Vorbemerkung zur PDF-Ausgabe

Diese PDF-Datei wurde auf der Grundlage des Postscript-Dateien des Bandes 11 der *Halbgrauen Reihe zur historischen Fachinformatik* (HGR), Reihe B, erstellt. Es handelt sich um die Ausgabe der Systembeschreibung des Datenbankprogramms $\kappa\lambda\epsilon\iota\omega$ aus dem Jahr 1993. Damals waren die Versionen 5.1.1 (Windows) und 6.1.1 (UNIX) aktuell.

In der Zwischenzeit hat der Umfang des Programms deutlich zugenommen, ohne allerdings in einer Veröffentlichung vollständig dokumentiert worden zu sein. Die in diesem Buch enthaltene Programmbeschreibung ist zwar unvollständig, im Wesentlichen aber immer noch gültig.

Neben der Systembeschreibung existieren folgende Hilfsmittel, die sich mit dem Datenbankprogramm $\kappa\lambda\epsilon\iota\omega$ beschäftigen:

- ein Tutorial aus dem Jahr 1993 (HGR, A 23), das ebenfalls als PDF-Datei vorliegt;
- diverse Bände der Serie A der HGR, die sich mit unterschiedlichen Themenschwerpunkten auseinandersetzen (wobei in den älteren Bänden noch die lateinische Kommandosprache zum Einsatz kommt);
- diverse Bände der Serie B der HGR, die einzelne Softwarekomponenten beschreiben;
- die unvollständige und in Arbeit befindliche WWW-Dokumentation, die auch neue Features dokumentiert (`http://www.hki.uni-koeln.de/kleio/new/index.htm`);
- die *Benchmark* mit Testaufgaben, an deren Beispiel die Funktionsweise einzelner Bestandteile der Kommandosprache nachvollzogen werden kann (leider nicht immer zugänglich: `http://gilgamesch.hki.uni-koeln.de/develop/bench/index.htm`).

Mit der Einführung der Version 7 im Jahr 1995 erhielt das Datenbanksystem $\kappa\lambda\epsilon\iota\omega$ eine graphische Benutzeroberfläche, die u.a. die Menüführung bei der Volltextanalyse ersetzt hat. Der volle Funktionsumfang der Software ist aber immer noch nur über die Kommandosprache zugänglich. Gleichzeitig wurde das Konzept der logischen Umwelt für verteilte Datenbasen umgesetzt. Waren bis zu diesem Zeitpunkt die Definitionen eines logischen Objektes immer genau einer Datenbasis zugeordnet, gelten sie jetzt für alle Datenbasen, die sich innerhalb einer logischen Umwelt befinden.

Einen weiteren größeren Einschnitt stellte die Einführung der Version 8.1 dar. Die Regel, dass jede Definition innerhalb einer logischen Umwelt nur genau einmal vorkommen darf, wurde auf die Beschreibung der Datenbasen ausgedehnt. Jede Gruppe und jedes Element darf innerhalb einer logischen Umwelt nur einmal definiert werden.

Die skizzierten grundsätzlichen Veränderungen spielen im Regelfall erst bei der Verwendung von zwei oder mehr Datenbasen eine Rolle.

Diese PDF-Datei wurde für eine Lehrveranstaltung zur quellenorientierten Datenverarbeitung im Wintersemester 2005/06 an der Universität Trier erstellt.

Thomas Grotum
Universität Trier
`grotum@uni-trier.de`

Halbgraue Reihe
zur Historischen Fachinformatik

Herausgegeben von
Manfred Thaller
Max-Planck-Institut für Geschichte

Serie B: Softwarebeschreibungen

Band 11

Manfred Thaller

$\kappa\lambda\epsilon\iota\omega$
A Database System

Max-Planck-Institut für Geschichte
In Kommission bei
SCRIPTA MERCATURAE VERLAG

St. Katharinen, 1993

Print: Konrad Pachnicke, Göttingen
Cover design: Basta Werbeagentur, Göttingen
ISBN: 3-928134-91-4

Preface

κλειω is a software system which over the years has been developed at the Max-Planck-Institut für Geschichte in Göttingen, with support from many different research institutions and individuals. The most recent example for such a cooperative effort is the translation of the system into English, which has been organised by a consortium of British institutions. The University of Southampton, Queen Mary and Westfield College of the University of London, the British Academy and the Royal Historical Society all contributed to it. The history of this project is described in detail in the introduction to the set of manuals, which forms volume B 12 of this series.

κλειω has been known widely for being the only substantial software system, which had a Latin command language. This decision was originally taken in 1977, considered a practical joke at the time. Now the command language has been translated into English as well as the documentation. A concordance of both command languages forms appendix 'A' of this manual. For at least three years the Latin commands will still be supported: they shall be gradually replaced by the English ones, more recent parts of the system being available only by English commands. For the time being, this manual is not only a translation of the German version, therefore, but the most complete description of the system available.

The translation of the system and its documentation into English coincides with some significant changes in the *internal* structure of the system. While to the general user *κλειω* 5.1.1/6.1.1 seems to be almost identical to the older version 4, the most central structures of the system have been changed, to open new possibilities for the future. For the time being, this is more or less a potential of the system only, as great care has been taken, that no individual command which worked under version 4 has to be rewritten or rethought to be applied under version 5/6. Basically two major changes in the design philosophy have been made:

- *κλειω* 5/6 has been prepared to handle individual components of databases more independently than so far. Besides being convenient for many applications, this is a precondition for the possibility to use some of the tools of *κλειω* – like spelling standardisation or handling of medieval coinage – without preparing the data upon which these tools shall operate as a full fledged *κλειω* data base.
- Furthermore, the system can be "configured" now, to create, without modifying the software, versions which look radically different and are adapted to casual users of prepared data bases.

Both capabilities are almost hidden from the users at the moment: while the configuration system is introduced briefly in appendix 'B' of this manual, the new logic of administering the "logical environment" of data bases is not described in the printed documentation at all. We expect these two areas to provide a rapidly increasing number of additional tools, which will be described in additional release-related material, which is easier to update than the printed documentation.

- Finally: the new internal structure of the databases reduces in some cases their size quite spectacularly.

The current version of $\kappa\lambda\epsilon\iota\omega$ is referred to as version "5.1.1/6.1.1". Why two versions at the same time? Version 5.1.1 (popularly known as *low-tech* $\kappa\lambda\epsilon\iota\omega$) comprises the command language and the non-graphic menu interface. As such this version is machine independent and works independent of the platform on which $\kappa\lambda\epsilon\iota\omega$ is running. Version 6.1.1 (popularly known as *high-tech* $\kappa\lambda\epsilon\iota\omega$) is identical to version 5.1.1; restricted to a number of UNIX platforms at the moment, it provides, though, a first graphical user interface, which is so far predominantly geared to the handling and processing of image data.

The rather cumbersome designation was choosen, because version 5 will more or less be "frozen" at its current stage: it will continue to support the DOS operating system, no effort at any graphic components will be made, however. Version 6 will in the near future be ported to Windows NT; whether after that it is feasible to bring it from there to other Windows versions remains to be seen. All future development efforts will be dedicated to version 6.

Göttingen, August 1993

Table of Contents

1. Introduction

1.1 Purpose of this part of the documentation

This description of $\kappa\lambda\epsilon\iota\omega$ is intended to give users who are already familiar with the system a brief but comprehensive overview of currently available functions. This description is in fact the most up-to-date description of the system. Where this manual conflicts with other parts of the documentation – in particular descriptions of individual functions published in the "Halbgraue Reihe zur Historischen Fachinformatik" – the manual takes precedence.

1.2 The principles underlying command-based work

Users generally issue their instructions to software systems in one of two different ways: either by giving commands, formulated in a formal *command language*, or by choosing from a series of options displayed in the form of a *menu*.

$\kappa\lambda\epsilon\iota\omega$ gives the user the option of choosing a menu-driven interface. This is described in a separate manual (volume B10 in the "Halbgraue Reihe", and in part in the English tutorial volume), which is included in the system documentation for version 4.3.1 and subsequent versions. The menu interface allows the user to access $\kappa\lambda\epsilon\iota\omega$'s most important functions: in fact, certain parts of the system, such as the interactive full-text system, are only accessible via the menu interface because of the way they work. Generally speaking, however, $\kappa\lambda\epsilon\iota\omega$'s menu interface has two limitations. First, there is still a considerable amount of work to be done before users can access *all* the system's functions from menus; and second, many of the commands in the menu-driven version are deliberately somewhat limited in terms of flexibility, to keep the interface as clear and user-friendly as possible.

Thus the full power of the system is only available to users who are prepared to familiarise themselves with the command language. The main disadvantage of a command-based system is that users must learn the command language, which takes time and effort. The advantage of this kind of system, on the other hand, is that once users have mastered the command language they can formulate highly sophisticated queries for themselves, including queries which even the software's original designers may not have imagined in detail.

1.3 Some practical tips

The commands described below should be written in a command file. You can use any text editor or word processing program[1] to write a command file, after which you import it into $\kappa\lambda\epsilon\iota\omega$ for processing.

Regardless of the operating system you are using, you can import a command file into $\kappa\lambda\epsilon\iota\omega$ by entering the following command:

```
kleio [ <input file> | - [ <output file> ] ]
```

If you call up $\kappa\lambda\epsilon\iota\omega$ *without* first defining an `<input file>` or `<output file>`, the system will operate in interactive mode, i.e. from the menu-driven interface described in volume B10 of the "Halbgraue Reihe". If you activate the program in this mode unintentionally, you can always quit the program by pressing the ESCAPE key on your computer keyboard.

The `<input file> and <output file>` share the following characteristics: if the files are stored in the same directory (catalogue, file directory, etc.) as the one you are currrently in, every filename component should be joined to other filename components by a full stop (period). So for example, on a computer running MS-DOS you would write `file.ext`; on a VM/CMS system you would write `fn.ft.fm` – **not** `fn fm ft`. Similarly, whenever you refer to another file system (directory, catalogue, file directory etc.), you should use standard operating system conventions in order to specify the location of the files you want to access. Thus on an MS-DOS system, for example, you might enter `\mydirectory\mysubdirectory\myfile.ext`; whereas on a UNIX system you would enter `/mydirectory/mysubdirectory/myfile`.

If no `<input file>` exists yet, or if the local operating system will not allow you to read the input file's contents, $\kappa\lambda\epsilon\iota\omega$ will abort instead of executing any further commands.

If the `<output file>` already exists, $\kappa\lambda\epsilon\iota\omega$ will attempt to overwrite it. If the operating system prevents the program from doing this, or does not allow you to create a brand new output file, $\kappa\lambda\epsilon\iota\omega$ will abort instead of executing any further commands.

[1] However, you must take care to ensure that any command files you write are *ASCII files*, i.e. that your text editor or word processor does not insert any internal printing control codes into the text. $\kappa\lambda\epsilon\iota\omega$ interprets tabs as single spaces. $\kappa\lambda\epsilon\iota\omega$ supports most of the best-known word processing programs, including: (a) WordPerfect, Nota Bene, XY Write and Euro Script, any of which you may use as long as you do not attempt to set specific printing attributes. (b) Word: this program automatically places binary data at the beginning and end of every file. Methods of suppressing this feature vary depending on the version of Word and particular preset options you are using. You can distinguish an ASCII file by the fact that it does not contain any incomprehensible characters when you output the file to screen using the `type` command at the DOS prompt. (c) WordStar: careful! – this program uses its own internal code for individual characters. Save your file as a Non-Document File; you are unlikely to experience any problems with it unless it contains special foreign characters. If it does, you should convert it into a true ASCII file using WS Convert before attempting to import the file into $\kappa\lambda\epsilon\iota\omega$.

If you call up $\kappa\lambda\epsilon\iota\omega$ by entering:

```
kleio <input file>
```

$\kappa\lambda\epsilon\iota\omega$ will execute all commands contained in the `<input file>` and send the results to the default output device (almost always the screen). *If you use this syntax to call up $\kappa\lambda\epsilon\iota\omega$, the program will* not *make any attempt to wait until you are ready before sending each page of output to the screen or other output device.*

If you call up $\kappa\lambda\epsilon\iota\omega$ using the following syntax:

```
kleio <input file> <output file>
```

the program will execute the commands in the `<input file>` and send the results to the `<output file>`.

The following syntax:

```
kleio - <output file>
```

instructs $\kappa\lambda\epsilon\iota\omega$ to expect commands from the default input device – usually the keyboard – and send the processed results to the `<output file>`.

2. General

2.1 κλειω programs

Every κλειω program consists of one or more *tasks.* A task is a sequence of instructions which provide the system with all the information it needs to select data from one or more databases and present you with the result.

Tasks consist of *instructions* (generally referred to as *commands*), and *definitions.*

2.1.1 Instructions

An instruction always begins in the first column of a new line, and the first component is always a *command word.* κλειω only interprets the first four characters of the command word; the remaining characters can be omitted or written just as you like (even incorrectly!). The system does not distinguish between upper-case and lower-case characters in command words.

2.1.1.1 Command words

To date, the following command words have been implemented in the program: `case`, `category`, `confirm`, `continue`, `create`, `cumulate`, `delete`, `describe`, `exit`, `stop`, `image`, `index`, `item`, `read`, `text`, `negate`, `database`, `note`, `number`, `options`, `bridge`, `query`, `relation`, `catalogue`, `write`, `location`, `mapping`, `date` and `translation`.

2.1.1.2 Types of instruction

An instruction either changes the way κλειω works in general (*definition instruction*), or causes κλειω to perform a specific action (*executable instruction*).

Definition instructions available to date include: `category`, `image`, `text`, `note`, `number`, `options`, `relation`, `location` and `date`.

