives specific to this type of definition:
|
`exit` directive

The `item` command and the `exit` directive *must* both have a `name=` parameter, to which the same name is assigned in both cases.

You can address all classes of logical object using the command language before introducing them to the system via the appropriate definition. However, the system must be aware of them if you want the command referring to them to be executed. If $\kappa\lambda\epsilon\iota\omega$ encounters a situation in which a logical object is unknown, it starts by examining the permanent environment of the database which has just been processed in order to see whether an object of this name and class exists there. Only if this is not the case will $\kappa\lambda\epsilon\iota\omega$ refuse to execute the task in question.

7.3.1.1 The `item` command

The general form of the `item` command is:

NAME=<name_of_definition>	**(MUST be specified!)**
USAGe=TEXT | DATE | NUMBer | CATEgory | RELAtion | CONVersion | SOUNdex | SKELeton | GUTH | SUBStitution | CODEbook | LOCAtion | ORDEr | CATAlogue | CONNection | IMAGe | CHROnology	**(MUST be specified!)**
TYPE=Permanent | Temporary	**(default: Temporary)**
OVERwrite=No | Yes	**(default: No)**
REPEat=Yes | No	**(default: existing status)**
SOURce=<name_of_database>	**(default: none)**

It is used to redefine and modify logical objects in local and permanent logical environments.

7.3.1.1.1 The `name=` parameter

The `name=` parameter expects a valid user-defined name as a value. This name is used to enter the newly defined logical object into the local and/or permanent environment or prepare it for modification. You *must* define this parameter.

7.3.1.1.2 The `usage=` parameter

This parameter expects a value consisting of one of the keywords `text`, `number`, `date`, `category`, `relation`, `conversion`, `soundex`, `skeleton`, `guth`, `substitution`, `codebook`, `location`, `order`, `catalogue`, `connection`, `image` or `chronology`, all of which can be abbreviated to their four first characters. The parameter determines which kind of logical object is to be defined or modified. You *must* define this parameter.

This parameter also accepts a number of other keywords, which may cause the system to display the following error message: `system feature not yet implemented: item`.

7.3.1.1.3 The `type=` parameter

This parameter expects one of the keywords `permanent` or `temporary` as a value. Both of them can be abbreviated to their respective first character. You can use this parameter to specify whether the logical object should be modified or included only in the local or also in the permanent environment.

- `temporary` tells the system that the relevant definition only refers to the local environment, i.e. explicitly specifies the system's default setting.
- `permanent` instructs the system to include the newly defined or modified logical object as a part of the permanent environment of the database which is just being processed, or of the database named in the `source=` parameter.

7.3.1.1.4 The `overwrite=` parameter

This parameter expects one of the keywords `yes` or `no` as a value. Both of them can be abbreviated to their respective first character. This parameter determines whether the system may overwrite a definition of the type specified in `usage=` which already exists in the permanent environment of the database which the system is currently processing, or whether attempting to do so will result in an error message.

- `no` forbids the system to destroy an already existing logical object of the same name and class, thus explicitly instructs the system to behave as it would by default.
- `yes` allows the system to destroy an object which already exists in the permanent environment.

This parameter only refers to the permanent environment; if a logical object with the name you specified already exists in the local environment, you must explicitly instruct the system to destroy it using the `delete` *command before you go on to redefine the object (cf. section 7.3.2).*

7.3.1.1.5 The `repeat=` parameter

This parameter expects one of the keywords `yes` or `no` as a value. Both keywords can be abbreviated to their respective first character. The parameter tells the system whether to reproduce the definition's directives on the default output medium. By default the system bases its behaviour in this respect on the condition prevailing when it encountered the `item` command.

- `yes` instructs the system to output the imported directives,
- `no` suppresses them.

7.3.1.1.6 The `source=` parameter

This parameter expects as a value the name of a database which the system can access.

If you defined `type=permanent`, $\kappa\lambda\epsilon\iota\omega$ refers by default to the permanent environment of the database you have just been processing, i.e. the database last addressed by `query`. The `source=` parameter instructs the system to use the database with the name you specify instead. Once it has finished processing the `item` definition the previous condition is restored: hence the "database which is currently being processed" does *not* change.

7.3.1.2 The `text` definition

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

signs
exit (MUST be the last directive specified)

Definitions of this kind determine how the system behaves when it is dealing with continuous text, that is, how it converts such texts into `text` data in a database or processes the constants which it is supposed to compare with these elements.

By default data of this data type is converted according to the following rules:

- entries are separated by data signal character 8,
- the data signal character itself is not stored with the data in the database,
- the flags ("?" and "!") are incorporated into the particular entry's core information,
- the system attempts to transform the input into "elegant" text.

You can explicitly instruct the system to behave in this way by arranging for the logical object "`text`", of the `text` type, to be brought in for processing.

By using an explicitly defined logical object, you can also modify this behaviour as follows.

7.3.1.2.1 The `signs` directive

The general form of the `signs` directive is:

PART=<constant>	**(default: data signal character 8)**
WITHout=Yes | No	**(default: Yes)**
SIGNs=Yes | No	**(default: Yes)**
FORM=Yes | No	**(default: Yes)**
ALSO=Yes | No	**(default: Yes)**
NULL=No | Yes	**(default: No)**

It allows you to define how the system should treat different classes of characters.

7.3.1.2.1.1 The `part=` parameter

The `part=` parameter accepts as a value a constant consisting of between one and twelve characters. This constant is used as a democratic separator between individual entries.

7.3.1.2.1.2 The `without=` parameter

The `without=` parameter accepts the keywords `yes` or `no` as values; both of them can be abbreviated to their respective first character.

- `yes` means that the character(s) separating the individual entries are not imported into the relevant entry's core thus instructing the system to behave as it would by default.
- `no` instructs the system to interpret the character(s) separating the entries as the last character of the first of the two separate entries. Thus including it in the core of the latter.

7.3.1.2.1.3 The `signs=` parameter

The `signs=` parameter accepts one of the keywords `yes` or `no` as a value. Both of them can be abbreviated to their respective first character.

- `yes` instructs the system to include the flags in the database as part of the entry's core, hence to behave as it would by default.
- `no` means that the flags determine the entry's status but are not included in the database as part of the entry's core.

7.3.1.2.1.4 The `form=` parameter

The `form=` parameter accepts the keywords `yes` or `no` as values. Both of them can be abbreviated to their respective first character.

- `yes` instructs the system to ensure that words in the text are only separated by a single blank space. If the system comes across several spaces in a row in the input data it ignores the second and all subsequent spaces. This is the system's default setting.
- `no` instructs the system to include all blank spaces appearing in the input data in the database.

7.3.1.2.1.5 The `also=` parameter

The `also=` parameter accepts the keywords `yes` or `no` as values. Both of them can be abbreviated to their respective first character.

- `yes` instructs the system to interpret the character specified in the `part=` parameter as a separator between individual records. To support this function more precisely, the system only interprets these characters as separators between two entries if the other surrounding characters do not indicate that an abbreviation is involved. Currently, this means that a character specified in `part=` is only interpreted as the end of an entry if:
    - at least one space appears after it followed by a capital letter or by a row of spaces followed by a capital letter and
    - a lower-case letter appears in front of it.

  This is the system's default setting.
- `no` means that the characters specified in `part=` are always interpreted as the end of an entry.

  If you did *not* specify a `part=` parameter, the `also=` parameter has no effect.

7.3.1.2.1.6 The `null=` parameter

The `null=` parameter accepts the keywords `no` or `yes` as values. Both of them keywords can be abbreviated to their respective first character.

If the list of special characters you specified in the `part` parameter contains a space, the system removes it before saving the special characters as separators. This means you can never use a blank space as a separator between entries.

- `no` explicitly instructs the system to behave in this way.
- `yes` instructs the system to treat blank spaces as separators for entries, regardless of the list of characters you may have defined in `part`.

Note: this method is not an efficient way of breaking down continuous text into individual words.

7.3.1.3 The `date` definition

Definitions with a parameter value of `usage=date` accept the following directives:

type
exit (MUST be the last directive)

Definitions of this kind define how the system should treat calendar dates, i.e. how it should convert calendar texts into `date` data within a database or process constants intended for comparison with such elements.

By default, data of this data type is converted according to the following rules:

- The system expects calendar dates in the "Day.Month.Year" format.
- The system expects all three parts of the date to appear in numerical form.
- The system assumes that each year begins on the first of January.

The user can change these settings, as described below. Regardless of any changes which the user may make, the following *always* applies:

- Wherever abbreviations and keywords are referred to in the following explanation note that any distinctions between upper-case and lower-case characters are *always* ignored.
- The default flags – the question mark and exclamation mark – are accepted as date components; first, they are included in the status of the entry concerned and second, the system remembers whether they were assigned to the day, month or year of the date in question.
- In all date components which can be expressed in numerical form (day, month, year), the number zero means unknown.
- You can link two calendar dates together by a hyphen. They are then interpreted as *terminus ante quem* and *terminus post quem*) respectively.

In all calendar notations derived from *Christian* calendars, the following also applies:

- The contractions `OS`[1] and `NS` are interpreted as "Old Style" and "New Style"; i.e. the date is treated as Julian or Gregorian regardless of whether or not it precedes the local introduction of the Gregorian calendar.
- The contractions `CCS`, `ROM`, `ANS`, `PAS`, `BS` and `NC` indicate that the dates next to which they appear relate to years beginning on 1 January , 1 March, 25 March, Easter Sunday, 1 September or 25 December respectively regardless of the new year specified in the `start` parameter described in section 7.3.1.3.1.4.

You can explicitly instruct the system to behave in this way by arranging to call in a logical object "`date`" of the `date` type for processing.

Using an explicitly defined logical object, you can also modify this behaviour as follows.

[1] `AS` for "Alter Stil" in Systems with German error messages.

7.3.1.3.1 The `type` directive

The general form of the `type` directive is:

NAME=Numbers | Western | Islam | Revolution | Byzantine | Moses | Latin | Saints	**(default: Numbers)**
SAME=No | Yes	**(default: No)**
FIRST=Day | Month | Year	**(default: Day)**
SECOnd=Month | Day | Year	**(default: Month)**
STARt=Circumcision | Roman | Easter | Byzantine | Nativity | Annunciation	**(default: Circumcision)**
IDENtification=<list of constants>	**(default: dependent on style)**
DATE=<name of definition>	**(default: None)**
SIGN=<character string>	**(default: None)**

The directive allows you to define how the system should process a particular class of calendar dates. If you use this directive several times, the system applies each selected calendar style in turn to the date which it is trying to convert, until it finds a style which allows it to interpret the date.

So if you defined three `type` directives assigning the values `latin`, `moses` and `numbers` to the `name=` parameter one after another, $\kappa\lambda\epsilon\iota\omega$ would first try and interpret the date according to the rules of the Roman calendar. If this did not work, the system would apply the rules of the Jewish calendar: if this also failed to work, $\kappa\lambda\epsilon\iota\omega$ would attempt to interpret the date using `numbers` notation.

7.3.1.3.1.1 The `name=` parameter

This parameter accepts the keywords `numbers`, `western`, `islam`, `revolution`, `byzantine`, `moses`, `latin` or `saints` as values all of which can be abbreviated to their respective first character. It instructs the system to interpret the calendar date on the basis of a given style of notation. The individual keywords cause the system to proceed as follows:

- `numbers`: day, month and year are expressed as Arabic numerals and separated by full stops (periods). The date is interpreted as a date in the Christian calendar *after* the birth of Christ.
- `western`: the components of the date are separated by blank spaces; day and year are expressed as Arabic numerals, month by one of the following names of months, which can be abbreviated to their first three characters only: `unknown`, `January`, `February`, `March`, `April`, `May`, `June`, `July`, `August`, `September`, `October`, `November` and `December`. The date is interpreted as a date in the Christian calendar *after* the birth of Christ.
- `islam`: the components of the date are separated by blank spaces; day and year are expressed as Arabic numerals, month by one of the following names of months, which cannot be abbreviated in the current version of the software: `unknown`, `Muharram`, `Safar`, `Rabi1`, `Rabi2`, `Jumada1`, `Jumada2`, `Rajab`, `Sha'ban`, `Ramadan`, `Shawwal`, `Dhul-qa'da` and `Dhul-hijjah`. The date is interpreted as a date in the Islamic calendar *after* the Hijrah.

- `revolution`: the components of the date are separated by blank spaces; day and year are expressed as Arabic numerals, month by one of the following names of months, which can be abbreviated to their first four characters: `unknown`, `vendemiaire`, `brumaire`, `frimaire`, `nivose`, `pluviose`, `ventose`, `germinal`, `floreal`, `prairial`, `messidor`, `thermidor`, `fructidor` and `jour`. The date is interpreted as a date in the French Revolutionary calendary for years I to XIV of the Republic.
- `byzantine`: the components of the date are separated by blank spaces: day and year are expressed as Arabic numerals, month *for the time being* [2] by one of the following names of months, which can be abbreviated to their first three characters: `unknown`, `January`, `February`, `March`, `April`, `May`, `June`, `July`, `August`, `September`, `October`, `November`, and `December`. The date is interpreted as a date in the Byzantine Era; any of the additional notations permitted in Christian dating systems are also valid in this context. In accordance with the practice in Russia the year begins on the first day in January, as from 1 January 1700; in the case of earlier dates the new year begins on 1 September.
- `moses`: the components of the date are separated by blank spaces; day and year are expressed as Arabic numerals, month by one of the following names of months, which can be abbreviated to their first three characters: `unknown`, `Tishri`, `Cheshvan`, `Kislev`, `Tevet`, `Shevat`, `Adar`, `ve-Adar`, `Nisan`, `Iyyar`, `Sivan`, `Tammuz`, `Av` and `Elul`. The date is interpreted as a date in the Jewish Era.
- `latin`: the components of the date are separated by blank spaces. The system expects the date to be expressed in days relative to the Kalendae, Nonae or Ides, as defined in the traditional manuals of chronology, like Cheney's *Handbook of Dates* or Grotefend's *Taschenbuch der Zeitrechnung*, taking into account the "bis" notation for leap years. The Roman names for the months can be abbreviated to their first three letters. The date is interpreted as a date in the Christian calendar *after* the birth of Christ. However, the number of days relative to the Kalendae, Nonae or Ides has to be given in Arabic numerals.
- `saints`: the date is formulated according to the rules of a calendar based on Church feast days. These rules are described in detail in section 7.3.1.18.1. If you assign this value to the `name=` parameter, you must also include a `date` parameter in the `type` directive.

7.3.1.3.1.1.1 The `same=` parameter

The `same=` parameter accepts the keywords `no` or `yes` as values. Both keywords can be abbreviated to their respective first character.

This parameter is meaningful only with the calendar style `latin`. In that style the standard assumption is, that the year given with a date of the form $<$ *number* $>$ `kal Jan` refers to the year of January. So `18 kal Jan 1244` would be resolved as 15^{th} of December 1243.

[2] This implementation of the Byzantine calendar should be regarded as temporary. We are searching for appropriate data records for testing purposes.

- `no` explicitly instructs the system to behave in this way.
- `yes` instructs the system to assume, that the year quoted is the same year in which the December is located. With this parameter value the date given above would therefore be understood to be the 15^{th} of December 1244.

7.3.1.3.1.2 The `first=` parameter

This parameter accepts any of the keywords `day`, `month` or `year` as values; they can be abbreviated to their respective first character. You can apply it to any style of calendar notation except the Roman: it is used to define which of the three date components should be written first. If you specify this parameter you *must* also specify `second=`.

7.3.1.3.1.3 The `second=` parameter

This parameter accepts the keywords `day`, `month` or `year` as values, and they can be abbreviated to their respective first character. You can apply to any style of calendar notation except the Roman; it is used to define which of the three date components should be written second. If you specify this parameter you *must* also specify a `first=` parameter.

7.3.1.3.1.4 The `start=` parameter

This parameter accepts any of the keywords `circumcision`, `roman`, `annunciation`, `easter`, `byzantine` or `nativity` as a value; they can all be abbreviated to their first character. This parameter instructs the system to begin the new year on a given date in the calendar style you selected in the `type` directive. Because this only makes sense in the context of Christian calendar styles, the system only accepts this parameter if you specify `numbers`, `western`, `byzantine`, `latin` or `saints` in the `name=` parameter. The keywords are interpreted as follows:

- `circumcision`: year begins on 1 January (default setting).
- `roman`: year begins on 1 March.
- `annunciation`: year begins on 25 March.
- `easter`: year begins on Easter Sunday. [3]
- `byzantine`: year begins on 1 September.
- `nativity`: year begins on 25 December.

7.3.1.3.1.5 The `identification=` parameter

This parameter accepts as a value a list of comma-delimited alphanumeric constants which are interpreted as the names of months. It is valid for all styles of calendar notation except the `numbers`. The list must contain a number of constants equal to the number of months which appear in the calendar notation in question. In particular, it must also include some way of writing the "unknown" month which appears in each style before the first actual month.

The user is currently responsible for making sure that the following program characteristic is adhered to. You must write enough letters at the beginning of the name for each month in the `identification=` parameter to allow the system to identify each month uniquely.

[3] Currently, κλειω only calculates the actual change of the year, making no distinction between days *post* and *ante pascha*. If you require this form of new year, please let us know, as we do not currently have access to a realistic set of data for test purposes.

7.3.1.3.1.6 The `date=` parameter

This parameter accepts the name of a chronology definition as a value (cf. section 7.3.1.18). It tells the system which definition to use in order to process the contents of this element.

This parameter is only valid if you assigned the keyword `saints` to the `name=` parameter.

7.3.1.3.1.7 The `sign=` parameter

This parameter accepts as parameter value a character string of *up to twelve characters*. It can be used to specify which of a set of calendar styles shall be selected in a given case.

Assume the following situation: In a source there are dates of two communities, both of which use the same notational style, community "A", however, starts the year with January 1^{st}, while in community "B" the year starts on Christmas. A definition of the form:

```
type name=western
type name=western;start=nativity
```

would not solve the problem with data like:

```
... /15 December 1244 # community "A"/ ...
... /15 December 1244 # community "B"/ ...
```

as both dates can be resolved according to the `western` style and would therefore be converted according to the first of the two `type` directives.