Executable instructions include: `case`, `confirm`, `continue`, `create`, `cumulate`, `delete`, `describe`, `exit`, `stop`, `index`, `item`, `read`, `negate`, `database`, `bridge`, `query`, `catalogue`, `write`, `mapping` and `translation`.

2.1.1.3 Specification

Immediately after the command word introducing a given instruction, you will find a *specification*, separated from the command word by one or more spaces.

If there is not enough room for the specification on one line of the input medium, it may continue over as many lines as necessary. In this case, you should leave the first column of every continuation line empty. If you want to lay out the various parts of the specification so it is even clearer, you can insert any number of extra spaces after this column (examples appear throughout this manual).

An empty line *always* terminates the preceding instruction.

2.1.1.3.1 Parameters

Every specification consists of a series of *parameters.*

A parameter consists of a *parameter name*, followed by an equals sign and a *parameter value.* If a parameter is followed by additional parameters within a single instruction, each parameter value should be separated from the name of the next parameter by a semicolon (";"). Parameters are identified by their first four characters; you may omit or replace the remaining characters as you wish. No distinction is made between upper-case and lower-case characters in parameter names. You may insert any number of blank spaces before and/or after equals signs and semicolons, or join up the characters, parameter names and parameter values in a continuous line, as you prefer.

As a rule, the order in which the parameters appear is irrelevant, unless a parameter depends for its interpretation on the existence of another parameter. Throughout this manual, we will draw your attention to these special cases in our descriptions of the various instructions.

2.1.1.3.1.1 Parameter values

Parameter values are either:

- constants,
- user-defined names (henceforth referred to simply as names),
- keywords,
- built-in functions or
- lists of the above values.

2.1.1.3.1.1.1 Constants

Constants are

- natural numbers, written without delimiters and including decimal points where required,
- individual characters or
- character strings, bounded by a pair of inverted commas (“ ’ ”) or quotation marks (“ " ”).

2.1.1.3.1.1.2 Names

Names always start with a letter and are between one and twelve characters long. Any additional characters are ignored.

Names may consist of combinations of the following characters:

- alphabetic characters,
- numeric characters,
- full stops (periods) (“.”), hyphens (“-”) and underscores (“_”).

The way “alphabetic characters” are defined depends entirely on your computer. As a rule of thumb, any character defined as part of a national alphabet on a given computer is a valid name character. If your computer uses non-standard extended ASCII characters, however – such as ø, Ø in Denmark, č, Č in Slavic countries, Cyrillic, Greek or Hebrew – $\kappa\lambda\epsilon\iota\omega$ may have to be *re-configured* to accept such characters in names. Please consult appendix B in this manual on the *$\kappa\lambda\epsilon\iota\omega$-configuration program*.

Normally, $\kappa\lambda\epsilon\iota\omega$ does not distinguish between alphabetic characters in upper or lower case, but users have the option of making such a distinction in certain categories of name.

2.1.1.3.1.1.3 Keywords

Keywords are determined by the system, depending to some extent on the given installation. Keyword names, in which no distinction is made between characters written in upper or lower case, can often be abbreviated to a single character, and can always be abbreviated to four characters. Any additional characters are simply ignored.

2.1.1.3.1.1.4 Built-in functions

A built-in function consists of a keyword, which can be abbreviated to four characters, followed by an “open square bracket – close square bracket” (“[...]”), containing a variable number of the built-in function’s arguments. Any spaces before or after the square brackets are ignored. Individual arguments inside the brackets are separated from one another by commas.

2.1.1.3.1.1.5 Lists of values

A list of values consists of a series of the other parameter values mentioned above, each value separated from the next one by a comma.

2.1.1.4 Standard format

Thus the standard format of a $\kappa\lambda\epsilon\iota\omega$ command is as follows:

$< command\ word >< space >< parameter\ name >=< parameter\ value >;$
$< parameter\ name >=< parameter\ value > ...$

2.1.2 Definitions

A definition is a construction in which a series of separate but interrelated statements are used to describe a logical object, which is assigned a name by the user. Henceforward, the object can be addressed by this name.

2.1.2.1 Commands which open definitions

Each definition is opened by a command, which obeys the same syntactical rules as any other instruction.

To date, the following commands have been defined for opening definitions: `item`, `database` and `bridge`.

Any command which opens a definition must include the `name=` parameter. This is used to assign a name to the logical object.

2.1.2.2 Definition directives

The command opening the definition is followed by a variable number of *directives*; you can define as many directives as you like.

A directive is a statement describing one aspect of a complex logical object. A directive's syntax is often identical to that of an instruction, but allows various logical objects to be defined in massively abbreviated form.

2.1.2.3 `exit` directive

The last directive in every definition consists of the command word `exit`, together with the `name=` parameter. The latter's parameter value is the same name as the one assigned to its logical twin in the command which opened the definition.

2.1.2.4 Standard format

Hence a $\kappa\lambda\epsilon\iota\omega$ definition is normally written as follows:

```
Command opening the definition
                    |
                    |
Defining directives
                    |
                    |
exit directive
```

2.2 Tasks

κλειω attempts to link instructions and definitions together in a sequence, creating an executable program in the process. *κλειω* continues to do this until it finds one of the three following situations:

1) the final instruction is identified by your computer's End of File symbol,
2) it identifies a `stop` command or
3) it identifies a `continue` command.

Any one of the above situations prompt the system to join together the instructions and definitions accumulated up to that point. The product of this fusion is a *κλειω* task – i.e., as we remarked at the beginning, the main subdivision (or "segment") of a *κλειω* program. Only by the end of a *κλειω* task must every single logical object used in any of the instructions have been defined. Hence the user may, if so desired, only declare a logical object after defining the instructions which use that object.

Unless *κλειω* finds an error, it will now start to execute the task.

2.2.1 Effective duration of instructions and definitions

If an End of Task is indicated by either of the first two situations described above, *κλειω* will automatically terminate its activity once it has finished preparing the result of this particular task. Thus the end of a *κλειω* program may be indicated either by the computer's own End of File symbol, or by *κλειω*'s `stop` command.

If, on the other hand, you use the `continue` command to indicate the end of the task, *κλειω* will "forget" all executable instructions in this particular task and prepare to process the next task.

However, the system does remain aware of all definitions describing logical objects which appear in this task until the *κλειω* program finally terminates; similarly, the effects of any definition instructions continue to apply until the end of the program.

3. How to structure tasks

Two commands are available for structuring a *κλειω* program as a series of tasks.

3.1 The `continue` command

The general form of a `continue` command is:

TARGet=<name of file on your computer>	**(default: none)**
OVERwrite=No | Yes	**(default: No)**

The `continue` command indicates that a defined task is to be performed, after which the system should expect to find further instructions.

3.1.1 The `target=` parameter

The value of the `target=` parameter normally consists of a constant corresponding to the name – or part of the name, identified more exactly by the appended parameter – of a file which has been correctly named according to the conventions of the operating system you are using.

All the results of the last task you formulated – i.e. without preceding commands – are written to the named file. If you specify an invalid filename (from the operating system's point of view), *κλειω* will *not* perform the task. The filename's full length (including path) is almost unlimited (4095 characters in *κλειω* as delivered). This means you can define long and complex paths in MS-DOS, UNIX, VMS and similar systems. However, we recommend that you only ever define *absolute* paths, as future versions of *κλειω* running on these systems will manage their own directories.

3.1.2 The `overwrite=` parameter

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

- `no` instructs the system not to perform the task if the output file you specified already exists (default value).
- `yes` instructs the system to overwrite any existing file which has the same name as the output file you specified.

3.1.3 Example

A correctly written `continue` command looks like this:

```
continue target="result.5";overwrite=yes
```

3.2 The `stop` command

The `stop` command instructs *κλειω* to perform a given task and then exit.

The command accepts the same parameters as the `continue` command and interprets them in exactly the same way.

3.2.1 Example

A correctly written `stop` command looks like this:

```
stop target="result.3"
```

4. Setting up a database

κλειω uses an internal format to manage data. This is specific to *κλειω* and *cannot* be processed by other programs. Before *κλειω* can begin to process data, it must convert it to this internal format.

This should be done by entering the data as a simple ASCII file using the text editor of your choice and then importing the ASCII file into *κλειω* for subsequent processing.

κλειω needs two things in order to convert data imported in this form:

1) data, collected according to rules composed by the user for his source from a stock of existing conventions, and
2) a description of these rules for the benefit of the system.

From these two constituents, *κλειω* generates a database which can then be processed using the system's other components. Physically, the database consists of a variable number of files, but the user addresses them all by a single common name.

4.1 Entries

The smallest unit of information in a database which *κλειω* can access is called an *entry*. Each entry consists of:

- the *core information* and the entry's
- *status*.

4.1.1 Core information

The core information in an entry consists of a character string composed of any or all of the characters which have not been reserved by the system for special purposes. The user can redefine the way these special characters are used; by default, the following characters in all input data are treated as reserved characters:

- dollar sign ("$"), henceforth: *data signal character 1*,
- slash ("/"), henceforth: *data signal character 2*,
- equals sign ("="), henceforth: *data signal character 3*,
- hash sign ("#"), henceforth: *data signal character 4*,
- percentage sign ("%"), henceforth: *data signal character 5*,
- open angled bracket ("<"), henceforth: *data signal character 6*,
- close angled bracket (">"), henceforth: *data signal character 7*,
- semicolon (";"), henceforth: *data signal character 8*,
- colon (":"), henceforth: *data signal character 9*,
- backslash ("\"), henceforth: *data signal character 10*.

Data signal character 9 can be used in the core information, like any other normal character; data signal character 10 is normally used to communicate with special forms of data description, but can also be used more generally to disable data signal characters 1 to 8 temporarily.

Thus for example, if the input data includes the character string `xxx \= yyy`, only the characters `xxx = yyy` will be imported into the database: the equals sign will not be interpreted as a data signal character. Otherwise, a data signal character 10 appearing in front of non-reserved characters is treated like any normal character for input purposes. Thus the character string `\typesetting command` would be accepted into the database as it stands, including the backslash.

Core information is assigned a particular *data type*. This determines what kinds of data may exist in the core information in question.

4.1.1.1 Rules for inputting specific data types

The following data types have been defined to date:

4.1.1.1.1 `text` data

Core information of this data type may consist of any non-reserved characters. The maximum length of an core information item of this data type depends on your implementation. The maximum length is 4095 characters in the system as delivered.

In addition, different rules apply to different kinds of elements, depending on the following circumstances:

- The logical object being used for the given element (cf. sections 7.3.1.2 and 5.4.7.1 for more about this).
- The general rules for managing `text` data specified in the special database (cf. also section 7.3.4.1).

4.1.1.1.2 `date` data

The system must be able to convert core information of this data type into a calendar date. Precisely which rules are applied depends on two conditions:

- The logical object being used for the given element (cf. sections 7.3.1.3 and 5.4.7.2 for more about this).
- The general rules for managing `date` data specified in the special database (cf. also section 7.3.4.2).

4.1.1.1.3 `number` data

The system must be able to convert core information of this data type into a number. Precisely which rules are applied depends on two conditions:

- The logical object being used for the given element (cf. sections 7.3.1.4 and 5.4.7.3 for more about this).
- The general rules for managing `number` data specified in the special database (cf. also section 7.3.4.3).

4.1.1.1.4 `category` data

Core information items of this data type always consist of separate characters, which are analysed independently of one another. The order in which these characters appear is not important, but no single character should appear more than once in any one item of core information.

In addition, different rules apply to different kinds of elements, depending on the following circumstances:

- The logical object being used for the given element (cf. sections 7.3.1.5 and 5.4.7.4 for more about this).
- The general rules for managing `category` data specified in the special database (cf. also section 7.3.4.4).

4.1.1.1.5 `relation` data

The system must be able to convert core information of this data type into "network identifiers" (network identification codes). This is done as follows:

a) To start with, each item of core information is split up into a first part consisting of a maximum of 12 characters, which must obey the rules for formulating names, and an (optional) second part, separated from the first by at least one space. The first of these two parts is the *network identifier*, the second is a *relationship comment*.

b) The relationship comment obeys the same rules as core information items of the `text` data type. It is stored in the database, but has no real importance as far as operations with the `relation` data type are concerned.

c) The network identifiers of all those elements which, in order to process the `relation` data type, use a logical object of the `relation` type with the same name, link together to form a *relational network* known by the same name.

d) When the network is being created, the system first establishes the location in the database of all identical network identifiers.

e) At a given moment, the given network is then *activated*. This means that wherever a network identifier appears in the database, a note is made of all the other locations in the database at which this network identifier has also been found. This results in a *subnetwork*. Using the command language's linguistic resources, as described in section 8.1.1.6, the user can move from any point in the database at which a network identifier belonging to a given subnetwork appears, directly to any other point which is also involved in this subnetwork.

f) Networks may be activated *implicitly* or *explicitly*. A subnetwork is activated implicitly if the system encounters a group which opens a document, or where the `part` directive has been defined with the `relation=` parameter applying to the network in question (cf. section 5.3.6.1). It is activated explicitly if the `relation` definition defining the network contains instructions to ignore any implicit activations, and an instruction to activate the network then appears at a later stage (cf. section 7.3.4.5).

In addition, different rules apply to different kinds of elements, depending on the following circumstances:

- The logical object being used for the given element (cf. sections 7.3.1.6 and 5.4.7.5 for more on this).
- The general rules for managing `relation` data specified in the special database (cf. also section 7.3.4.5).

4.1.1.1.6 `location` data

The system must be able to convert core information items of this data type into the names of topographical objects in a `location` definition. This means that an appropriate name must appear at the beginning of each core information item. After one or more spaces, you can append an explanatory text (which can be as long or short as you like).

The names of topographical objects are standard *κλειω* names.

The link between this data and the applicable `location` definition (cf. sections 7.3.4.6 and 5.4.7.6 for more information on this topic) is managed dynamically, however. This means that the core information item is accepted into the database even if the relevant `location` definition did not contain the topographical name at the time the database was created. The system does check to see whether a name is present, but any error messages which may appear are primarily diagnostic.

All *κλειω* commands, with the sole exception of the `mapping` command (cf. section 8.3.7), process core information items of this data type as `text` information. Only when the system is processing a `mapping` command is it actually necessary for all names used for topographical objects to have been defined in the relevant `location` definition.

4.1.1.1.7 `image` data

The system must be able to convert core information of this data type into the names of digitised images in an `image` definition. This means that an appropriate name must appear at the beginning of the core information item. You may append a variable-length explanatory text after one or more spaces.

The names of digitised images are valid file names on your computer, at the time of printing restricted to a length of twelve characters.

The link between this data and the applicable `image` definition (cf. sections 7.3.4.7 and 5.4.7.7 for more information on this topic) is managed dynamically. This means that the core information is accepted into the database even if the relevant `image` definition did not contain the name of a digitised image at the time the database was created. The system does check to see whether a name is present, but any error messages which may appear are primarily diagnostic.

All *κλειω* commands, with the sole exception of the `write` command (cf. section 8.3.7), process core information items of this data type as `text` information. Only when the system is processing a `write` command in interactive mode it is actually necessary for all names used for digitised images to have been defined in the relevant `image` definition.

4.1.2 Status

The status of an entry is indicated by the presence/absence of two *tagging characters* or *flags*: the exclamation mark ("!") and the question mark ("?"). The user can decide whether to retain these status characters as integral constituents of the core information during input, or whether to remove them from the core. The status of an entry is not affected by the number of times the two flags appear in the entry. Neither the order nor the position of the flags is significant.

4.1.3 Data type and entry preamble

An entry's data type is usually determined by the default data type which the user has defined for entries at this particular point in the database (known as the entry's *preferred data type*). But the user can also explicitly define an entry's data type using an *entry preamble*.

An entry preamble is written as an annotation appearing in front of the relevant entry, as follows:

$< data\ signal\ character\ 6 >< data\ type\ identifier >< data\ signal\ character\ 7 >$

The system recognises the following data type identifiers: `text`, `date`, `number`, `category`, `relation`, `location` and `image`.

Each of these identifiers can be written in abbreviated form, using the first letter of the identifier only. There is no need to make a distinction between upper and lower case.

You can leave as many spaces as you like to either side of the data signal characters.

4.1.4 Examples

We include some examples of correctly-written entries below:

```
This is a text
12.12.1678-24.3.1679
Blackmason?!
<Number>14guilders
<d> 3 kal mar 1412
```

4.2 Elements

Any number of entries can be combined to form an *element*.

An element is described in terms of the following specifications:

- *name*,
- *visibility*,
- *view* and
- *entries contained.*

4.2.1 Names of elements

An element's name is a user-defined name in the sense understood by $\kappa\lambda\epsilon\iota\omega$. By making the appropriate entry in an `options` instruction, you can specify whether the system should distinguish between upper-case and lower-case characters in element names (cf. section 7.1.2.1.1). By default, the system does not make any distinction.

An element's name is either derived from the element's position within the group containing it, or explicitly specified as a prefix in the input data, separated from the element by use of the data signal character 3, defaulting to the equals sign. Any spaces before or after each data signal character 3 are ignored.

As a convenience for data input, elements can have the *null string* as a name, that is, a line containing an element may immediately start with the equals sign. The special keyword `null` is used to access such elements in the command language.

4.2.2 Visibility of elements

An element's visibility is the product of its original visibility multiplied by the visibility of the group which contains it.

By default, every element has an original visibility of 1. This can be explicitly modified in the input data, by appending a *visibility clause* to the name with which the user explicitly prefixed the element. The visibility clause should precede the data signal character 3 which is the name's end delimiter. Hence it is *not* possible to assign an explicit value for original visibility to an element if the latter's name has been derived from its position within the group containing it.

4.2.2.1 Visibility clauses

A visibility clause is opened by data signal character 9. This is followed by:

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

Any other specifications included in the visibility clause will cause the element name next to which it is positioned to be ignored. In most cases, this means that the system will also ignore the element's contents.

4.2.3 How an element is viewed

The existence of an element can be restricted to a specific view: that is, the element is only selected for processing if this view is explicitly activated for a given processing task with the database in question. Furthermore an element can also be allowed to exist in a number of different views, and to have different names in each one.

If an element has not been allocated to a specific view it exists in all views. If an element has not been allocated a name for a specific view the usual name for that element is the name for any explicitly or implicitly defined views.

If an element is selected independently from the group in which it is contained, only the view assigned to the element is important. If we consider the following *κλειω* task:

```
options also=3
query name=example;part=:element
```

a specific instance of the `element` is only considered for processing, either if no view was assigned to the element on input, or if the view "3" was explicitly assigned to it on input. In this selection process the view of the group containing `element` is never considered.

In the task

```
options also=3
query name=example;part=group:element
```

on the other hand, `group` as well as `element` have either to exist in all views or have to have been explicitly assigned to view "3".

4.2.3.1 Elements in a single view

You can assign an element to a single view by prefixing its name with a level code for the view, separated by data signal character 3. A level code for a view consists of a whole number (integer) between one and an upper limit which depends on your particular implementation. Typical upper limits include: 8, 16, 32.

To assign "`element`" explicitly to view "3" you must enter:

```
.../3=element=content of the element/...
```

4.2.3.2 Elements in several views

You assign an element to several views by prefixing it with several names. The last name is separated from the element's contents, and each name is separated from the next name, by data signal character 3.

If you join the names together like this, the system assumes that the name that appears after the first data signal character 3 applies to alternate view "1", the name after the second data signal character 3 applies to alternate view "2", etc. etc. While the names are being processed in this way, two data signal character 3's appearing one after the other will cause the ordinal number of the view to be incremented by one, although you will not be able to address the element by the ordinal number corresponding to this "gap" in the series.

The following example would therefore assign "`Johansen`" to the element "`surname`" in view "1" and to the element "`patronymic`" in view "2", consider it not to exist in view "3" and assign it to the element "`spurious`" in view "4".

```
.../surname=patronymic==spurious=Johansen/...
```

The user can modify this behaviour by prefixing each separate name for the element with an ordinal number for the related view, separated from the name by a data signal character 3; in short, the same principle which applies to an element in a single view. If you start to prefix each alternate name for an element with an ordinal number, you must continue to do so: once you start to assign explicit ordinal numbers, you *cannot* revert to a system in which the ordinal numbers are implied by the sequential order of the appearance of names and data signal characters.

The following example would therefore assign "`Johansen`" to the element "`surname`" in view "`3`", to the element "`patronymic`" in view "`1`", and to the element "`spurious`" in view "`7`".

```
.../3=surname=1=patronymic=7=spurious=Johansen/...
```

You can explicitly assign a visibility factor to *every* element name, independently of the other names (i.e. other views) for the element, as described above in section 4.2.2.

When you assign an element to several views using various different names, it exists independently in each one. So if you use the system editor[1] to edit the contents of the element under the name applying to view "n", the element will continue to exist unchanged in all the other views under the names which apply to each of these views.

4.2.3.3 Restriction

κλειω currently does not allow you to define the different views for an individual element as components of several different groups. All views for one element must be contained in the same group.

4.2.4 Entries contained in an element

Entries contained in an element belong to one of the element's several *aspects*. The user can specify whether entries should be interpreted as a subordinate unit of the aspects or aspects interpreted as a subordinate unit of the entries. The second option means that each entry can have several aspects; the first (default) option means that each entry belongs to just one of the element's aspects (although each aspect can consist of many entries). Once you have decided which rule to apply, it applies to every element with this name, and so cannot be disabled temporarily from within the data. If various names have been assigned to a single element in various views, the system will only respect the hierarchical relationships defined for the first name; all subsequent names are treated as if the same hierarchical relationships applied to them as well.

4.2.4.1 Aspects

Every element – or every entry, if you have subordinated aspects to entries – begins in the element's *basic information* aspect. Every entry which the system encounters is assigned to this aspect until either data signal character 4 or data signal character 5 is encountered. If the system encounters data signal character 4, it regards all subsequent entries as components of the *comment* aspect, and if it encounters data signal character 5, it regards them as components of the *original text* aspect. Just because the system encounters one of these two data signal characters does not mean that it will not encounter the other at a later stage. In the event that entries have been subordinated to aspects, however, any data signal character encountered for the second time in a single element is simply ignored. Spaces preceding and/or following either of these two data signal characters are also ignored.

[1] A system editor is currently only available in the Version 6, not Version 5 of *κλειω*.

4.2.4.2 Joining up entries

If an element contains several entries, they are joined up by the separator you defined for that particular element. By default, this is data signal character 8. Any spaces appearing after this data signal character are always ignored. You can also specify whether the entry separators in a given element should be imported into the entry's core information, or be removed. If the user decides on the second (default) option, any spaces in front of the separator are also ignored.

4.2.5 Examples

Here are some examples of correctly written elements:

```
Munich
place=Munich
occupation=farmer; innkeeper#appears to held both jobs
    at the same time
occupation=innkeeper#1848%Red Lion;farmer#1871
place_of_origin=family_name=bremer
place_of_origin=family_name=12=patronym:?=bremer#garbled
    patronym?%bremer[sen]?
3=date=15.6.1423
```

4.3 Groups

Any number of elements can be combined to form a single group. A group is described by the following characteristics:

- *name*,
- *identification*,
- *visibility*,
- *view*,
- *relationships with other groups*,
- *ordinal number* and
- *elements contained.*

4.3.1 Names of groups

A group's name is a user-defined name in the sense understood by $\kappa\lambda\epsilon\iota\omega$. By making the appropriate entry in an `options` instruction, you can specify whether the system should distinguish between upper-case and lower-case characters in group names (cf. section 7.1.2.1.1). By default, the system does not make any distinction.

Every group begins on a new line, with the name of the group appearing in front of data signal character 1 (defaulting to the dollar sign).

As a convenience for data input, groups can have the *null string* as a name, that is, a line containing a group may immediately start with the dollar sign. The special keyword `null` is used to access such groups in the command language.

4.3.2 Identification

Every group in a database has an *identification*. This consists of a string of up to 12 characters, which does not contain any data signal characters or spaces.

Identifications are constructed as follows:

- If the group contains an element in which the `element` directive's `identification=` parameter has been set to `yes`, the first twelve characters of the first entry in this element are used as the group identification.
- If you are not using this parameter or have set it to `no`, but you are using the `part` directive's `sign=` parameter for the group, the system composes an identification consisting of the `sign=` parameter's value followed by a hyphen ("-") and the group's ordinal number (cf. section 4.3.5.5).
- If you are not using the `sign=` parameter either, the system composes an identification consisting of the first three characters of the group's name, a hyphen, and its ordinal number.

4.3.2.1 Addressability of an identification

You can address any group by its identification. In this case, a distinction is made between:

- *Relative addressability.* This allows you to select the group from the set of groups with the same name which are dependent from the same group, using conventional incremental access. (`Son ''x''` out of the five `sons` of one `father`.)
- *Absolute addressability.* This allows you to select the group directly with the help of the `sign[]` built-in function, resulting in the same access speed as when using a catalogue.

4.3.3 Visibility of groups

A group's visibility is the product of its original visibility multiplied by the visibility of the group which contains it.

By default, every group has an original visibility of 1. This can be explicitly modified in the input data, by appending a *visibility clause* to the group's name. The visibility clause has to precede the data signal character 3 or 1 which is the end delimiter of the group's name.

4.3.3.1 Visibility clauses

A visibility clause is opened by data signal character 9. This is followed by:

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

Any other specification included in the visibility clause causes the system to ignore the group name to which it relates. In most cases, this means that its contents are also ignored.

4.3.4 How a group is viewed

The existence of a group can be restricted to a specific view: that is, the group is only selected for processing, if this view is explicitly activated for a given processing task with the database in question. Furthermore a group can also be allowed to exist in a number of different views and to have different names in each one.

If a group has not been allocated to a specific view it exists in all views. If a group has not been allocated a name for a specific view the usual name for that group is the name for any explicitly or implicitly defined views.

If a group is selected independently from the group in which it is contained, only the view assigned to the group is important. If we consider the following $\kappa\lambda\epsilon\iota\omega$ task:

```
options also=3
query name=example;part=group
```

a specific instance of the `group` is only considered for processing, if either no view was assigned to the group on input, or if the view "3" was explicitly assigned to it on input. In this selection process the view of the group containing `group` is never considered.

In the task

```
options also=3
query name=example;part=group1/group2
```

on the other hand, `group1` as well as `group2` have either to exist in all views or have to have been explicitly assigned to view "3".

4.3.4.1 Groups in a single view

You can assign a group to a single view by prefixing its name with a level code for the view, separated by data signal character 3. A level code for a view consists of a whole number (integer) between one and an upper limit which depends on your particular implementation.. Typical upper limits include: 8, 16, 32.

To assign "`group`" explicitly to view "3" you must enter:

```
3=group$...
```

4.3.4.2 Groups in several views

You assign a group to several views by prefixing it with several names. The last name is separated from the group's contents by data signal character 1, and each name is separated from the next name by data signal character 2.

If you join the names together like this, the system assumes that the name that appears after the first data signal character 2 applies to alternate view "1", the name after the second character applies to alternate view "2", etc. etc. While the names are being processed in this way, two data signal character 2's appearing one after the other will cause the ordinal number of the view to be incremented by one, although you will not be able to address the group by the ordinal number corresponding to this "gap" in the series.