`sign=` allows in such cases to specify explicitly in the data which style shall apply. With the definition:

```
type name=western
type name=western;start=nativity;sign="B-style"
```

the data can be input as

```
... /15 December 1244 # community "A"/ ...
... /15 December 1244 B-style # community "B"/ ...
```

and the correct result is guaranteed.

7.3.1.4 The `number` definition

Definitions with a parameter value of `usage=number` accept the following directives:

signs	**may only be specified once**
simple	**may only be specified once**
double	**may only be specified once**
triplet	**may only be specified once**
text	
evaluation	**may only be specified once**
exit	**(MUST appear as the last directive)**

Definitions of this kind define how the system should treat numerical data, i.e. how it should convert numerical texts into `number` data within a database or process constants intended for comparison with such elements.

By default, data of this data type is converted according to the following rules:

- The system expects decimal numbers: where they contain decimal places a full stop (period) should be used as the decimal separator.
- The standard flags, the question mark ("?") and exclamation mark ("!"), are included in the status of the element concerned.
- Instead of individual numbers, you can formulate an expression using the operators ("+") (addition), ("-") (subtraction), ("*") (multiplication) and (":") (division). This expression can be as complex as you like.
- Numbers separated by spaces are interpreted as if they were linked by the addition operator ("+").
- *These expressions are always resolved from left to right. All arithmetic operations are strictly performed from left to right; thus the expression '2+3*4-5' results in '15' and not '9'.*
- You can group these expressions in parentheses ("(") and (")"), nested to any depth, in order to control the order in which they are calculated.
- You can insert any combination of *uncertainty operators* in front of any item of information of this data type, *but not in front of every number in an expression.* You can use any of the following uncertainty operators; `equal`, `circa`, `less` or `greater`. They can all be abbreviated to their respective first two characters. You can combine these uncertainty operators to create expressions which are as complex as you like; however the system will always analyse these combinations in terms of the most complex of the combinations appearing below: `equal circa`, `equal less`, `equal greater`, `circa less`, `circa greater`, `equal circa less` and `equal circa greater` (cf. section 8.1.2.3.3 for an explanation of how these operators are interpreted).

You can explicitly instruct the system to behave in this way by arranging to call in a logical object "`number`" of the `number` type for processing.

Using an explicitly defined logical object, you can also modify this behaviour as follows.

7.3.1.4.1 The `signs` directive

The general form of the `signs` directive is:

ORDEr=Yes | No	**(default: Yes)**
PRECise=No | Yes	**(default: No)**
CUMUlate=Yes | No	**(default: Yes)**
PART=<character>	**(default: none)**
MORE=<character>	**(default: none)**
SEQUence=No | Yes	**(default: No)**
TEXT=No | Yes	**(default: No)**

This directive is used to define the operators in use and describe the way the other directives interact in greater detail.

7.3.1.4.1.1 The `order=` parameter

The `order=` parameter accepts the keywords `yes` or `no` as values. Both of them can be abbreviated to their respective first character.

- `yes` means that the order of the parts of an arithmetical expression determines the order in which the system should perform each arithmetical operations, hence explicitly instructs the system to behave as it would by default.
- `no` means that the mathematical precedence of the operators is taken into account, i.e. that — unless indicated otherwise by bracketing — multiplication and division are performed before addition and subtraction.

7.3.1.4.1.2 The `precise=` parameter

The `precise=` parameter accepts one of the keywords `no` or `yes` as a value. Both of them can be abbreviated to their respective first character.

- `no` instructs the system to interpret uncertainty operators, thus instructs the system to behave as it would by default.
- `yes` means that these operators are no longer known to the system, so that it interprets them as errors whenever they appear.

7.3.1.4.1.3 The `cumulate=` parameter

The `cumulate=` parameter accepts the keywords `yes` or `no` as values. Both of them can be abbreviated to their respective first character.

- `yes` means that blank spaces are analysed as addition operators, thus instructs the system to behave as it would by default. In the case of qualified numbers, as introduced by the `text` directive for example (cf. section 7.3.1.4.5), this implies that an identifier separated by a space from a preceding number is interpreted as a qualifier of the number one.
- `no` means that spaces are not interpreted as operators. In this condition a qualifier following a number and separated from it by any number of spaces is interpreted as the qualifier of this number. In this case, you should avoid writing several numbers separated by blank spaces in a row as the system will interpret this as an error.

7.3.1.4.1.4 The `part=` parameter

This parameter expects an otherwise unused character as a value. This is interpreted as an additional operator which allows two successive characters (or expressions) to be interpreted as interval boundaries. This "from – to" operator may only be used once in each entry.

7.3.1.4.1.5 The `more=` parameter

This parameter expects an otherwise unused character as a parameter value. This is interpreted as an additional operator, which separates two (or more) successive qualifiers into one primary and one (or more) secondary qualifiers, as described in section 7.3.1.4.5.

7.3.1.4.1.6 The `sequence=` parameter

The `sequence=` parameter accepts the keywords `yes` or `no` as values. Both of these can be shortened to their respective first character.

- `no` means that where a sequence consisting of $< number >< space >< qualifier >$ appears the system treats the space in the middle as a separator, i.e. as an addition sign or error, depending on the value of the `cumulate=` parameter. This is the default system setting.
- `yes` means that the system ignores any spaces between numbers and qualifiers, so that the sequence alone is sufficient to set up a relationship between number and qualifier.

7.3.1.4.1.7 The `text=` parameter

The `text=` parameter accepts the keywords `no` or `yes` as values. Both of them can be abbreviated to their respective first character.

- `no` means that only the result of the conversion to the numeric data type shall be preserved in the database. This is the default behaviour of the system.
- `yes` means that the textual representation shall be preserved as well.

This parameter has the same function as the `text=` parameter of the `number` command (cf. section 7.3.4.3.2). While this command defines the behaviour for *all* data of datatype `number`, however, the currently discussed parameter allows to differentiate in this respect between individual `number` definitions.

7.3.1.4.2 The `simple` directive

The general form of the `simple` directive is:

MINImum=<number> **(default: 0.0)**
MAXImum=<number> **(default: 1000000.0)**
NUMBer=<number> **(default: 1.0)**

It defines how the system should treat *simple numbers*, i.e. numbers which do not have either a decimal separator or a qualifier.

7.3.1.4.2.1 The `minimum=` parameter

This parameter instructs the system to reject any number less than the value assigned to this parameter.

7.3.1.4.2.2 The `maximum=` parameter

This parameter instructs the system to reject any number greater than the value assigned to this parameter.

7.3.1.4.2.3 The `number=` parameter

This parameter instructs the system to multiply every simple number by the value assigned to this parameter before using the number in arithmetical operations.

7.3.1.4.3 The `double` directive

The general form of the `double` directive is:

NUMBer=<number>	**(default: 1.0)**
LIMIt=<character>	**(default: '.')**
FIRST=<number>	**(default: none)**
SECOnd=<number>	**(default: conversion into decimal fraction)**
MINImum=<number>	**(default: 0.0; can be specified several times)**
MAXImum=<number>	**(default: 1000000.0; can be specified several times)**
SIGN=<character>	**(default: none)**
PART=<number>	**(default: none)**
CUMUlate=No | Yes	**(default: No)**

This directive defines how the system should process *split numbers*, i.e. numbers consisting of two parts linked by a separator. When it finds such a number the system always starts by processing the two parts separately. This "processing" means that the system converts the part of the number to the right of the decimal point into a decimal fraction: it then adds both parts of the number together.

7.3.1.4.3.1 The `number=` parameter

This parameter instructs the system to multiply every split number by the value assigned to this parameter – after both parts of the number have been added together, but before the number is used in any other arithmetical operations.

7.3.1.4.3.2 The `limit=` parameter

This parameter introduces the character assigned to it as the operator which separates both parts of a split number; if you do not define this parameter the system uses a decimal point as the default separator.[4]

7.3.1.4.3.3 The `first=` parameter

This parameter instructs the system to multiply the part of the split number in front of the separator by the numerical value assigned to this parameter, before adding the two parts of the number together to obtain a total.

7.3.1.4.3.4 The `second=` parameter

This parameter instructs the system to multiply the part of the split number after the separator by the numerical value assigned to this parameter, before adding the two parts of the number together to obtain a total. If you do not define this parameter the system treats the part of the number following the separator as a decimal fraction. If you do not define either this parameter or the `limit=` parameter, the system will treat these numbers as "normal" decimal numbers with decimal points.

---

[4] If you are using split numbers and triplets simultaneously you can use the value assigned to `limit=` for both types of number. In general, however, we would strongly suggest that you use different separators for the two different types of number.

7.3.1.4.3.5 The `minimum=` parameter

This parameter instructs the system to reject any number less than the value assigned to this parameter. This parameter should only appear after `first=` or `second=`. Its position indicates to the system whether it is referring to the first or second part of the split number.

7.3.1.4.3.6 The `maximum=` parameter

This parameter instructs the system to reject any number greater than the value assigned to this parameter. This parameter should only appear after `first=` or `second=`. Its position indicates to the system whether it is referring to the first or second part of the split number.

7.3.1.4.3.7 The `sign=` parameter

This parameter introduces the character assigned to it as an addition operator which further subdivides the two parts of a split number. The system treats the two resultant addition subdivisions in the same way as the original parts of the split number: i.e. it adds them together before processing them together as a part of the original split number. [5]

7.3.1.4.3.8 The `part=` parameter

This parameter instructs the system to multiply the part of the number to the right of the character introduced by `sign=` by the value assigned to the `part=` parameter, before going on to use the number in other operations. If you do not specify `part=`, the system treats the character assigned to `sign=` as a decimal separator in a non-decimal split number.