The following example would therefore assign "`...`" to the group "`person`" in view "1" and to the group "`witness`" in view "2", consider it not existing in view "3" and assign it to the group "`spurious`" in view "4".

```
person/witness//spurious$...
```

The user can modify this behaviour by prefixing each separate name for the group with an ordinal number for the related view, separated from the name by a data signal character 3; in short, the same principle which applies to a group in a single view. If you start to prefix each alternate name for a group with an ordinal number, you must continue to do so: once you start to assign explicit ordinal numbers, you *cannot* revert to a system in which the ordinal numbers are implied by the sequential order of the appearance of names and data signal characters.

The following example would therefore assign "`...`" to the group "`person`" in view "3", to the group "`witness`" in view "1", and to the group "`spurious`" in view "7".

```
3=person/1=witness/7=spurious$...
```

You can explicitly assign a visibility factor to every group name, independently of the group's other names (i.e. other views), as described above in section 4.3.3.

When you assign a group to several views using various different names, it exists independently in each. So if you use the system editor[2] to edit the contents of the group under the name applying to view "n", the group will continue to exist unchanged in all the other views under the names applying to each of these views.

[2] A system editor is currently only available in the Version 6, not Version 5 of $\kappa\lambda\epsilon\iota\omega$.

4.3.5 Relationships with other groups

4.3.5.1 Implicit association

Every group encountered in the data has a unique relationship with every other group contained in the database.This is achieved by establishing which of the following conditions is true for each group included in the input data:

- whether it was included in the `part=` parameter of the `part` directive defined for the last group encountered (*hence is logically subordinated to this group*),
- whether it was included in the `part=` parameter of the `part` directive belonging to the group which introduced the group to which the last-encountered group was logically subordinated (*logically equivalent to the last-encountered group*) or
- whether it was included in the `part=` parameter of the `part` directive which opened a group to which the last-encountered group was logically subordinated (*logically superior to the last-encountered group*).

If none of these three conditions applies, the system assumes it is unable to assign this group and therefore ignores it. An error message is displayed.

Unless the user makes other arrangements, the system assumes that groups within a parent group appear in exactly the same order in which they appeared in the `part=` parameter of the `part` directive which opened this parent group. The system ignores any groups appearing after one of the groups which followed them in the parameter and displays an error message.

4.3.5.2 Explicit association

By prefixing the group's name with an *association character*, the user can explicitly define the relationship between this group and the last-encountered group. The number of spaces between the association character and the group name has no significance.

Association characters were introduced for use in recursive data models, that is to say, in data models in which a group directly or indirectly contains itself. We would advise you *not* to use these characters in any other context.

The following association characters have been defined:

- Data signal character 3: if this data signal character appears in front of a group's name, the system attempts to assign the same logical status to the group as it did to the last-encountered group.
- Data signal character 6: if this data signal character appears in front of a group's name, the system attempts to assign a logical status to the group which is superior to that of the last-encountered group.
- Data signal character 7: if this data signal character appears in front of a group's name, the system attempts to assign a logical status to the group which is inferior to that of the last-encountered group.

If the system does not perform the actions requested by the association characters – because the name of the newly-encountered group was not included in the `part=` parameter of the `part` directive relating to this association – the system attempts to associate the groups in the order specified in the previous section.

4.3.5.3 Relationships in other views

The relationships between one group and the others in the database are calculated for each one of the group's names independently. Thus the system attempts to relate each name in a group to the first name of the last-encountered group, according to the rules described in the last two sections.

4.3.5.4 Relocating groups within the structure

In addition to the above options, you also have the option of arranging for a given group to relate to another group than the one immediately preceding it, in each of the views represented by its second and all subsequent names. This is done by defining a *relocation symbol* between the association character (if present) and the group's name. The different symbols are interpreted as follows:

- a minus sign ("-") means that in this view, the group should be subordinated to the group which logically precedes the group with which this relationship would normally be formed. So in the following example:
  ```
  father$father-1
  son$son-1-1
  father$father-2
  son/-$son-2-1
  ```
  in the first alternate view, `son-2-1` is regarded as the son of `father-1`.
- a plus sign ("+") means that in this view, the group should be subordinated to the group which logically follows the group with which this relationship would normally be formed. So in the following example:
  ```
  father$father-1
  son/+$son-1-1
  father$father-2
  son$son-2-1
  ```
  in the first alternate view, `son-1-1` is regarded as the son of `father-2`.

  If no name appears after the relocation symbol, the group's first name is used instead.

4.3.5.4.1 More dramatic relocations within the structure

Between the relocation symbol and the group's name, you can also insert a *by option*. This consists of an integer (whole number) positioned between data signal character 6 and data signal character 7. The number tells the system how many groups it should shift by. A relocation symbol without a by option is thus equivalent to a symbol with a by option of "1".

4.3.5.5 Ordinal number of a group

Every group is assigned an ordinal number within the group which contains it. This is a serial number which distinguishes the group from all other groups *of the same name* contained in that particular group. This ordinal number is incremented with no regard for visibility or view, and thus reflects the order within *all* groups of the same name contained in the parent group. This counting of a given group always starts again at 1 whenever the group containing it is encountered. (Hence the numbering of the "sons" of two "fathers" would begin at 1 in both cases.)

4.3.6 Elements held in groups

The first element held in a group is written after data signal character 1, which delimits the group's name. If the group holds more than one element, the remainder are appended in sequence, each element separated from the last by data signal character 2.

The names of the elements can be deduced from the order in which they appear. Names are assigned in conformity with the name specified in the `position=` parameter of the main group's `part` directive.

You can introduce every element in this list using the name prefixed to the element, followed by data signal character 3; every other element must be introduced in this way.

If you start to specify an element's name, you must continue to specify the names of all subsequent elements in the group.

4.3.7 Examples

The following groups are correctly written; the three dots in each case represent the elements held in the respective group.

```
person$
person$ ...
Head_of_household/son$ ...
Head_of_household/////son$ ...
Head_of_household/fellow occupant/8=son$ ...
5=entry:?$ ...
item/-<4>$ ...
item/<-<2>sum:!$ ...
=entry$ ...
```

4.4 Documents

A *document* is a group which is held in the database as such, i.e. no other group contains it. For all databases you have to assign the name of the *preferred document* to the `database` command's `first=` parameter. The names of additional documents within that database are introduced by `part` directives with the parameter `start=yes`.

The first group to follow a `read` command should always be a document.

If none of the elements in the group which is being treated as a document is introduced by a `element` directive with an `identification=` parameter defined with a value of `yes`, the system tacitly assumes that this parameter is the first element to appear in the group.

A document's identification is always absolutely addressable.

4.5 Dynamic fixed-field formats

4.5.1 Preliminary remarks

κλειω is primarily intended for processing data with which traditional software packages are, for one reason or another, unable to cope. Among other things, this means that the various input options are primarily intended to allow the user to process data consisting of different combinations of elements, all of widely varying lengths, as conveniently as possible.

By their very nature, such input techniques become less efficient and convenient when the data concerned appears in the following format:

Surname	first name	number	number	number	number	number	number	sum
	of the	of the	of the	of the	of the	of the	of the	of the
head of	the household	men	women	boys	girls	men	maid	people
						servants	servants	
Woehrer	Willhelm	1	1	1	-	-	-	3
Heyse	Johannes	2	3	2	1	1	1	10
Leyrer	Gernot	1	2	2	3	11	11	30

Currently, *κλειω* is not very good at processing this kind of data; in fact, data appearing in the this format – as a series of regular fields of unvarying length – is often mentioned in training courses as a good reason not to use *κλειω* to manage a particular set of data. This has changed up to a degree with version 5.1.1/6.1.1 of *κλειω*, where changes to the internal design reduced the overhead for highly regular data. Still: if you are handling data which look very much like a statistical data matrix, *κλειω* might not be the tool you are looking for.

There do exist quite valid reasons, though, why regular data have to be entered into *κλειω*. For example, when a data set is basically irregular in structure, contains parts, however, which represent small tables. For these purpose a family of input tools have been developed, which make entering such data much more convenient.

The options described on the following pages apply to the data format required to set up a database, which is why we are describing them in this chapter. However, they only affect an input convention, not the basic data structure described earlier; for this reason we have chosen to present them in a way which presupposes that the reader is reasonably familiar with the conventional form in which data is imported into *κλειω*. This is why we have also avoided making the otherwise strict distinction between our description of input conventions and the command language elements required to declare them.

4.5.2 Fixed formats

4.5.2.1 Input

Using the $\kappa\lambda\epsilon\iota\omega$ resources we have described to date, combined with the `part` directive

```
part name=h;
        position=surname,firstname,mennum,womennum,
          sonsnum,daughnum,servnum,maidnum,sum
```

you would have to format the data described above as follows in order to input it:

```
H$Woehrer/Willhelm/1/1/1////3
H$Heyse/Johannes/2/3/2/1/1/1/10
H$Leyrer/Gernot/1/2/2/3/11/11/30
```

This kind of input format is obviously a much less efficient way of entering data appearing in *this* form than the fixed formats used in conventional statistical software packages, which might look like this:

```
Woehrer  Willhelm  1110000003
Heyse    Johannes  2321010110
Leyrer   Gernot    1223111130
```

On the other hand, historical sources may also appear in this format, characterised by all the peculiarities in terms of additional information and inaccuracies which provided the inspiration for developing $\kappa\lambda\epsilon\iota\omega$'s own input conventions and data structures in the first place.

Hence we are looking for an input format which combines the advantages of both approaches.

The solution we have adopted lies in enabling $\kappa\lambda\epsilon\iota\omega$ to switch over – at any point in a list of elements assigned to a `position=` parameter – to a fixed format in the traditional sense. This means you can enter the data in our example as follows:

```
H$Woehrer/Willhelm/1110000003
H$Heyse/Johannes/2321010110
H$Leyrer/Gernot/1223111130
```

As $\kappa\lambda\epsilon\iota\omega$ ignores redundant spaces, the following input format (which may be easier to check) is equally suitable:

```
H$Woehrer  /Willhelm  /1110000003
H$Heyse    /Johannes  /2321010110
H$Leyrer   /Gernot    /1223111130
```

4.5.2.2 Structure description and the `read` command

To indicate to *κλειω* that it should switch over from its normal logic – "elements are separated from one another by data signal character 2 (slash)" – to behaviour based on the premise that "elements have a fixed length", you should assign all elements to the `position=` parameter as normal, but this time define a required length for each element after its name and a data signal character 9 (colon). The following `part` directive allows you to switch over to the fixed-field input convention:

```
part name=h;
        position=surname,firstname,mennum:1,womennum:1,
          sonsnum:1,daughnum:1,servnum:2,maidnum:2,sum:2
```

This option is totally independent of the data type used. However, experience suggests that the kind of fixed format described above is normally used for numeric data, so we have made special provisions for the latter. One of these is equivalent to an "implicit decimal point", with which users of statistical software (among others) will be familiar. Thus if you enter two length specifications after an element, separated from the element's name and each other by data signal character 9's (colons), the system assumes that there are two values involved which should be linked together by a full stop (period). Thus the following fragment of a `part` directive:

```
...  ;position= ...  ,numbers:2:2, ...
```

would cause the value

```
        2225
```

to be interpreted as

```
        numbers=22.25
```

By extension you may also use this principle to define triplets or dates, simply by appending a third length specification.

```
...  position= ...  ,date:2:2:4, ...  ,price:2:2:1, ...
```

would thus be the correct way of instructing the system to interpret

```
        02031786 ...  12102
```

as

```
        date=02.03.1786 ...  price=12.10.2
```

In many instances you may wish to distinguish between missing entries and entries which are known to have a numerical value of *zero*, even in data entered in the manner described above. This you can do with the help of the `read` command's `null=` parameter (cf. section 6.2.3.3). The parameter expects a single-character value; any fields which have been defined as fixed-format fields using the convention described above, and which are composed of this character (or several of these characters) alone, are regarded as non-existent. If we take the example of a `part` directive written thus:

```
    part name=h;
            position=surname,firstname,mennum:1,womennum:1,
              sonsnum:1,daughnum:1,servnum:2,maidnum:2,sum:2
```

the read command

```
read name=example;null=-
```

would cause the input data

```
H$Woehrer / Willhelm/111-----03
H$Heyse / Johannes/2321010110
H$Leyrer / Gernot/1223111129
```

to be imported as if it was identical to

```
H$Woehrer/Willhelm/1/1/1////3
H$Heyse/Johannes/2/3/2/1/1/1/10
H$Leyrer/Gernot/1/2/2/3/11/10/29
```

In the case of this second form of input data, you would of course use the "conventional" form of the part directive, thus:

```
    part name=h;
            position=surname,firstname,
              mennum,womennum,sonsnum,daughnum,servnum,maidnum,sum
```

If a list of fixed-length elements is declared, a series of assumptions come into play regarding the way exceptional cases should be treated.

If the first element in a sequence of elements with declared fixed lengths is missing, the system assumes that the entire list is missing.

If you use the part directive

```
    part name=example;
            position=name,firstname,meadows:3,fields:3,woods:3,occupation
```

the following data

```
    example$Fricke/Jobs//farmer
```

is correctly interpreted as

```
    example$name=Fricke/firstname=Jobs/occupation=farmer
```

If, in a list of fixed-length elements, the system encounters a data signal character 2 (slash) or identifies a new group before it has found sufficient data to be able to assign data to all the elements in the list, an error message is displayed.

So taking the example of the part directive above, the following line of data:

```
    example$Fricke/Jobs/200/farmer
```

would result in an error message.

If, in a list of fixed-length elements, the system encounters data after assigning data to all fixed-length elements immediately following, an error message is displayed: all data in front of the next data signal character 2 (slash) is ignored.

Thus if we take the example of the `part` directive above, the following line of data:

```
example$Fricke/Jobs/200150400200/farmer
```

would also result in an error message.