7.3.1.4.3.9 The `cumulate=` parameter

The `cumulate=` parameter accepts either of the keywords `no` or `yes` as a value. Both of them can be shortened to their respective first letter.

- `no` means, that a *qualified split number* shall be treated as a qualified decimal number, irrespective of any `double` directive. So `2.5Pd` would be treated as two and one half `Pd`s whether or not a `double` directive has redefined the relationship between the two numeric parts. This is the default behaviour of the system.
- `yes` means, that in such cases the `double` directive should be considered: using the qualifier to differentiate, e.g., between "paper" and "metal" of the same non-decimal currency system.

---

[5] If you are using split numbers and triplets simultaneously you can use the value assigned to `sign=` for both types of number. In general, however, we would strongly advise you to use different secondary separators for the two different types of number.

7.3.1.4.4 The `triplet` directive

The general form of the `triplet` directive is:

LIMIt=<character>	**(default: '.')**
FIRST=<character>	**(MUST be defined)**
SECOnd=<number>	**(MUST be defined)**
MORE=<number>	**(MUST be defined)**
MINImum=<number>	**(default: 0.0; can be specified more than once)**
MAXImum=<number>	**(default: 1000000.0; can be specified more than once)**
SIGN=<character>	**(default: none)**
PART=<number>	**(default: none)**

This directive defines how the system should process *triplets*, i.e. numbers consisting of three parts separated by two occurrences of the same separator. When it encounters such a number the system always starts by processing the three parts separately and then adds them together once it has performed any necessary operations.

7.3.1.4.4.1 The `limit=` parameter

This parameter introduces the character assigned to it as the operator which separates the three parts of a triplet; if you do not define this parameter the system uses a decimal point as the default separator. [6]

7.3.1.4.4.2 The `first=` parameter

This parameter instructs the system to multiply the part of the triplet in front of the first separator by the numerical value assigned to this parameter before adding all three parts of the number together. You must specify this parameter.

7.3.1.4.4.3 The `second=` parameter

This parameter instructs the system to multiply the part of the triplet positioned between the two separators by the numerical value assigned to this parameter before adding all three parts of the number together. You must specify this parameter.

7.3.1.4.4.4 The `more=` parameter

This parameter instructs the system to multiply the part of the triplet after the second separator by the numerical value assigned to this parameter before adding all three parts of this number together. You must specify this parameter.

[6] If you are using split numbers and triplets simultaneously you can use the value assigned to `limit=` for both types of number. In general, however, we would strongly advise you to use different separators for the two different types of number.

7.3.1.4.4.5 The `minimum=` parameter

This parameter instructs the system to reject any number less than the value assigned to this parameter. This parameter should only appear after `first=`, `second=` or `more=`. Its position indicates to the system whether it is referring to the first, second or third part of the triplet.

7.3.1.4.4.6 The `maximum=` parameter

This parameter instructs the system to reject a number greater than the value assigned to this parameter. This parameter should only appear after `first=`, `second=` or `more=`. Its position indicates to the system whether it is referring to the first, second or third part of the triplet.

7.3.1.4.4.7 The `sign=` parameter

This parameter introduces the value assigned to it as an addition operator, which further subdivides the three parts of a triplet. The system treats the two resultant subdivisions in the same way as the original parts of the triplet: i.e. it adds them together before processing them together as part of the original triplet. [7]

7.3.1.4.4.8 The `part=` parameter

This parameter instructs the system to multiply the part of the number to the right of the character introduced by `sign=` by the value assigned to the `part=` parameter before going on to use this part of the number in other operations. If you do not specify `part=` the system treats the character assigned to `sign=` as a decimal separator in a non-decimal triplet.

[7] If you are using split numbers and triplets simultaneously you can use the value assigned to `sign=` for both types of number. In general, however, we would strongly advise you to use different secondary separators for the two different types of number.

7.3.1.4.5 The `text` directive

The general form of the `text` directive is:

NAME=<character string>	**(MUST be specified)**
NUMBer=<number>	**(MUST be specified)**
MORE=No | Yes	**(default: No)**
MINImum=<number>	**(default: 0.0)**
MAXImum=<number>	**(default: 1000000.0)**

This directive defines how the system should process *qualified numbers*, i.e. numbers beginning with a numerical part – which can be a simple number, a split number or a triplet – and then defined more precisely by an immediately following alphanumeric character string (cf. section 7.3.1.4.1.3 above) known as the qualifier. Before going on to process such a number further the system first multiples its numerical part by the numerical value corresponding to the number's qualifier. If the number possesses several qualifiers, separated by a multiple qualification operator (cf. section 7.3.1.4.1.5 above), the system performs this multiplication operation several times in succession.

7.3.1.4.5.1 The `name=` parameter

The value of the `name=` parameter is a character string beginning with a letter and not containing any of the currently active numerical operators or any data signal characters. The system interprets this character string as a qualifier described more precisely by the other parameters.

7.3.1.4.5.2 The `number=` parameter

This parameter expects a number as a parameter value. The system regards it as the numerical equivalent of an operator, used for resolving qualified numbers.

7.3.1.4.5.3 The `more=` parameter

The `more=` parameter accepts either of the keywords `yes` or `no` as a value. Both of them can be shortened to their respective first letters.

- `no` means that the value assigned to `text` must appear immediately after the number it qualifies; this is the system's default setting.
- `yes` means that the system only recognises the qualifier if it appears after a multiple qualification operator (cf. section 7.3.1.4.1.5 above). If you want the system to identify a qualifier in either position you should introduce the qualifier in two separate `text` directives.

7.3.1.4.5.4 The `minimum=` parameter

This parameter instructs the system to reject any qualified number, the numerical part of which is less than the value assigned to this parameter.

7.3.1.4.5.5 The `maximum=` parameter

This parameter instructs the system to reject any qualified number, the numerical part of which is greater than the value assigned to this parameter.

7.3.1.4.6 The `evaluate` directive

The general form of the `evaluate` directive is:

NUMBer=<number> (default: 1.0)

It defines how the system should treat a numerical result once it has been converted into the system's own internal format.

7.3.1.4.6.1 The `number=` parameter

This parameter expects a number as a value. In a final step during the conversion of input numbers into the system's own internal format. The system multiplies the numerical result to date by this number.

7.3.1.5 The `category` definition

Definitions with a parameter value of `usage=category` accept the following directives:

text **may only be specified once**
part
signs
exit **(MUST be the last directive in the series)**

Definitions of this kind define how the system should deal with categorical abbreviations, i.e. how it should convert such texts into `category` data in the database or process constants intended for comparison with such elements.

By default data of this data type is converted according to the following rules:

The following settings always apply:

- Every character is treated as an independent value.
- The system does *not* keep to the original order of the characters.
- You can define a maximum of 32 different characters per `category` definition.

You can control the following settings:

- Question marks and exclamation marks may appear in any combination, but only ever once.
- Question marks and exclamations marks appear as themselves in all printouts.
- When either of these characters is converted into the `number` data type it is represented by a number which reflects the order in which the original characters were included in the codebook.
- After displaying an error message the system ignores any other characters.
- The system makes a distinction between upper-case and lower-case.

You can explicitly instruct the system to behave in this way by arranging to call in the logical object "`category`" of the `category` type for processing.

Using an explicitly defined logical object, you can also modify this behaviour as follows:

7.3.1.5.1 The `signs` directive

The general form of the `signs` directive is:

SIGNs=<character>	**(MUST be specified)**
WRITe=<character string>	**(default: input character)**
NUMBer=<number>	**(default: ordinal number)**

It defines which characters the system may process and how they should appear in printouts etc.

7.3.1.5.1.1 The `signs=` parameter

This parameter expects a value consisting of one character which you are not currently using as a data signal character. It tells the system that it may use this character.

7.3.1.5.1.2 The `write=` parameter

This parameter expects a character string as a value. In all processing situations in which the contents of the converted entries must be converted into printer characters for a given processing operation, the character assigned to `signs=` is replaced by the character string assigned to `write=`.

7.3.1.5.1.3 The `number=` parameter

This parameter expects a number as a parameter value. This parameter is used whenever the same value in a entry of the `category` data type needs to be converted into a `number` data type, e.g. in preparation for transfer to a statistics program.

7.3.1.5.2 The `part` directive

In this version of $\kappa\lambda\epsilon\iota\omega$ you cannot assign any parameters to the `part` directive.

It controls the way in which characters defined by `signs=` can be grouped together. The following rules apply:

- You can combine characters which were defined before the first `part` directive with other characters just as you like.
- You can combine characters defined after a `part` directive with any characters which do not appear between the same pair of `part` directives (or after the last `part` directive in the definition).

7.3.1.5.3 The `text` directive

The general form of the `text` directive is:

SIGNs=Yes | No   (default: Yes)

This directive controls the sensitivity of the definition covering differences in spelling.

7.3.1.5.3.1 The `signs=` parameter

The `signs=` parameter accepts the keywords `yes` or `no` as values. Both of them can be shortened to their respective first character.

- `yes` means that the system distinguishes between upper-case and lower-case characters. This is the system's default setting,
- `no` means that this distinction is ignored.

7.3.1.6 The `relation` definition

Definitions with a parameter value of `usage=relation` accept the following directives:

redundancy	**may only be specified once**
part	**may only be specified once**
bridge	
exit	**(MUST be the last directive in the series)**

Definitions of this type define how the system should treat non-hierarchical relationships, i.e. how it should convert such texts into `relation` data in a database or process constants intended for comparison with such elements.

By default the system uses the following rules to convert data of this data type:

- If a network identifier only appears once in a group it is tacitly suppressed as it can not be used to set up a linkage.
- If a network identifier appears several times in a group, in such a way that the group refers to itself, this reference is tacitly suppressed.
- While the system is performing a `read` task, it collects together all network identifiers and – unless it has already done so in the course of the `read` task – integrates them into the database as references to relationships, at the latest before it completes the task.
- The file which the system uses to collect these network identifiers together is deleted once the `read` task has been completed.

You can explicitly instruct the system to behave in this way by arranging to call in the logical object "`relation`" of the `relation` type for processing.

Using an explicitly defined logical object, you can also modify this behaviour as follows.

7.3.1.6.1 The `redundancy` directive

The general form of the `redundancy` directive is:

WARNings=No | Yes (default: No)
SELF=No | Yes (default: No)

It controls the number and scope of the warnings issued to the user whenever the system suppresses redundant network identifiers.

7.3.1.6.1.1 The `warnings=` parameter

The `warnings=` parameter accepts the keywords `yes` or `no` as values. Both of them can be shortened to their respective first character.

- `no` instructs the system to suppress network identifiers which only appear once, thus behaving as it would by default.
- `yes` instructs the system to print out a warning every time it suppresses a network identifier.

7.3.1.6.1.2 The `self=` parameter

The `self=` parameter accepts the keywords `yes` or `no` as values. Both of them can be shortened to their respective first character.

- `no` instructs the system to suppress any network identifiers which appear several times in a group and hence causes this group to refer to itself.
- `yes` instructs the system to print out a warning every time it suppresses a network identifier in this way.

7.3.1.6.2 The `part` directive

The general form of the `part` directive is:

TYPE=Temporary | Permanent (default: Temporary)
PART=Yes | No (default: Yes)

It controls the exact moment at which the system actually creates the network of relationships.

7.3.1.6.2.1 The `type=` parameter

The `type=` parameter accepts the keywords `temporary` or `permanent` as values. Both of these can be shortened to their respective first character.

- `temporary` instructs the system to delete the file in which it is temporarily storing any network identifiers which it has not yet processed once it has completed the `read` task: it thus explicitly instructs the system to behave as it would by default.
- `permanent` instructs the system to save this file for further use.

7.3.1.6.2.2 The `part=` parameter

The `part=` parameter accepts the keywords `yes` or `no` as values. Both of these can be shortened to their respective first character.

- `yes` instructs the system to integrate any network identifiers which have not yet been integrated into the database by the time it finishes the `read` task; thus it explicitly instructs the system to behave as it would by default.
- `no` prevents the system from doing this when it completes the `read` task, so keeping any network identifiers which the system has not yet linked together in their temporary condition.

7.3.1.6.3 The `bridge` directive

The general form of the `bridge` directive is:

NAME=<database name> (default: none)

The purpose of this directive is the introduction of `relation` definitions, which are prepared to link a number of databases to the current one. In normal use of the system, this directive will scarcely be used, as the services it performs are provided implicitly by the `bridge` command.

7.3.1.6.3.1 The `name=` parameter

The `name=` parameter expects as parameter value the name of a database which does not need to exist at the time of execution of this directive.

7.3.1.7 The `conversion` definition

Definitions with a parameter value of `usage=conversion` accept the following directives:

type	**(Can be specified more than once)**
substitution	**(MUST be specified at least once, can be specified more than once)**
cycle	
limit	
training	
exit	**(MUST be the last directive in the series)**

This directive is used to define systematic changes which the system should make to a character string before processing the latter any further. This kind of definition is distinguished by the fact that it rarely appears alone. It is often used in opening and/or closing operations by other classes of logical object.

7.3.1.7.1 The `type` directive

The general form of the `type` directive is:

TYPE=Insertion| Substitution | Destruction | Creation	**(default: Insertion)**
USAGe=Long| Short	**(default: Short)**
WITHout=<character string>	**(default: none)**
CUMUlate=Yes | No	**(default: No)**
LIMIt=Yes | No	**(default: none)**
STARt=Yes | No	**(default: none)**
ONLY=Yes | No	**(default: none)**

You use this directive to define the nature of any substitutions. You do not have to specify it if you are happy with the default settings, but you can define it several times in one `conversion` definition if you wish.

7.3.1.7.1.1 The `type=` parameter

This parameter expects as a parameter value one of the keywords `insertion`, `creation`, `substitution` or `destruction`. The parameter value you select defines how the system should interpret any subsequent `substitution` or `cycle` directives.

- `insertion` instructs the system to insert a new replacement operation into an already existing definition. This is the system's default setting.
- `creation` instructs the system to create a new replacement operation or new cycle (see below).
- `substitution` instructs the system to overwrite an existing replacement operation for the specified character string.
- `destruction` instructs the system to delete an existing translation operation or existing cycle.

7.3.1.7.1.2 The usage= parameter

This parameter expects one of the keywords short or long as a value. It allows you to control the system's behaviour when making substitutions.

- long instructs the system to replace the longest matching character string it can find in another character string. This is the system's default setting.
- short instructs the system to replace the shortest matching character string it can find in another string.

7.3.1.7.1.3 The without= parameter

This parameter expects a character string as a parameter value. The system ignores the characters you specify in this string while it searches for a character string to replace in the character string which is being processed. This parameter affects all the cycles in a conversion definition. Consequently, you should define it before the first cycle directive appears.

7.3.1.7.1.4 The cumulate= parameter

This parameter expects one of the keywords yes or no as a value. This controls the system's behaviour where a character string which has already been modified can be modified a second time.

- no instructs the system not to modify parts of a character string which it has already modified. This is the system's default setting.
- yes means that substitutions can be cumulative. Thus the system may continue to modify parts of a character string which it has already modified.

Setting this parameter to yes may in certain circumstances result in a conversion definition which generates a logical loop, because every newly generated character string will cause the system to re-apply another rule or even the same rule. So that the system is not inadvertently brought to a grinding halt, every time you activate this parameter you also activate two system checks. In extreme cases these may prevent *κλειω* from performing a conversion which in itself makes sense. The system suspects the presence of a continuous logical loop whenever it encounters one of the two following situations:

- More than one hundred substitution directives in succession can be applied to a single original character string.
- As a result of continuous replacements, an original character string grows to such a length that it is threatens to exceed the maximum (application-dependent) length specified for this particular character string.

In both cases, *κλειω* aborts the conversion process, replaces the character string to be converted with the phrase "*(unknown)" and prints out a warning to the user, followed by the first seventy characters in the unprocessed character string in the state they were in when the situation that generated the error was first discovered.

7.3.1.7.1.5 Parameters for determining where a substitution should be made

Each of the three following parameters affects the way the system interprets the other two; in other words, all three are mutually interdependent. Together they define the positions in a character string at which the substitutions you have specified should take effect. In general, the following applies:

Location of substitution	`limit`	`start`	`only`
Everywhere	–	–	–
At start and in middle	no	–	–
Only at end	yes	–	–
In middle and at end	–	no	–
Only at start	–	yes	–
Only in middle	no	no	–
At start and at end	yes	yes	–
Only in the identifier of both character strings	–	–	yes

7.3.1.7.1.5.1 The `limit=` parameter

This parameter expects one of the keywords `yes` or `no` as a parameter value. They are used to specify the location of possible substitutions.

- if you do not specify the `limit=` parameter, substitutions will not be confined to the end of the character string, although they may refer to it. This is the system's default setting.
- `no` means that substitutions are confined to the beginning and middle of the character string.
- `yes` means that substitutions will take place at the end of the character string as long as you have not *simultaneously* assigned a value of `yes` to the `start=` parameter.

You can combine the `limit=` parameter with other parameters affecting the positioning of substitutions as follows:

- with `start=` set to `yes` and its own keyword set to `yes` if you only want to replace the beginning and end of the character string.
- with `start=` set to `no` and its own keyword set to `no` if you only want to replace the middle of the character string.
- however, you should never combine this parameter with the `only=` parameter.

7.3.1.7.1.5.2 The `start=` parameter

This parameter expects one of the keywords `yes` or `no` as a value. You can also use this parameter to define the location of substitutions.

- if you do not specify this parameter, substitutions will not be confined exclusively to the beginning of the character string, although they may refer to it. This is the system's default setting.
- `no` means that substitutions will only take place in the middle and end of the character string as long as you have not defined any other parameters which specify the location of substitutions.
- `yes` means that substitutions will only take place at the beginning of the character string as long as you have not defined any other parameters which specify the location of substitutions.

You can combine this parameter with other parameters which control the location of subsitutions as follows:

- with `limit=` set to `yes` and its own keyword set to `yes` if you only want to replace the beginning and end of the character string.
- with `limit=` set to `no` and its own keyword set to `no` if you only want to replace the middle of the character string.
- however, you should never combine this parameter with the `only=` parameter.