4.5.3 Dynamic formats and special cases

There is one problem associated with the fixed-field formats we have just described. In many cases they apply to 99% of the data concerned. However, their use precludes precisely that characteristic which makes κλειω unique; that is to say, the ability to represent rare special cases while remaining true to the source. In order to obviate this problem, κλειω uses very much more complex rules for fixed-length fields than those we have discussed so far. The enhancements presented in this section are all based on this definition: for this reason, there is no need explicitly to request any of the following options in the structure definition.

As an example, the following source – a list of wage payments:

Name	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Johannsen	12.2	12.2	12.2	12.2	12.2	12.2
Ulrich	13.1	13.1	11.1fine: 2.0	13.1	13.1	10.0
Hendrik	10.0	11.2	10.3	9.8	10.0	12.4

Using the `part` directive

```
part name=wage;position=name,
    monday,tuesday,wednesday,thursday,friday,saturday
```

we can store the reason for the in deduction in Ulrich's wages in the comment aspect and the amount of the deduction in the original text aspect, thus:

```
wage$Johannsen/12.2/12.2/12.2/12.2/12.2/12.2
wage$Ulrich/13.1/13.1/11.1#fine%2.0/13.1/13.1/10.0
wage$Hendrik/10.0/11.2/10.3/9.8/10.0/12.4
```

Obviously it would be more efficient to enter this kind of data in the format we have just described, using the `part` directive

```
part name=wage;position=name,
    monday:2:1,tuesday:2:1,wednesday:2:1,thursday:2:1,friday:2:1,
    saturday:2:1
```

and entering the data as follows:

```
wage$Johannsen        /122122122122122122
wage$Ulrich           /131131111131131100
wage$Hendrik          /100112103098100124
```

which means that the option of importing wage deductions has been lost.

4.5.3.1 Input options

There is an option available for dealing with such cases. It is based directly on the following underlying definition of κλειω's dynamic fixed-field formats:

If the first character in an element which has been given a fixed length in the structure definition is a comma (","), the element in question comprises all data between this opening comma and the next comma located in the data, regardless of the fixed length specified in the structure definition.

Between this delimiting pair of commas, you can enter the element using any of the options κλειω provides. In this case the data can be represented as follows:

```
wage$Johannsen       /122122122122122122
wage$Ulrich          /131131,111#fine%2.0,131131100
wage$Hendrik         /100112103098100124
```

You may find, in a small number of special cases, that a large number of successive elements from sources of the kind discussed above have the same value. (In our example, Johannsen's daily wage is always `12.2`, and Ulrich's is usually `13.1`.) Another tool is available for this kind of situation, but we advise you only to use it in exceptional circumstances. In normal cases this convention tends to make data substantially less clear and more difficult to check.

By adding an additional parameter, `more=` (cf. section 6.2.3.3.2), to the `read` command, you can declare a character which causes fixed-length elements to be repeated. More precisely:

If the first character in an element which has been given a fixed length in your structure definition is the same as the value of the `read` *command's* `more=` *parameter, the system attempts to find a number after this character and before a concluding data signal character 9 (colon). This number specifies how many of the subsequent fixed-length elements should be assigned the same value (i.e. the value specified immediately after the colon.)*

Thus by using the following `read` command:

```
read name=source;more=&
```

we could enter the data listed above as follows:

```
wage$Johannsen       /&6:122
wage$Ulrich          /&2:131,111#fine%2.0,&2:131100
wage$Hendrik         /100112103098100124
```

As a rule it is only advisable to use this convention if you can also make use of the following additional feature:

If there is no *number between the opening repeat character and the concluding colon, the element in question will be repeated until all subsequent fixed-length elements have been filled.*

Or to put it another way:

```
wage$Johannsen       /&:122
wage$Ulrich          /&2:131,111#fine%2.0,&2:131100
wage$Hendrik         /100112103098100124
```

These repetitions take place at element level: in other words it makes no difference whether or not the element which is to be repeated is precisely the same length as the element which is to be filled with the repeated data.

4.5.3.2 Extra features in the structure description

Unlike the options we have described so far, there are two additional problems which do necessitate additional entries in the structure description, i.e. in the `part` directive.

4.5.3.2.1 Comments to Lists as a whole

In some cases, you may find that a source contains information relating as a whole to a list of fixed-length fields, but only appearing very occasionally in the actual source. So if we turn once again to our earlier example

```
part name=example;
        position=name,firstname,meadows:3,fields:3,woods:3,occupation
```

this would be the case if, after the entries concerning the property values, a second author added the note that these values were assessed at a later date as having been over-estimated by a total of two hundred guilders. This information affects all three property values, so it would not make sense to enter it in one aspect of just one of the three elements. The obvious step to take is to enter the information as follows:

```
example$Fricke/Jobs/200150400-200#subsequent amendment/farmer
```

But according to the rules we described at the start, you do not have this option, because a list of fixed-length elements containing more data than can be assigned to the individual elements held in it is interpreted as an error.

For such eventualities, you have the option of terminating a list of fixed-length elements by an element with a "negative" – that is, unknown – length. If you specify this kind of element, for example in the `part` directive

```
part name=example;
        position=name,firstname,meadows:3,fields:3,woods:3,
        addition:-1,occupation
```

the following additional rules apply:

After all fixed-length elements have been processed, any additional data appearing before the next slash is assigned to an element of unknown length positioned at the end of the list.

Thus the data

```
example$Fricke/Jobs/200150400-200#subsequent amendment/farmer
```

would be interpreted as follows:

```
example$name=Fricke/firstname=Jobs/meadows=200/fields=150/woods=400/
        addition=-200#subsequent amendment/occupation=farmer
```

If no surplus information is left over after the system has processed a list of fixed-length elements, any adjoining element with an unknown length is simply ignored.

Hence

```
example$Fricke/Jobs/200150400/farmer
```

is thus interpreted in both cases as:

```
example$name=Fricke/firstname=Jobs/meadows=200/fields=150/woods=400/
        occupation=farmer
```

4.5.3.2.2 Dynamic formats with explicit labels

The whole mechanism described so far required, that the data being entered in a dynamic fixed-field format form part of a chain of `position=` elements. This can be very inconvenient, if sources only infrequently contain series of numeric data or if there are other important reasons why they should be entered at the end of the data constituting a group. As `position=` elements have to precede all elements which are entered with their names explicitly mentioned, the following would *not* be possible:

```
example$name=Fricke/firstname=Jobs/occupation=farmer/200150400
```

For sources where such input conventions are desirable, an additional mechanism has been created, which requires the use of an additional directive in a structure declaration, known as a `series` directive. (The formal syntax of that directive is given in section 5.6 of this manual.).

With this directive we have the possibility to assign a whole series of dynamic fixed format input elements to a collective name, by which the series as a whole can be addressed in the input data and in other directives of a structure declaration. In the example above this would amount to

```
example$name=Fricke/firstname=Jobs/occupation=farmer/list=200150400
```

where we want to express, that `list` is not an element, but an indication that what follows in the input data shall be interpreted as a series of the elements `meadows`, `fields` and `woods`.

This would be accomplished by the `series` directive:

```
series name=list;
        position=meadows:3,fields:3,woods:3
```

A name defined by a `series` directive may be used in a structure declaration wherever an element name may be used; the element `list` will in our example, however, be considered redundant, when a user attempts to address it, for example, by a `query` command, as the data following it are divided between the three elements which constitute the series.

In cases where there are data being related to the series as a whole, as in the example discussed in section 4.5.3.2.1 above, it will obviously be convenient to store these additional data in the element, which contains the series. This is accomplished by an additional parameter. With the `series` directive

```
series name=list;
        position=meadows:3,fields:3,woods:3;also=yes
```

the input data

```
example$name=Fricke/firstname=Jobs/occupation=farmer/
        list=200150400-200#subsequent amendment
```

will be stored as if the would have been entered as

```
example$name=Fricke/firstname=Jobs/occupation=farmer/meadows=200/
        fields=150/woods=400/list=-200#subsequent amendment
```

If the `also=` parameter had not been specified, the data after “400” would have been diagnosed as superfluous.

5. Structure definitions

5.1 General

The following sections (5.2 to 5.7) start by defining the commands available to users for creating simple *κλειω* databases. These simple databases fulfil the following conditions:

- they hold one and only one kind of document,
- every name used in the database always stands for a set of facts and
- different names always stand for different sets of facts.

All three assumptions can be violated: but to do so requires greater understanding of the logic underlying the system. Formal descriptions of the parameters required for these more advanced options are included below, but they are only discussed in depth in a later section, 5.8, with the help of detailed examples.

5.1.1 Function of a structure definition

Structure definitions

- define the logical structure of a database, together with the semantic properties of the identifiers used within it, and
- create all the files which *κλειω* needs in order to fill up your database (sporting the name of your choice) with a succession of data.

If the structure definition is successfully completed – i.e. no errors are found – a database has been created. It does not yet contain any data, but is otherwise ready to use. If the system discovers an error in a structure definition, the definition is marked as invalid, hence unsuitable for the efficient recording of data. If you want to record data in the database thus afflicted, you must first correct the invalid definition using the (not yet implemented) structure editor. Alternatively, you may if you wish amend the instructions used to define the definition, effectively overwriting the invalid structure by creating a new database.

5.1.2 General conventions

A structure definition consists of:

- an opening `database` command,
- as many `part` directives as you like, but never less than one,
- as many `element` directives as you like and
- an `exit` directive.

The `database` command is always the first element to appear in a structure definition, just as the `exit` directive is always the last. The order of the directives is unimportant, unless the way a given directive is interpreted depends on the result of the interpretation of a preceding directive.

Directives in structure definitions are formatted in exactly the same way as *κλειω* commands,

- thus they always begin in the first column of a new line,
- they can be continued over as many lines as you like, as long as the first column is left free, and
- they have a list of parameters as their specification.

5.2 The `database` command

The general form of the `database` command is:

NAME=<filename>	**(MUST be specified!)**
FIRSt=<document identifier>	**(MUST be specified!)**
TYPE=Permanent | Temporary	**(default: Permanent)**
OVERwrite=Yes | No	**(default: No)**
WRITe=Names | Parts | Structure | Generic	**(default: no report; can be specified several times over)**
IDENtification=No | Yes	**(default: No)**
MORE=No | Yes	**(default: No)**

This command declares the new database's general characteristics.

5.2.1 Defining structure designations

If all the parameters required to identify the structure are present κλειω instructs your computer's operating system to create the necessary number of physical files and then initialises the tables which describe these files.

The names for these physical files are derived from the names given by the user; their exact form is determined by the computer on which your version of the program is running.

5.2.1.1 The `name=` parameter

The `name=` parameter accepts a user-defined name as a value. No distinction is ever made between upper-case and lower-case characters. The various physical files which comprise the database are henceforth managed under this name.

5.2.1.2 The `first=` parameter

The `first=` parameter accepts the name of a group as a value. This name should also appear as the value of a `part` directive's `name=` parameter in the same structure definition.

5.2.2 Saving your structure definition

You normally make a structure definition in order to define a structure for a new database, to which you will later add a succession of data. By default the following applies:

- the structure is stored in physical files, which are preserved when you exit from κλειω, and
- κλειω refuses to declare a structure if there is any risk of destroying an existing structure which has the same name.

5.2.2.1 The `type=` parameter

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

- `permanent` explicitly requests that the database should always be ready for the user to access; this is the default setting.
- `temporary` instructs the system to delete the database once the task has been completed.

5.2.2.2 The `overwrite=` parameter

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

- `no` prevents *κλειω* from overwriting an existing database with the same name, thus explicitly instructing the system to behave as it would by default.
- `yes` instructs the system to overwrite (i.e. delete) an existing database with the same name. If no database with this name exists this parameter is simply ignored.

5.2.3 Controlling the scope of printed output

By default *κλειω* prints out the most important codes for the entire database you are creating once you have completed your structure definition. These include the valid data signal character set, document names, setup date, etc.

5.2.3.1 The `write=` parameter

The `write=` parameter accepts the keywords `names`, `parts`, `structure` or `generic` as values. All these keywords can be abbreviated to their respective first character. You can specify this parameter as often as you like, using a different keyword each time, in order to obtain any extra printed output you require. (That is, in the form `write=names;write=parts;...` *Not* as `write=names,parts,...`

- `names` instructs *κλειω* to print out a list of descriptions of all valid element names in the database,
- `parts` instructs *κλειω* to print out a list of descriptions of all valid group names in the database,
- `structure` instructs *κλειω* to print out a description of the structural relationships between all non-generic identifiers in the database, as well as giving details of any structural restrictions (minimum / maximum permitted frequency), and
- `generic` instructs *κλειω* to print out a list of descriptions of all valid names in the database for generic names (of elements *and* groups).

5.2.4 Identifying groups

By default every document in a database is labelled with an identifier which

- corresponds to the contents of the first entry in the group if the group forming the document does not include an element which has been explicitly designated as an identifier.
- must be unique, i.e. only appear once in the database. If you try to use a document identifier *or any other absolutely addressable group identifier* already contained in the database, the system makes a number of attempts – the exact number determined by the way your system has been installed – to generate a unique identifier by permutation. If this is unsuccessful, the system generates a standard identifier (consisting of identifying prefix, hyphen and the group's ordinal number) instead.

5.2.4.1 The `identification=` parameter

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

- `yes` means that if there is no element with the property of a group identifier in the group forming the document, the system should use the first element in the group for this purpose. Thus, `yes` explicitly instructs the system to behave as it would by default.
- `no` means that whenever the group forming the document does not contain an element with the characteristics of a group identifier, the system should use a standard group identifier (identifying prefix, hyphen, ordinal number) instead.