7.3.1.7.1.5.3 The `only=` parameter

This parameter expects one of the keywords `yes` or `no` as a parameter value. You can also use this parameter to control the location of substitutions.

- `no` means that substitutions may take place anywhere as long as you have not defined any other parameters which specify the location of substitutions. This is the system's default setting.
- `yes` means that the substitution will only take place if the character string to be replaced in this instance is absolutely identical to the character string which forms the object of the replacement operation.
- You should only define this parameter's value as `yes` as long as you have not specified either the `start=` or `limit=` parameters.

7.3.1.7.2 The `substitution` directive

The general form of the `substitution` directive is:

CURRent=<character string> (MUST be specified!)
RESUlt=<character string> (MUST be specified!)

This directive forms the core of a `conversion` definition. You use it to formulate the substitutions you require.

7.3.1.7.2.1 The `current=` parameter

This parameter requires as a value an alphanumeric constant. This is used to define the character string which is to be replaced.

7.3.1.7.2.2 The `result=` parameter

This parameter requires as a value an alphanumeric constant. This is used to define the character string which should replace the string defined in the `current=` parameter.

7.3.1.7.3 The `cycle` directive

The general form of the `cycle` directive is:

ORDEr=<number> (default: context-sensitive)

You can use this directive to generate or delete cycles. A *cycle* is defined as the sum total of a number of substitution directives, combined in one logical unit. Any character string to be converted is always processed from left to right; i.e. beginning with the first character and continuing with each character in succession, the system attempts to apply the existing substitution directives to the part of the character string beginning with that parcicular character. At the end of one cycle, the converted character string is passed on to the next, and the system applies the directives in the next cycle to the new character string, once again beginning with its first character.

This directive is only valid without additional parameters if you set the parameters in the preceding `type` directive to `type=creation` or `type=destruction`. In the first case a new cycle is added to the definition, in the second case the cycle which has just been processed is deleted.

7.3.1.7.3.1 The `order=` parameter

This parameter expects a number as a parameter value.

- If you did not declare `type=creation` or `type=destruction` in the `type` directive κλειω attempts to find a cycle with an ordinal number matching this parameter's numerical value. If the system cannot find a cycle with this ordinal number, it displays an error message.
- If you declared `type=destruction` in the `type` directive the system deletes the cycle with the ordinal number corresponding to this parameter's value. If no such cycle exists, an error message appears.
- If you declared `type=creation` in the `type` directive, the system creates a new cycle for this definition, with an ordinal number corresponding to this parameter's value.

7.3.1.7.4 The `limit` directive

The general form of the `limit` directive is:

LIMIt=<Number> (default: 100)

Within each cycle *κλειω* checks, how many substitutions have been performed upon any string handed to the conversion algorithm. If this number reaches acertain limit, the conversion is aborted, as the algorithm is suspected to loop endlessly. (Appropriate diagnostics are displayed.) By default that limit is set to 100 substitutions.

7.3.1.7.4.1 The `limit=` parameter

This parameter accepts as parameter value a number, which replaces the current limit for legal substitions in the current cycle of the algorithm.

7.3.1.7.5 The `training` directive

The general form of the `training` directive is:

CUMUlate=Yes | No (default: Yes)

You can test the `conversion` definition by specifying this directive. After this directive *κλειω* expects lines containing one character string each. Each of these strings will immediately be processed by the specified definition. A line starting with an asterisk `‘‘*’’` ends this training mode.

7.3.1.7.5.1 The `cumulate=` parameter

This parameter expects one of the keywords `yes` or `no` as a value. For test purposes, you can specify whether character strings should be processed cumulatively by each separate cycle.

- `yes` means that as from the present cycle each cycle affects the character string cumulatively, albeit in test mode.
- `no` means that each cycle is self-sufficient.

7.3.1.8 The `soundex` definition

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

conversion
part **(Can be specified more than once)**
training
exit **(MUST be the last directive in the series)**

Using a *Soundex* algorithm, the system converts the characters in a character string into a "phonetic" [8] coded value. You should explicitly specify the coding scheme used to make the conversion in a `soundex` definition.

7.3.1.8.1 The `conversion` directive

The general form of the `conversion` directive is:

PREParation=<name>	**(default: none)**
AFTEr=<name>	**(default: none)**
FIRSt=No | Yes	**(default: No)**
MAXImum=<number>	**(default: 4)**
WITHout=<character string>	**(default: none)**
FORM=No | Yes	**(default: No)**

This directive determines the circumstances in which the system may proceed to convert character strings into a "phonetic" code.

7.3.1.8.1.1 The `preparation=` parameter

This parameter expects the name of a `conversion` definition as a parameter value. The associated logical object does not have to exist at the very moment it is referred to but must exist by the time the system starts to execute the `soundex` definition. If the `conversion` definition does not exist at this very last possible moment the system refuses to execute the program and displays an error message. If you assigned the name of a logical object to this parameter the system will only be able to apply the `training` directive if the logical object already exists.

When you specify this parameter the system uses the `conversion` definition to pre-process the character strings concerned. *Only when this procedure is complete will the system encode the character string.*

7.3.1.8.1.2 The `after=` parameter

This parameter also expects as a value the name of a `conversion` definition. The only difference between this parameter and the previous one is that the system uses the `conversion` definition to which this parameter refers to post-process the coded value resulting from the `soundex` definition.

---

[8] "Phonetic" in this case should be taken with a pinch of salt. Within $\kappa\lambda\epsilon\iota\omega$, definitions of this type are used to create a framework within which the program can process any systems of rules whereby characters are combined to form groups. As a rule, it is at least as important to combine characters which are easily confused visually as it is to combine characters which are easily confused phonetically.

7.3.1.8.1.3 The `first=` parameter

This parameter expects one of the keywords `yes` or `no` as its parameter value.

- `no` instructs the system to encode the first letter of a word. If this letter was defined as a separator in `without=`, the system assigns it a coded value of `1`. This is the system's default setting.
- `yes` instructs the system not to encode the first letter of a word. This corresponds to most *Soundex* codes appearing in literature.

7.3.1.8.1.4 The `maximum=` parameter

This parameter expects a number as a value. This number is used to define the number of characters in the *Soundex* code. Experience to date has shown that 4 or 5 characters produce the best result. By default 4 is assumed.

7.3.1.8.1.5 The `without=` parameter

This parameter expects a constant as a value. Characters appearing in this constant are treated as separators during the coding operation. The system assigns them an internal coded value of `1`, which is only output if a separator appears as the first character of a word and this first character is also encoded (default and `first=no`).

7.3.1.8.1.6 The `form=` parameter

This parameter expects one of the two keywords `yes` or `no` as a value. It controls the Soundex algorithm's sensitivity to differences between upper-case and lower-case characters.

- `no` explicitly instructs the system to behave as it would be default. It is defined as follows:

  Every declaration issued for a lower-case or upper-case letter also applies to the alternative form of the letter, unless another declaration explicitly applies to this alternative form.

  ```
  part signs="bp"
  ```

  is thus interpreted as: `part signs="bBpP"`

  But the system interprets the two directives:

  ```
  part signs="bp"
  part signs="PS"
  ```

  (intended e.g. to take into account easily confused initials in an ornamental script) as follows:

  ```
  part signs="bBp"
  part signs="PSs"
  ```

- `yes` disables this mode of behaviour. In this case, the system distinguishes between upper-case and lower-case characters; if a letter is only defined in one form of spelling, it is ignored in the other.

7.3.1.8.2 The `part` directive

The general form of the `part` directive is:

SIGNs=<character string>	**(MUST be specified!)**
MORE=part | sign	**(default: part)**

This directive forms the core of the `soundex` definition. It is used to formulate the "phonetic" groups which define how character strings are encoded. You should specify one `part` directive for each "phonetic" group. The system assigns a coded value of `2` to the characters specified in the first group, a coded value of `3` to the characters in the second group, etc. etc.

7.3.1.8.2.1 The `signs=` parameter

This parameter expects an alphanumeric constant as a parameter value. These characters are all regarded as members of the same "phonetic" group and encoded with the same code number.

7.3.1.8.2.2 The `more=` parameter

This parameter expects one of the keywords `phonetic` or `character` as a parameter value. It is used to define the procedure to be followed where two characters in succession are identical or have the same coded value, but are not separated by another character (a separator).

- `phonetic` instructs the system to output a single code number whenever it encounters two characters with the same coded value which are not separated by a separator. This is the system's default setting.
- `character` instructs the system to suppress the second code number only if the two successive characters are identical.

7.3.1.8.3 The `training` directive

The `training` directive does not currently accept any parameters.

You can test the `soundex` definition by specifying this directive. After this directive *κλειω* expects lines containing one character string each. Each of these strings will immediately be processed by the specified definition. A line starting with an asterisk `''*''` ends this training mode.

7.3.1.9 The `skeleton` definition

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

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

Words are reduced to a framework of essential characters by means of a *skeletonising algorithm*. You must tell the system which characters to ignore during the skeletonising operation.

7.3.1.9.1 The `conversion` directive

The general form of the `conversion` directive is:

PREParation=<name> **(default: none)**
AFTEr=<name> **(default: none)**
MAXImum=<number> **(default: 20)**
WITHout=<character string> **(MUST be specified!)**

You use it to tell the system which characters to ignore and define how the character string should be pre- and post-processed. Since the actual skeletonising operation only produces useful results in the case of relatively standardised character strings, these pre-processing and post-processing stages, which should be implemented in their own separate `conversion` definitions, play a very important role.

7.3.1.9.1.1 The `preparation=` parameter

This parameter expects the name of a `conversion` definition as a parameter value. The associated logical object does not have to exist at the very moment it is referred to but must exist by the time the system starts to execute the `skeleton` definition. If the `conversion` definition does not yet exist at this very last possible moment the system refuses to execute the program and displays an error message. If you assigned the name of a logical object to this parameter the system will only be able to apply the `training` directive if the logical object already exists.

When you specify this parameter the system uses the `conversion` definition to pre-process the character strings concerned. *Only when this procedure is complete will the system skeletonise the character strings.*

7.3.1.9.1.2 The `after=` parameter

This parameter also uses the name of a `conversion` definition as a parameter value. The only difference between this parameter and the previous one is that the system uses the `conversion` definition to which it refers to post-process the framework of characters resulting from the `skeleton` definition.

7.3.1.9.1.3 The `maximum=` parameter

This parameter expects a number as a value. This number specifies the number of characters in the *skeleton*. You only need to define this parameter if the default setting of max. 20 characters does not suit your needs.

7.3.1.9.1.4 The `without=` parameter

This parameter expects a constant as a value. During the skeletonising operation the system ignores any characters which appear in this parameter. In this parameter you should list all characters – including special characters – which the system should ignore during the skeletonising operation. Only blank spaces are ignored by default.

7.3.1.9.2 The `training` directive

The `training` directive does not accept any parameters in the current version of $\kappa\lambda\epsilon\iota\omega$.

You can test the `skeleton` definition by specifying this directive. After this directive $\kappa\lambda\epsilon\iota\omega$ expects lines containing one character string each. Each of these strings will immediately be processed by the specified definition. A line starting with an asterisk `‘‘*’’` ends this training mode.

7.3.1.10 The `guth` definition

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

conversion
cycle **(MUST be specified at least once, can be specified more than once)**
training
exit **(MUST be the last directive in the series)**

The `guth` definition attempts to quantify the degree of similarity between two character strings in numerical terms.

To do this the system starts by comparing the two strings, beginning with the first character in each case. In conceptual terms, this means that the system places a pointer on the first character in each string. If the two characters are *not* identical the system performs the tests described in the `cycle` directives, relative to the position of the pointer in each string; if at least one of these tests is successful the system moves the two pointers one character to the right from the positions they have reached during the tests and initiates the next series of tests. If during any part of this operation *none* of the tests is successful, the system aborts the comparison operation and displays a degree of similarity of "zero" is displayed. The system calculates degrees of similarity by:

$$\frac{the\ sum\ of\ the\ weightings\ of\ successful\ tests}{100 * the\ number\ of\ characters\ in\ the\ longer\ character\ string}$$

Accordingly, a comparison of "`note`" with "`note`" gives a result of $\frac{400}{100*4} = 1.00$, a comparison of "`note`" with "`node`" gives a result of $\frac{300}{100*4} = 0.75$ and a comparison of "`note`" with "`no`" gives a result of $\frac{200}{100*4} = 0.50$, assuming that you retain the default weighting of 100.

7.3.1.10.1 The `conversion` directive

The general form of the `conversion` directive is:

MAXImum=<number>	**(default: 100)**
WITHout=<character string>	**(default: none)**
PREParation=<name>	**(default: none)**
PART=yes | no	**(default: no)**

This directive controls the way the system pre-processes words which are to be processed and the way the `guth` definition behaves in general.

7.3.1.10.1.1 The `maximum=` parameter

This parameter expects as its value a number defining the number of characters to be compared per word. You only need to specify this parameter if you do not wish to use the default setting of *100* characters.

7.3.1.10.1.2 The `without=` parameter

This parameter expects an alphanumeric constant as a parameter value. The characters contained in the constant are ignored during comparisons of both character strings.

7.3.1.10.1.3 The `preparation=` parameter

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

If you specify this parameter, the system pre-processes the character chains you wish to work with using the `conversion` definition. *Only when this procedure is complete will the character chains be compared.*

7.3.1.10.1.4 The `part=` parameter

This parameter expects one of the two keywords `yes` or `no` as a parameter value. It determines the way the system should behave in the event that one character string contains the other character string in its entirety.

- `no` instructs the system to calculate the extent to which they are identical in the usual way, even if one character string is entirely contained in the other. This is the system's default setting.
- `yes` instructs the system to treat both character strings as identical even if one of them is entirely contained in the other and even though there may be extra characters in one of them.

7.3.1.10.2 The `cycle` directive

The general form of the `cycle` directive is:

FIRST= + | - <number> **(default: 0)**
SECOnd= + | - <number> **(default: 0)**
NUMBer=<number> **(default: 100)**

You use this directive to define *a single one* of the successive tests which comprise the Guth comparison. To do this you define the positions of the characters to be compared in both character strings during this particular operation. You can specify this directive as often as you like. If you do not specify a `cycle` directive the two character strings are only checked for identical features after the system has pre-processed them using `conversion`.

7.3.1.10.2.1 The `first=` parameter

This parameter expects a positive or negative number as a value. It defines the position of the character to be compared in the first character string. If you do not define this parameter or assign the default value *0* to it, this character position is always defined by the current position of the pointer.

7.3.1.10.2.2 The `second=` parameter

This parameter defines the position of the character to be compared in the second character string in a given test, just as `first=` does for the first character string. Everything which applies to `first=` also applies to `second=`.

7.3.1.10.2.3 The `number=` parameter

This parameter expects a number between 1 and 100 as a parameter value. It determines the weighting factor which should be assigned to the successful completion of this test during the calculation of the resulting degree of similarity. By default the system assumes a weighting of 100.

7.3.1.10.3 The `training` directive

The `training` directive does not accept any parameters at the present time.

You can test the `conversion` definition by specifying this directive. After this directive $\kappa\lambda\epsilon\iota\omega$ expects lines containing one character string each. The character strings on two successive lines will immediately be compared by the specified definition. A line starting with an asterisk `''*''` ends this training mode.

7.3.1.11 The `substitution` definition

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

name **(MUST be specified)**
part
substitution
exit **(MUST be the last directive in the series)**

They are used to define *substitution paths*, i.e. access paths which the system uses if it cannot find an element or group specified by the user in the required location while it is processing a database.

Substitution paths are very general and can be defined independently of individual databases. If they are addressed – implicitly or explicitly – while the system is processing a particular database, they are updated and displayed in a manner which reflects the actual condition of the database. However, this means you can only check the reliability or unreliability of the names while they are actually in use.

A $\kappa\lambda\epsilon\iota\omega$ user who is not familiar with a database will be unable to distinguish between the name of a substitution path and the name of an element or group; the system interprets them according to the context of the command in which they appear.

All set definitions for $\kappa\lambda\epsilon\iota\omega$'s paths also apply in this case: an element which is replaced by a substitution definition may thus correspond, for example, to the set of all those entries which can be encountered at the end of a substitution path.

7.3.1.11.1 The `name` directive

This directive has the general form:

NAME=<Name> **(MUST be specified)**
TYPE=Part | Name **(default: Part)**

This directive defines the origin of the substitution path. It thus determines which group – or element, if the group is missing – should be replaced.

7.3.1.11.1.1 The `name=` parameter

The `name=` parameter expects a user-defined name as a value. When the system is following the substitution path it first attempts to find this group or element starting from your current position in the database. If it succeeds, no further steps are necessary.

7.3.1.11.1.2 The `type=` parameter

The `type=` parameter accepts the keywords `part` or `name` as values. Both of these can be shortened to their respective first character.

- `part` instructs the system to interpret `name=`'s value as the name of a group. This is the system's default setting.
- `name` instructs the system to interpret `name=`'s value as the name of an element.

7.3.1.11.2 The `part` directive

This directive has the general form:

NAME=<name>	**(default: none)**
PART=<name>	**(default: none)**
RELAtion=<name>	**(default: none)**
TARGet=<name>	**(default: none)**
SOURce=<number>	**(default: none)**
MORE=<number>	**(default: none)**
SUBStitution=<name>	**(default: none)**

It describes a path which continues in stages from the point at which the system failed to find the element or group which it was expecting to replace. The last stage of this path should lead to an element or group which replaces the missing group or element. This substitution is structurally transparent: in other words, any additional access operations are performed as if the replacement unit was located in the same position as the unit being replaced.

If while the system is processing this path it discovers that it cannot execute a step because one of the replacement elements or groups is missing, all parts of the system involved behave as if the data to be replaced was missing.

Within a path described by `part=`, the system also recognises the names of other substitution paths. It is therefore possible to address another substitution path from within the current one. However, since there is currently no way of protecting yourself completely from logical loops, the user is warned not to nest substitutions too deeply.

You can use all the parameters as often as you like.

7.3.1.11.2.1 The `name=` parameter

This parameter expects as a parameter value either the name of an element contained in the database or a dynamic path name, introduced by the `substitution` directive and activated earlier on (cf. section 7.3.1.11.3 below).

While the system is processing the substitution path, this step causes it to search for the element corresponding to this parameter's value.