5.2.4.2 The `more=` parameter

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

- `no` means that every absolutely addressable identifier in the database must be unique. Thus, `no` explicitly instructs the database to behave as it would by default.
- `yes` means that absolutely addressable identifiers may appear more than once; in this case, *κλειω* itself is responsible for distinguishing between the separate instances in which a given identifier appears.

5.2.5 Examples

Below are some examples of correctly written `database` commands:

```
database name=source;first=doc
```

```
database name=book;first=chapter;
      write=structure;write=parts;
      overwrite=yes
```

5.3 The `part` directive

The general form of the `part` directive is:

NAME=<list of group names>	**(MUST be specified!)**
WRITe=<constant>	**(default: same value as name=)**
ORDEr=No | Yes	**(default: No)**
SEQUence=Yes | No	**(default: Yes)**
IDENtification=No | Yes	**(default: No, in the case of a document Yes)**
SIGN=<list of characters>	**(default: first three name characters)**
SOURce=<group name>	**(default: none)**
SUFFix=No | Yes	**(default: No)**
PREFix=No | Yes	**(default: No)**
POSItion=<list of element names>	**(default: none)**
ALSO=<list of element names>	**(default: none)**
GUARanteed=<list of element names>	**(default: none)**
READ=<list of element names>	**(default: none)**
PART=<list of group names>	**(default: none)**
ALWAys=<list of group names>	**(default: none)**
ONLY=<list of group names>	**(default: none)**
ARBItrary=<list of group names>	**(default: none)**
RELAtion=<list of definition names>	**(default: none)**
STARt=No | Yes	**(default: No)**
ALIAs=<group name>	**(default: none)**
USAGe=<definition name>	**(default: none)**
MORE=No | Yes	**(default: No)**
SUBStitution=<definition name>	**(default: none)**
CUMUlate=No | Yes	**(default: No)**

The `part` directive defines the properties of one or more groups and most importantly, determines how the groups are interrelated.

5.3.1 Defining group names

If more than one group is declared in a `part` directive, those groups will share the same properties.

5.3.1.1 The `name=` parameter

The `name=` parameter accepts a list of group names as a value. Every name which appears as the value of a `part` directive's `name=` parameter should also appear as part of the value of a `part=` parameter in at least one other `part` directive in the structure definition. The only exception to this rule is the group name which appears as the value of the `database` command's `first=` parameter, i.e. the name of the group comprising the document.

It makes no difference whether a group is mentioned for the first time in a `part=` or a `name=` parameter.

If you assign a name in a `name=` parameter to more than one `part` directive in a structure definition, it is imported into the database with the properties of the last `part` directive. In skilled hands, this procedure can be combined with `source=` parameters and generic group names in order to pass on group characteristics in very complex ways, but this is definitely *not* something which the beginner should try.

Group names are standard κλειω names; as for convenience during data input the possibility exists, though, to use the *null string* as a group name, some convention has to be provided to describe this as a syntactically correct name. This is done by the keyword `null`. If it is assigned to the `name=` parameter, the system assumes that this group will be represented during data input by lines where nothing is in front of the data signal character (i.e. the dollar sign) introducing the group. The same keyword can be used with any parameter, which needs to address such a group "without a name", for example the `part=` parameter of the `query` command.

5.3.1.2 The `write=` parameter

This parameter accepts a constant of any length as a value.

If you specify this parameter, because you intend to print the group's name κλειω displays the constant for all groups defined with a `name=` parameter. *In input data and when specifying paths, however, you must continue to specify the name declared in the* `name=` *parameter.*

If you do not specify this parameter the name displayed is the one you declared in the `name=` parameter.

The name used for display appears both in listings of results computed from this database and in all menus which are requested to list this group for selection or to enter data into an instance of it.

5.3.2 Defining the properties of a group

Before κλειω attempts to interpret any of the other parameters in a `part` directive, it first defines standard properties for every group mentioned in the `part` directive's `name=` parameter, according to the following rules:

- If the group's name can be linked to a generic group name, the group takes on the properties of this generic group;
- otherwise all the group's user-definable properties are set to the default specified in your particular installation.

5.3.2.1 The `source=` parameter

The `source=` parameter accepts a group name *which should already have appeared in a preceding* `part` *directive's* `name=` *parameter* as a value. It causes the characteristics of the group mentioned as the parameter value of the `source=` parameter to be assigned to all groups mentioned in the current `part` directive's `name=` parameter, before any of the remaining parameters are interpreted. If you specify this parameter the system does not check to see whether it can link one of the group names mentioned as a value for the `name=` parameter to a generic group name.

5.3.2.2 Defining a group identifier

By default, a group identifier is composed according to the following rules:

- If the group contains an element with the properties of a group identifier, the first twelve characters in this element's first entry – or as many characters as this entry holds up to the first blank space – are called in to serve as the group's identifier.
- otherwise, the system generates an identifier composed of the following: "identifying prefix, hyphen, group's ordinal number". By default the first three characters of the group's name are used as the identifying prefix.

By default each group's identifier is relatively addressable.

5.3.2.2.1 The `sign=` parameter

The `sign=` parameter accepts a character string of between one and three non-data signal characters as a value. These take the place of the identifying prefix which the system would otherwise assign to the group by default.

5.3.2.2.2 The `identification=` parameter

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

- `no` means that the group is relatively addressable via its identifier. The system is thus instructed to behave as it would by default.
- `yes` means that the group is absolutely addressable via its identifier.

5.3.3 Deciding which elements to include

By default the system makes no assumptions regarding the elements to be included in the group. However, any element name is valid if it appears in front of a data signal character 3 in the input data for this group. If the system encounters an element name in the input data which has not yet been included in the structure definition, it tacitly includes this name. If you defined generic element identifiers, these are checked for their applicability to the element name before the latter is accepted into the structure definition.

With respect to all the following parameters (which complement these default settings): it makes no difference whether the element names to which they refer were first mentioned in one of these parameters in a `part` directive or in an `element` directive's `name=` parameter. The effect is exactly the same in both cases.

By default when the system is saving elements it ignores the order in which they appeared in the input data. In each group elements are held in the order in which the element names were made known to *κλειω*.

5.3.3.1 The `position=` parameter

The `position=` parameter accepts a list of element names as a value. If you specify this parameter the system checks the first "n" elements with no element name encountered in data destined for a group against this list. "n" stands for the number of element names in the list.

The `position=` parameter can also be used to define *dynamic fixed-field formats*. To do this, a number, immediately preceded by data signal character 9 (a colon), is written after the name of the element in question, specifying the required field width. This feature is described in detail in section 4.5.

5.3.3.2 The `also=` parameter

The `also=` parameter accepts a list of element names as a value. It cites additional elements which may be included in the group. Because according to the existing rules every element may be included in any group, this parameter does not currently have any great influence. We nevertheless encourage users to include it, as useful preparation for impending system developments.

5.3.3.3 The `guaranteed=` parameter

The `guaranteed=` parameter accepts a list of element names as a value. It stipulates that the elements contained in the list must be held in each of the groups referred to in the `name=` parameter. If at least one of the quoted elements is missing from the input data for one of these groups, an error message is displayed.

If one of the elements referred to also appears in a `position=` parameter, it is considered to be recognisable by virtue of its position. Otherwise the appearance of an element in this parameter implies that it must have been mentioned in a `also=` parameter.

5.3.3.4 The `order=` parameter

The `order=` parameter accepts the keywords `yes` and `no` as values. Both keywords may be abbreviated to their respective initial character.

- `no` causes the system to ignore the order of the elements in the input data and store elements in each group in the order in which the elements were made known to *κλειω*; in other words it causes the system to behave as it would by default.
- `yes` causes the order of the elements in the input data to be preserved. In the case of groups which normally hold many different kinds of element, choosing this parameter may slightly reduce the speed with which the newly-created database is processed.

5.3.3.5 The `read=` parameter

If an input screen for the menu driven input system (cf. “Halbgraue Reihe”, volume B10, section 9) is derived from a group declared by a `part` directive, the input screen will include a separate field for each element referred to in `position=`, `also=` or `guaranteed=` parameters by default. If you activate this screen each of these fields will initially appear to be empty.

This last feature can be adjusted with the help of the `read=` parameter.

The parameter accepts a list of element names as a value. Whenever the input screen is activated it causes all the fields corresponding to these elements to retain the values displayed on the last occasion this same input screen was activated.

For the time being this only applies while the superior group remains the same.

The value of this parameter can be changed during menu driven input operations by the mechanism described in sections 9.3.3.3.4 and 9.3.3.3.5 of volume B10 in the “Halbgraue Reihe”.

5.3.4 Deciding which groups to include

By default the system assumes that a group does not hold any other groups.

If you use one or more of the following parameters to declare that a particular group does hold other groups, the latter are included in the containing group in the order in which they were made known to *κλειω*. The order of the groups in the input data is not taken into account.

It makes no difference whether a group's name is first used in one of the following parameters or in another `part` directive's `name=` parameter.

However, every group name used in one of the following parameters must also appear in the `name=` parameter of at least one other `part` directive.

If you wish you may bring in any name which is used as a value for the `name=` parameter to act as a value for one of the following parameters. In this case if two groups with the same name follow each other in the input data, the second group is always logically subordinated to the first group (with no need to use an assignment character).

Disregarding for a moment the properties we are about to introduce in the following sections, all the parameters discussed in sections 5.3.4.1 to 5.3.4.4 cause any subsequently quoted groups to be held as logical constituents of the group named in the `name=` parameter. Regardless of their actual order, they are always interpreted as if encountered in the order `part=`, `arbitrary=`, `only=` and `always=`. In order to avoid undesirable cumulative effects, we would urge the user to start by referring to *all* the groups which are supposed to be dependent on the group named in the `name=` parameter in the `part=` parameter, before describing their other characteristics in more detail using the other three parameters.

5.3.4.1 The `part=` parameter

The `part=` parameter accepts a list of group names as a value. It specifies that the groups quoted in it may be held in any of the groups cited in the `name=` parameter. The groups must appear in the input data in the same order as that specified in the `part=` parameter. Thus any group with a name which appeared as the first name in the list of group names should appear in the data prior to the group which is second on the list. Groups which flout this rule are ignored, and an error message is displayed.

5.3.4.2 The `always=` parameter

The `always=` parameter accepts a list of group names as a value. To each name you can append a positive whole number preceded by a data signal character 8. The number should appear before the comma separating the name from the next one on the list. If no such number is given, the system assumes a value of "1".

The parameter demands that whenever one of the groups named in `name=` appears, all groups named in `always=` must appear at least "n" times (where "n" is the number you specified). An error message is displayed if this is not the case.

The following is true of the relationship between `part=`, `arbitrary=` and `always=`:

- If a group mentioned in `always=` has already been mentioned in `part=` or `arbitrary=`, it acquires the additional property described, but its other properties do not change.
- If a group is mentioned in `always=` which is not mentioned in either of the other parameters, it is placed at the end of the list of groups defined by these two parameters. Thus groups introduced by `always=` should only ever appear in a fixed order.

5.3.4.3 The `only=` parameter

The `only=` parameter accepts a list of group names as a value. You can append a positive whole number to each name, after a data signal character 8 but before the comma separating the name from the next one on the list. If no number is given, the system assumes a value of "1".

The parameter demands that whenever one of the groups named in `name=` appears, all groups named in `only=` should appear not more than "n" times (where "n" is the number you specified). If a particular group appears more frequently, any group appearing more frequently in the input data is ignored and an error message displayed.

The following is true of the relationship between `part=`, `arbitrary=`, `always=` and `only=`:

- If a group mentioned in `only=` has already been mentioned in `part=`, `arbitrary=` or `always=`, it acquires the additional property described, but its other properties do not change.
- If a group is mentioned in `only=` which is not mentioned in any of the three other parameters, it is placed at the end of the list of groups defined by these three parameters. Thus groups introduced by `only=` should only ever appear in a fixed order.

5.3.4.4 The `arbitrary=` parameter

The `arbitrary=` parameter accepts a list of group names as a value. It causes the specified groups to appear in any order after each of the groups quoted in the `name=` parameter.

The following is true of its relationship to `part=`, `always=` and `only=`:

- If a group specified in `arbitrary=` has not yet been encountered in `part=`, it is placed at the end of the list defined by this parameter. All such groups may therefore appear in any order, but only after the groups defined by `part=`.
- If one of the groups specified in `arbitrary=` has already been defined in `part=`, an annotation is made to the effect that it may appear "in any order". This means that it may appear in any order with all other groups in the list of valid groups defined by the `part=` parameter which (a) appear immediately before or after it and have the same property *or* (b) appear before or after it and are linked to it by an unbroken chain of groups which have this same property. The properties themselves do not change simply because the groups are either mentioned or not mentioned in `always=` or `only=`.

5.3.4.5 The `sequence=` parameter

The `sequence=` parameter accepts the keywords `yes` and `no` as values. Both keywords may be abbreviated to their respective initial character.

- `no` causes the order of groups in the input data to be ignored, so that the groups are stored in the order in which they were made known to *κλειω*. This is the system's normal default behaviour.
- `yes` causes the order of groups in the input data to be preserved. In the case of groups which normally hold many other different groups, choosing this parameter may slow down the processing speed of the newly-created database.

5.3.5 Using generic group identifiers

Any group identifier mentioned in a `part` directive's `name=` parameter is regarded as part of the description of this group's logical structure. This means that κλειω assumes that the identifier actually appears in this form in the input data.

By contrast, a *generic group identifier* does *not* appear in the data, but acts as a model for all group identifiers with names which begin or end with the generic identifier. If a `part` directive does *not* include a `source=` parameter, the system determines whether a generic identifier exists which corresponds to the beginning or end of the name specified in the `name=` parameter.

5.3.5.1 The `prefix=` parameter

The `prefix=` parameter accepts the keywords `yes` and `no` as values. Both keywords may be abbreviated to their respective initial character.