7.3.1.11.2.2 The `part=` parameter

This parameter expects a parameter value consisting of the name of a group contained in the database or a dynamic path name, introduced by the `substitution` directive and activated earlier on (cf. section 7.3.1.11.3 below).

While the system is processing the substitution path, this step causes it to search for the group corresponding to this parameter's value.

7.3.1.11.2.3 The `relation=` parameter

This parameter expects a value consisting of the name of a `relation`-type element contained in the database.

While the system is processing the substitution path, this step causes it to search out the element corresponding to this parameter's value and then move to the group to which this element refers.

You can use this parameter to change over to another database, as enabled by the `bridge` mechanism described in section 10. To do this, you write the name of the database to which you want to make the transition in a pair of angular brackets at the end of the element assigned to `relation=`, for example as follows:

```
relation=transition<database2>
```

7.3.1.11.2.4 The `target=` parameter

This parameter expects a value consisting of the name of one of the elements contained in the database. You should only specify this parameter if is immediately preceded by the `relation` parameter.

While the system is processing the substitution path this step will cause it to check whether the element set up by `relation=` has the specified name. If this is not the case the system will not continue any further down this particular substitution path. Instead, it will examine the next alternative.

7.3.1.11.2.5 The `source=` parameter

This parameter expects a positive whole number as a value.

While the system is processing the path it will interpret this number as an instruction to move "upwards" from its current position in the database by <number> of logical levels, i.e. in the direction of the document level. So this parameter is equivalent to the `back[]` built-in function (cf. section 8.1.1.2.2.12), but, unlike the latter, it is database-independent.

7.3.1.11.2.6 The `more=` parameter

This parameter expects a positive whole number as a parameter value.

While the system is processing the path, it interprets this number as an instruction successively to bring in, from its current position, all groups not more than <number> of logical levels away from and dependent on the current group for further processing. So this parameter partly corresponds to the `total[]` built-in function (cf. section 8.1.1.2.2.10), but, unlike the latter, it is not database-dependent.

7.3.1.11.2.7 The `substitution=` parameter

This parameter expects as a value a dynamic path name (cf. section 7.3.1.11.3) defined in a preceding `substitution` directive.

This parameter instructs the system to activate this dynamic name.

See the section mentioned above for more information about dynamic path names.

7.3.1.11.3 The `substitution` directive

This directive has the general form:

NAME=<name> **(default: none)**
SOURce=<name> **(default: none)**

It is used to define dynamic path names.

A dynamic path name allows you to set up substitution paths depending on the contents of your database. The system writes the contents of an element in its place, which is converted when needed to the `text` data type. Next, the system attempts to find a group or element with this name.

The use of dynamic path names involves three stages:

1) The `substitution` directive's `source=` parameter is used to specify the name of the element, the contents of which should be used as the name of another element instead of the dynamic path name.
2) The `part` directive's `substitution=` parameter is used to instruct the system to activate the dynamic name, in other words to read the required element at the point it has reached along the substitution path.
3) Finally, by specifying the dynamic name as the value of a `name=` or `part=` parameter in the `part` directive, you cause the system actually to navigate to the element or group identified in this way.

7.3.1.11.3.1 The `name=` parameter

This parameter expects a user-defined name as a value. As from this directive, this name is used as a dynamic path name *within* the `substitution` definition.

7.3.1.11.3.2 The `source=` parameter

This parameter expects the name of an element in the database as a value. The contents of this element are transformed into names in the database once the system finally uses the dynamic path name.

7.3.1.12 The `codebook` definition

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

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

They are used to set up a translation table which makes it possible to translate any number of *alphanumeric identifiers* into any number of unrelated *numeric variables*. Additionally, explanatory remarks or *labels* referring to the individual numerical values of these variables can also be managed using these definitions.

The `part` directive is used to instruct the system which of these independent variables it should address and how; the `form` directive defines which value it should enter in this variable for which alphanumeric identifier and which label it should use for which of this variable's numerical values.

If no value is assigned to the variable derived from an alphanumeric identifier, the system uses a "missing value" instead, based on your installation's settings. If no label is defined for a numerical value, the system uses the character string "*(unknown)" instead.

A codebook generated explicitly by means of a definition, or implicitly by the `create` command, starts with two variables named `kleio` and `system`. Neither of these variable are created with labels. You can change or delete these variables like any others.

7.3.1.12.1 The `part` directive

This directive has the general form:

NAME=<name of variable>	**(default: existing name; initially: kleio)**
TYPE=Substitution | Insertion | Creation Destruction	**(default: Substitution)**
REPEat=Yes | No	**(default: existing condition)**
SELF=No | Yes	**(default: No)**

It determines which variables should be affected by the following `form` directives and which activities the system should perform using these variables.

7.3.1.12.1.1 The `name=` parameter

The `name=` parameter expects a user-defined name as a value. This name represents one of the variables in the codebook. The exact way it is interpreted depends on the value of the `type=` parameter.

7.3.1.12.1.2 The `type=` parameter

The `type=` parameter accepts a value consisting of one of the keywords `creation`, `insertion`, `substitution` or `destruction`, all of which can be shortened to their respective first character. These define how the system should interpret the following `form=` directives.

- `creation` instructs the system to introduce a new variable. You can only specify this parameter value if you also specify the `name=` parameter, to which you must assign the name of a variable which does yet exist in the codebook. This variable is set to "missing value" for all identifiers which are already present in the codebook. If it is followed by `form` directives, the system assumes that the newly introduced variable is in `type=insertion`.
- `insertion` instructs the system to treat the following `form` directives in such a way that the value of the `number=` parameter is assigned to each identifier specified in `text`. If you define an identifier in this mode to which a non-missing value has already been assigned, this value is not replaced: an error message is displayed instead. If you specify this mode without the `name=` parameter, the system continues to process the last variables you specified in the new mode; if you do assign a value to `name=`, it must involve a variable generated earlier on.
- `substitution` instructs the system to perform the same actions as `insertion`: in this mode, however, if you assign a new value to an identifier which has already been assigned a value, the new value overwrites the existing one instead of resulting in an error message.
- `destruction` instructs the system to delete a variable, a series of value assignments or a label. If you select this mode together with an assignment to the `name=` parameter, all traces of the variable identified in this way are removed from the codebook. If you do not specify a name in the `name=` parameter the system interprets the following `form` directives as an instruction to remove information about the last processed variable.

7.3.1.12.1.3 The `repeat=` parameter

This parameter accepts the keywords `yes` or `no` as values. Both of these can be shortened to their respective first character. It specifies whether the directives should be reproduced on the standard output medium.

- `yes` instructs the system to repeat them,
- `no` prevents it from doing so.

7.3.1.12.1.4 The `self=` parameter

This parameter accepts the keywords `yes` or `no` as values. Both of these can be shortened to their respective first character. It is required in order to process complex access operations in which the system uses the result of one codebook reference as an identifier with the help of which it can then access another codebook. In such cases, the "missing value", which is passed from one codebook to another as an access identifier, should always result in the return of the "missing value". However, in rare cases, this may result in inconsistencies, if the system has been explicitly instructed to transform "missing values" into another variable.

- `yes` activates the mode of behaviour described, which is also known as *self-referencing*,
- `no` suppresses it.

7.3.1.12.2 The `form` directive

This directive has the general form:

TEXT=<constant> (default: none)
NUMBer=<number> (default: none)
WRITe=<constant> (default: none)

It results in changes to information stored under an identifier or variable.

7.3.1.12.2.1 The `text=` parameter

This parameter expects a value consisting of a constant representing an identifier to be processed.

7.3.1.12.2.2 The `number=` parameter

This parameter expects a number as a value. It is assigned to the identifier specified in `text` for the currently active variable, or else has the label specified in `write=` assigned to it.

7.3.1.12.2.3 The `write=` parameter

This parameter expects a constant as a parameter value. It is interpreted as a label for the numerical value specified in `number=`.

7.3.1.12.2.4 Extent to which the parameters can be combined

The parameters in this directive can only be combined in very specific ways, dependent on the processing mode selected. The following combinations are possible: we include their meanings.

`form text="identifier";number=`*number*

Interpretation: *Assign* number *as the value for the current variable of the* identifier.
This combination is *illegal* in `type=destruction`.

`form text="identifier"`

Interpretation: *Remove the current variable's definition for* identifier.
This combination is *only* legal in `type=destruction`.

`form write="text";number=`*number*

Interpretation: *For the current variable, assign the* text *as a label for* number.
This combination is *illegal* in `type=destruction`.

`form number=number`

Interpretation: *Remove the current variable's label for* number.
This combination is *only* legal in `type=destruction`.

7.3.1.12.3 The `write` directive

This directive has the general form:

NUMBer=<number> **(default: none)**
TEXT=<constant> **(default: none)**

It is used to make changes to the information stored using a variable.

In the current version of $\kappa\lambda\epsilon\iota\omega$ this directive simply provides a subset of the options provided by `form`. At present it is only supported as a way of making additional options available.

7.3.1.12.3.1 The `number=` parameter

This parameter expects a number as a value. It is assigned the label defined in `text=`.

7.3.1.12.3.3 The `text=` parameter

This parameter expects a constant as a value. It is interpreted as a label for the numerical value specified in `number=`. In `type=destruction` you are not allowed to specify this parameter but you must define it in all other modes.