- `no` indicates that the value of the `name=` parameter does not include a group identifier, thus explicitly prompts the system to behave as it would by default.
- `yes` indicates that the identifiers specified in the `name=` parameter are not included in the input data, but that the values of the `name=` parameter for all subsequent `part` directives should be checked to see whether they contain one of the values of the object directive's `name=` parameter as a prefix.

5.3.5.2 The `suffix=` parameter

The `suffix=` parameter accepts one of the keywords `yes` or `no` as a value. Both keywords may be abbreviated to their respective initial character.

- `no` indicates that the value of the `name=` parameter does not include a generic group identifier, and thus explicitly prompts the system to behave as it would by default.
- `yes` indicates that the identifiers specified in the `name=` parameter are not included in the input data, but that the values of the `name=` parameter for all subsequent `part` directives should be checked to see whether they contain one of the values of the object directive's `name=` parameter as a suffix.

5.3.6 Behaviour in networks

The following applies by default:

- If a group is also the beginning of a document, it activates the open subnetworks in all networks in which a `relation` definition (cf. section 7.3.1.6) – rather than the `part=no` parameter – has been used to declare that the network in question is not confined to one document.
- All other groups leave existing networks unchanged (see section 4.1.1.1.5 for an explanation of the terminology used).

5.3.6.1 The `relation=` parameter

This parameter expects a list of names of `relation`-class logical objects as a value. If you declare this parameter the groups quoted in the `name=` parameter activate all open subnetworks in the specified networks.

5.3.7 Defining complex structures

5.3.7.1 The `start=` parameter

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

- `no` establishes that the group in question is not opening a new document (the system makes this assumption by default).
- `yes` establishes that the group in question is opening a new document.

This parameter is discussed in detail in section 5.8.1.

5.3.7.2 The `alias=` parameter

This parameter expects the name of a group as a value. It establishes that the groups named in the `name=` parameter may also be addressed by this other identifier.

This parameter is discussed in detail in section 5.8.2.

5.3.7.3 The `usage=` parameter

This parameter expects the name of a logical object of the `connection` type as a value. It indicates that from this group onward, all group names appearing in the data should be checked to see whether they should be translated into other names before they are processed.

This parameter is discussed in detail in section 5.8.3.

5.3.7.4

This section has been dropped in an earlier version, the numbering is kept for consistency, though.

5.3.7.5 The `more=` parameter

This parameter expects one of the keywords `no` or `yes` as a value. It determines whether the `usage=` parameter's `connection` definitions, when activated in succession, should replace or complement one another.

This parameter is described in detail in section 5.8.3.3.

5.3.7.6 The `substitution=` parameter

This parameter expects the name of a logical object of the `connection` type as a value. It indicates that from this group onward all names of elements appearing in the data should be checked to see whether they should be translated into other names before they are processed.

This parameter is discussed in detail in section 5.8.3.3 (but first see section 5.8.3.4).

5.3.7.7 The `cumulate=` parameter

This parameter expects one of the keywords `no` or `yes` as a parameter value. It determines specifies whether the `substitution=` parameter's `connection` definitions, when activated in succession, should replace or complement one another.

This parameter is discussed in detail in section 5.8.3.3 (but first see section 5.8.3.4).

5.3.8 Examples

Below you will find some examples of correctly defined `part` directives. There is no need to split them up over a series of indented continuation lines, but experience has shown that this does make them easier to read.

```
part name=inheritance;
      position=surname,firstname,dob,occupation;
      also=income,tax liability;
      part=father,mother

part name=person;
      prefix=yes;
      position=famname,firstname,dob,occupation;
      part=father,mother
```

5.4 The `element` directive

The general form of the `element` directive is:

NAME=<list of element names>	**(MUST be specified!)**
WRITe=<constant>	**(default: same value as name=)**
TYPE=Text | Date | Number | Category | Relation Location | Image	**(default: Text)**
FIRSt=Text | Date | Number | Category | Relation Location | Image	**(default: Text)**
SECOnd=Text | Date | Number | Category | Relation Location | Image	**(default: Text)**
ORDEr=Simple | Multiple	**(default: Simple)**
SOURce=<element name>	**(default: system defaults)**
SUFFix=No | Yes	**(default: No)**
PREFix=No | Yes	**(default: No)**
IDENtification=No | Yes	**(default: No)**
CUMUlate=No | Yes	**(default: No)**
ONLY=Yes | No	**(default: Yes)**
TEXT=<name of definition>	**(default: standard** `text` **definition**
DATE=<name of definition>	**(default: standard** `date` **definition**
NUMBer=<name of definition>	**(default: standard** `number` **definition**
CATEgory=<name of definition>	**(default: standard** `category` **definition**
RELAtion=<name of definition>	**(default: standard** `relation` **definition**
LOCAtion=<name of definition>	**(default: standard** `location` **definition**
IMAGe=<name of definition>	**(default: standard** `image` **definition**
ALIAs=<element name>	**(default: none)**

This allows you to specify the properties of elements so that they are different from the properties assigned by default.

5.4.1 Defining element names

You can declare several elements using an `element` directive. All elements declared in the same `element` directive will share the same properties.

5.4.1.1 The `name=` parameter

The `name=` parameter accepts a list of group names as a value. Every name which appears as the value of a `name=` parameter in an `element` directive must also be used as part of the value of a `position=`, `also=` or `guaranteed=` parameter for at least one `part` directive.

Element names quoted in these parameters without appearing in an `element` directive are tacitly included with the installation's default values.

The particular context in which an element's name first appears is not important.

If the same name is mentioned in several `element` directives in a structure definition it is imported into the database with the properties of the last of these directives. In certain circumstances this procedure can be combined with `source=` parameters and generic element names as a way of passing on properties in very complex ways, but this is definitely *not* something for the beginner to try.

Element names are standard $\kappa\lambda\epsilon\iota\omega$ names; though for convenience during data input the possibility exists to use the *null string* as an element name. As such some convention has to be provided to describe this as a syntactically correct name. This is done by the keyword `null`. If it is assigned to the `name=` parameter, the system assumes that this element will be represented during data input by lines where nothing is in front of the data signal character (i.e. the equals sign) introducing the element. The same keyword can be used with any parameter, which needs to address such an element “without a name”, for example the `part=` parameter of the `query` command.

5.4.1.2 The `write=` parameter

This parameter accepts a constant of any length as a value.

If the parameter is specified, this constant is printed out for all elements the names of which occur after the `name=` parameter, if the element's name is supposed to be displayed. *In input data and when specifying paths, however, you must specify the names declared with the help of the* `name=` *parameter.*

If you do not specify this parameter the name displayed is the one you declared using the `name=` parameter.

The name used for display appears both in listings of results computed from this database and in all menus which are requested to list this element for selection or to enter data into an instance of it.

5.4.2 Defining an element's characteristics

Before any other parameters are interpreted, standard properties are defined for every element mentioned in the `name=` parameter of an `element` directive, according to the following rules:

- If the element's name can be linked to a generic element name, the element takes the properties of this generic element;
- otherwise all the element's user-definable properties are set to the default specified in your particular installation.

5.4.2.1 The `source=` parameter

The `source=` parameter accepts an element name *which should already have appeared in a preceding* `element` *directive's* `name=` *parameter* as a value. It causes the characteristics of the element mentioned as the parameter value of the `source=` parameter to be assigned to all elements mentioned in the `name=` parameter of the current `element` directive, before any of the remaining parameters are interpreted. If this parameter is specified the system does not check to see whether one of the element names mentioned as a value for the `name=` parameter can be linked to a generic element name.

5.4.3 Selecting the preferred data type

Every aspect of an element is assigned a preferred data type; this data type is installation-specific. In almost every installation imaginable, this data type will be the `text` type.

5.4.3.1 The `type=` parameter

The `type=` parameter accepts one of the keywords `text`, `date`, `number`, `category`, `relation`, `location` or `image` as a value; all these keywords may be abbreviated to their respective initial character. It causes the preferred data type of the basic information of the elements named in the `name=` parameter to become the named data type.

5.4.3.2 The `first=` parameter

The `first=` parameter accepts one of the keywords `text`, `date`, `number`, `category`, `relation`, `location` or `image` as a value: all these keywords may be abbreviated to their respective initial character. It causes the preferred data type of the "comment" aspects belonging to the elements named in the `name=` parameter to become the named data type.

5.4.3.3 The `second=` parameter

The `second=` parameter accepts one of the keywords `text`, `date`, `number`, `category`, `relation`, `location` or `image` as a value: all these keywords may be abbreviated to their respective initial character. It causes the preferred data type of the "original text" aspects of the elements named in the `name=` parameter to become the named data type.

5.4.4 Assigning aspects and entries

By default entries are subordinated to an element's aspects. In other words an aspect may have as many entries as you like, whereas each of an element's three aspects can only appear once.

5.4.4.1 The `order=` parameter

The `order=` parameter accepts one of the keywords `simple` or `multiple` as a value: both keywords may be abbreviated to their respective initial character.

- `simple` explicitly requests the system to behave as it would by default.
- `multiple` causes aspects to be subordinated to entries. This means that for *this element*, each new entry is initially regarded as a constituent of the basic information. This means that the "comment" and "original text" aspects both end at the first character marking the end of the current entry.

5.4.5 Using an element as a group identifier

5.4.5.1 The `identification=` parameter

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

- `no` means that the element's contents are not regarded as the identifier for the group which contains it. `no` thus causes the system to behave as it would by default.
- `yes` causes the first twelve characters of this element's first entry to become the identifier for the group which contains it.

5.4.5.2 The `only=` parameter

The `only=` parameter accepts one of the keywords `yes` or `no` as a value: Both of them may be abbreviated to their respective initial character. It implies the existence of an `identification=` parameter with the keyword `yes`.

- `no` causes the element's contents – regardless of whether it is being used to identify the group in which it is held – to be incorporated into the database, thus ordering the system to behave as it would by default.
- `yes` causes the element to be used *exclusively* as a group identifier. This means that input data are *not* imported into the database as separate elements. If an element of this type is input with more than twelve characters – e.g. with an appended comment – the surplus characters are lost, that is, not imported into the database.

5.4.5.3 The `cumulate=` parameter

The `cumulate=` parameter accepts one of the keywords `yes` or `no` as a value. Both keywords may be abbreviated to their respective initial character. It implies the existence of an `identification=` parameter with the keyword `yes` as its value.

- `no` causes the element's contents alone to be reworked as a group identifier, thus explicitly ordering the system to behave as it would by default.
- `yes` means that before the contents of the element in question are used to form the group identifier it is first supplemented by the group identifiers of the groups immediately superior to it. This is done by writing the group identifiers of these groups, linked by a hyphen ("-") in front of the element's contents. If the resulting identifier consists of more than twelve characters only the last twelve characters are preserved.

5.4.6 Using generic element names

Any element identifier mentioned in an `element` directive's `name=` parameter is regarded as part of the description of this database's logical structure. This means that *κλειω* assumes that the identifier actually appears in this form in the input data.

By contrast, a *generic element name* is one which does *not* appear in the data in this form, but which acts as a model for all element names which begin or end with value of the relevant `element` directive's `part=` parameter. Where an `element` directive does not include a `source=` parameter the system attempts to determine whether a generic element name exists which corresponds to the beginning or end of each of the names specified in the `name=` parameter.

5.4.6.1 The `prefix=` parameter

The `prefix=` parameter accepts one of the keywords `yes` or `no` as a value. Both these keywords may be abbreviated to their respective initial character.

- `no` indicates that the value of the `name=` parameter is a normal, non-generic element name, thus explicitly prompting the system to behave as it would by default.
- `yes` indicates that the identifiers specified in the `name=` parameter are not included in the input data in this form, but that the values of the `name=` parameter for all subsequent `element` directives should be checked to see whether they contain one of the values of the object directive's `name=` parameter at the beginnings of their names.

5.4.6.2 The `suffix=` parameter

The `suffix=` parameter accepts one of the keywords `yes` or `no` as a value; both keywords may be abbreviated to their respective initial character.

- `no` indicates that the value of the `name=` parameter is a normal, non-generic element name, thus explicitly prompting the system to behave as it would by default.
- `yes` indicates that the identifiers specified in the `name=` parameter are not included in the input data in this form, but that the values of the `name=` parameter for all subsequent `element` directives should be checked to see whether they contain one of the values of the object directive's `name=` parameter at the ends of their names.

5.4.7 Parameters for specific data types

A *κλειω* element can hold entries with any combination of data types. For every element, you can make statements about all the data types which have been implemented. *κλειω* will then use the data types which are most appropriate for the data type actually contained in the data.

Note: *for precisely this reason, specifying one of the following parameters does* not *obviate the need to specify a* `type=`, `first=` *or* `second=` *parameter.*

By default, the data are converted to each data type according to the rules of the standard object brought in for this purpose (cf. section 7.3.4 for more on this). You can change the settings for each data type using the following parameters.

5.4.7.1 The `text=` parameter

The parameter accepts the name of a `text` definition as a value (cf. section 7.3.1.1).

This logical object is then used to convert the contents of all `text` entries in elements of this name.

5.4.7.2 The `date=` parameter

This parameter accepts the name of a `date` definition as a value (cf. section 7.3.1.2).

This logical object is then used to convert the contents of all `date` entries in elements of this name.

5.4.7.3 The `number=` parameter

This parameter accepts the name of a `number` definition as a value (cf. section 7.3.1.3).

This logical object is then used to convert the contents of all `number` entries in elements of this name.

5.4.7.4 The `category=` parameter

This parameter accepts the name of a `category` definition as a value (cf. section 7.3.1.4).

This logical object is then used to convert the contents of all `category` entries in elements of this name.

5.4.7.5 The `relation=` parameter

This parameter accepts the name of a `relation` definition as a value (cf. section 7.3.1.5).

This logical object is then used to convert the contents of all `relation` entries in elements of this name.

5.4.7.6 The `location=` parameter

This parameter accepts the name of a `location` definition as a value (cf. section 7.3.1.13).

This logical object is then used to convert the contents of all `location` entries in elements of this name.

5.4.7.7 The `image=` parameter

This parameter accepts the name of an `image` definition as a value (cf. section 7.3.1.17).

This logical object is then used to convert the contents of all `image` entries in elements of this name.

5.4.8 Defining complex structures

5.4.8.1 The `alias=` parameter

This parameter expects an element's name as a value. It indicates that elements named in the `name=` parameter may also be addressed by this alternative identifier.

This parameter is discussed in detail in section 5.9.2.

5.4.9 Examples

Below you will find some examples of correctly defined `element` directives. There is no need to split them up into a series of indented continuation lines, but experience has shown that this format does make them easier to read.

```
element name=price;
      type=number;
      second=date;
      order=multiple

element name=date,time;
      suffix=yes;
      type=date
```

5.5 The `limit` directive

The general form of the `limit` directive is:

NAME=<group identifier>	**(MUST be specified!)**
TYPE=Stop | Continue	**(default: Stop)**

The directive allows you to redefine the symbols for the "end of input data; terminate *κλειω* program" or "end of input data; continue *κλειω* program" (cf. section 6.1).

5.5.1 The `name=` parameter

As a value, the `name=` parameter expects a name which could be used as the name of a group.

In this database, it can be used instead of the symbols specified in the `type=` parameter.

5.5.2 The `type=` parameter

The `type=` parameter expects one of the two keywords `stop` or `continue` as a value; both may be abbreviated to their respective initial character.

- If this parameter is not specified, or if you explicitly assign it a value of `stop`, the symbol which you assigned to the `name=` parameter in the newly-defined database is used in the same way the `stop$` symbol described above would otherwise be used. You may then use `stop` as the name of a group in this database.
- If you assign a value of `continue` to this parameter, the symbol assigned to the `name=` parameter in the newly-defined database is used in the same way the `continue$` symbol described above would otherwise be used. You may then use `continue` as the name of a group in this database.

5.6 The `series` directive

The general form of the `series` directive is:

NAME=<element name>	**(MUST be specified!)**
POSItion=<list of element names>	**(default: none)**
ALSO=No | Yes	**(default: No)**

The usage of this directive is discussed extensively in section 4.5.3.2.2 of this manual. It can be used to assign a series of elements to a common name, which may be used to position this series anywhere in the data being entered for a group.

5.6.1 The `name=` parameter

As a value, the `name=` parameter expects a valid name of an element. When this element is encountered in the input data, the data following this element will be divided among the elements defined by the `position=` parameter.

5.6.2 The `position=` parameter

The `position=` parameter accepts a list of element names as a value. If you specify this parameter the system checks the first "n" elements with no element name encountered in the data after the start of the element with the name which has been defined by the `name=` parameter against this list. "n" stands for the number of element names in the list.

The `position=` parameter is normally used to define *dynamic fixed-field formats*. To do this, a number, immediately preceded by data signal character 9 (a colon), is written after the name of the element in question, specifying the required field width. This feature is described in detail in section 4.5.

5.6.3 The `also=` parameter

The `also=` parameter accepts one of the keywords `yes` or `no` as a value; both keywords may be abbreviated to their respective initial character.

- `no` indicates that all data following after the start of the element specified with `name=` has to be assigned to the elements defined with the `position=` parameter. Additional data is ignored as erroneous. This is the default behaviour of the system.
- `yes` indicates, that all data which remains after the last element specified with `position=` has been processed shall be assigned to the element which has been specified with `name=`.

5.7 The `exit` directive

The general form of the `exit` directive is:

NAME=<filename> (MUST be specified!)

This directive informs the system that it has reached the end of the structure definition. It ensures that the definition as a whole is checked for accuracy and causes the information about the structure requested by the `database` instruction to be printed out. At the same time all currently local logical objects relating to the data types are stored in the newly-created database's permanent environment (cf. section 7.3 for more information on the concept of environment).

5.7.1 The `name=` parameter

The value of the `exit` directive's `name=` parameter consists of the same name as the preceding `database` instruction's `name=` parameter. It does not matter whether the two names are written in upper or lower case, but they must be identical in all other respects.

5.8 Example for sections 5.2 to 5.7

Below is an example of a complete structure definition for a database which is about to be set up. Our decision to start by introducing the document and then to introduce the groups contained in the document is entirely arbitrary: there is *no* need to emulate it. Please note that the `person` group in the following example does not have to be included in the data; however, in order to safeguard the computer's consistency checks, we have made it dependent on `taxpayer`.

```
database name=taxlist;first=taxpayer
element name=date;
      suffix=yes;
      type=date
part name=person;
      position=famname,firstname,dob,occupation,origin;
      also=dod,register
part name=taxpayer;
      source=person;
      part=payment,father,mother,person
part name=father,mother;
      source=person
part name=payment;
      position=date,amount
element name=amount;
      type=number
exit name=tax list
```

5.9 Information units and their names

To make it easier to understand the options for defining more complex data structures discussed below, we are going to take a rather more general look at the concept of database as defined in κλειω.

The majority of existing database systems start from the premise that users present their data in the form of tables which are linked together in various ways. Hence the process of creating a database normally consists of designing a form which can then be used to describe each separate line of such a table. In this kind of system, "data input" means that information is collected together from a variety of sources and then reshaped so that it will fit into the previously designed tables.

κλειω, in contrast, is designed to give users the resources to manage items of information in an *historical source* – that is, an information object which is the end result of a very specific process – in their proper context, just as they have been handed down. Thus the users' task does *not* consist of designing and filling out tables, but first, in establishing what kinds of information object are included in their source, and second, in creating a *description language* – by defining groups and elements – which they can use to describe the complex of information which is present in the source.

This effectively means that groups and elements – both of which we will henceforth refer to as information objects – have three completely different tasks to perform:

- First and most obviously, they define the particular input convention to be used for a particular source of input data. For example, the definition of the group `person` currently specifies whether the `firstname` should appear before or after the `surname`; it also establishes where this information should be entered in an input screen. The definition of `price` specifies the names of the currencies which may be used during the input process.
- Similarly, information object definitions specify how these objects should be presented either in printed output or on-screen or both: different texts will appear in a printout depending upon which logical object is linked to a `category` entry.
- Given these obvious implications of information object definitions, users migrating over to κλειω from other database systems frequently tend to overlook the fact that the real importance of the object definitions lies elsewhere. κλειω does support hierarchical relationships between individual parts of the database, because practical experience has shown that such relationships are suitable for processing many different kinds of historical source. However κλειω is by no means a hierarchical system; it is in fact a *semantic* one. The fact that you are using the element *price* in four different groups does not just mean that this data is entered in a particular way; rather, the fact that you are using the same element in these four positions as good as tells κλειω that: "Information objects exist which share the property of having a `price`". Hence, regardless of the structures you have defined in your database, you can automatically process any database which includes `price` as if it was a "table" of objects which have a price.

This variety of meanings associated with "defining information objects" gives rise to a problem, inasmuch as these three points of view contradict each other in some respects.

If, for example, you want to process a complex source extending over three centuries, it would clearly be desirable to process all persons as a group of the same name – simply in order to be able to process your database as a prosopographical catalogue. At the same time, the fact that your source may include information about these individual persons in a completely different order after a particular point in time may mean that it is desirable to use different groups for the same information object at different times. The same obviously applies to prices before and after a currency system is reformed.

For this reason *κλειω* provides you with a range of tools for drawing a sharper and clearer distinction between these three aspects of your database definition.

- The option of managing different types of document in a single database should allow you to manage sources which came into being or evolved together in a single database, even when they contain very different parts (for example, like a parish register which begins each year with a concise history of the village, followed by a list of the baptisms, marriages and funerals which took place in the same year).
- The principle of "aliasing" is used to access an objectively uniform information object which has been entered in various forms.
- The "disambiguation" of input symbols should allow you to convert superficially similar parts of a source into a number of structures with different contents.

5.9.1 Several types of document in one database

Any $\kappa\lambda\epsilon\iota\omega$ database can contain completely different documents. The `part` directive's `start` parameter is used to declare them. The parameter expects either `yes` or `no` (the default) as a value. If you specify a `start` parameter with a value of `yes` for a `part` directive, this group will have the property of opening a new document.

Thus in order to process a "mixed" parish register of the kind described in our previous example, we might use this parameter in the following structure definition (which would of course need to be completed):

```
database name=source;first=marriage
part name=chronicle;
        start=yes;
        position=id,text
part name=birth;
        start=yes;
        position=id,date;
        part=child,father,mother,godparent
part name=marriage;
        start=yes;
        position=id,date;
        part=bride,bridegroom,bestman
part name=bride,bridegroom;
        part=father,mother
part name=death;
        start=yes;
        position=id,date;
        part=deceased,witness
part name=deceased;
        part=father,mother,spouse
exit name=source
```

Thus as we have defined it, this database holds four completely different kinds of document. The `first=` parameter named `marriage` is the "preferred" document, because if $\kappa\lambda\epsilon\iota\omega$ does not encounter a `query` instruction, or encounters a `query` instruction without a `part=` parameter, it uses this parameter instead. The following $\kappa\lambda\epsilon\iota\omega$ task:

```
query name=source
write
stop
```

would thus be interpreted by the system as:

```
query name=source;part=marriage
write
stop
```

and cause it to print out all the `marriages` – but only the `marriages`, *not* the `chronicle` entries, `births` or `deaths`.

So in a certain sense, we would be justified in regarding the structure described above simply as a collection of four totally independent databases, combined in a single physical database for practical reasons.

However, this interpretation is only partially justified. If we set $\kappa\lambda\epsilon\iota\omega$ the task:

```
query name=source;part=:name
index part=:name;part=:firstname
stop
```

we would obtain a list of *every single* person – regardless of their group identifiers, and also regardless of whether they were mentioned in connection with a `birth`, `marriage` or `death`.

How closely the individual types of document are linked to one another depends entirely on the structure we have defined: if two documents contain the same kinds of group and element (such as the three last-named in our example), they will appear as a single, almost homogeneous database for the purposes of most evaluations: if they do not have a great deal in common (like the three last-named and the notes in the `chronicle`), there will be very few types of evaluation in which they can be addressed collectively. If they have no information in common at all, it will be quite impossible to address them collectively.

For this reason, we would strongly advise the user to allow for one element, at least at document level – for practical purposes, we would recommend the element which identifies the documents – to appear in *all* the different types of document. Because we did precisely this in the previous example, we can use the following task:

```
query name=source;part=:id
write
stop
```

to print out the entire `source`, sorted by individual document type.

5.9.2 "Aliasing": different symbols for the same set of facts

Let us take a database containing two groups in which the elements only partially coincides; e.g. `father` and `mother`, where a distinction is made between `familyname` and `maidenname` and certain persons whose `firstname` we do not know because of gaps in our data.

The following fragment of a structure definition describes this condition:

```
part name=father;
        position=familyname,firstname
part name=mother;
        position=maidenname,firstname
```

In this example, we have two situations in which we have different identifiers for what is essentially the same entity. In any event, fathers and mothers are always persons; family names and maiden names are both the same kind of personal identifier – even if they are used differently in a social context. So we could say that the identifier `father` (or `mother`) is simply another – more specialised – form of `person` identifier, or to put it another way: is an "alias" for the `person` identifier. The same is true of the way in which the identifiers `familyname`, `maidenname` and `surname` relate to one another.

The `alias=` parameter is used to reproduce this relationship, and can be included in `part` and `element` directives. In our example, we might use this parameter as follows:

```
part name=father;
        alias=person; position=familyname,firstname
part name=mother;
        alias=person; position=maidenname,firstname
element name=familyname;
        alias=surname
element name=maidenname;
        alias=surname
```

The contents of this fragment of a structure definition mean that the six terms in question are understood exactly as we have described them above: `father` and `mother` are another, more specialised way of expressing the underlying concept of `person`. `familyname` and `maidenname` are specialised variations of `surname`.

In practice, this means that the task

```
query part=person
write
stop
```

applied to our database, prints out `fathers` and `mothers` in order of appearance, without making any distinctions. However, the following task:

```
query part=mother
write
stop
```

causes the named group – `mother` – and only this group, to be printed out.

Similarly, the task:

```
query part=:surname
write
stop
```

causes all groups which include `familyname` or `maidenname` to be printed out, whereas the task

```
query part=:maidenname
write
stop
```

simply causes groups containing `maidenname` to be printed out.

You can of course use this option in combination with the other new parameters. To test your understanding of the `start=` and `alias=` parameters try and establish why, if you use the following fragment of structure definition, the task

```
query name=source
write
stop
```

will cause all four types of document to print out.

```
database name=source;first=document
part name=chronicle;
        start=yes;
        alias=document;
        position=id,text
part name=birth;
        start=yes;
        alias=document;
        position=id,date;
        part=child,father,mother,godparent
part name=marriage;
        start=yes;
        alias=document;
        part=bride,bridegroom,bestman
part name=bride,bridegroom;
        part=father,mother
part name=death;
        start=yes;
        alias=document;
        position=id,date;
        part=deceased,witness
part name=deceased;
        part=father,mother,spouse
exit name=source
```

5.9.3 "Disambiguation": same symbols for different sets of facts

In many instances, we encounter sources containing lists which are superficially similar but in which the individual entries have been set up in different ways. This is true for example, of inventories, which frequently take the form of a "list of items"; so consist of an opening group describing a particular kind of possession (property, provisions, clothing ...), followed by very concise groups, each describing one of these objects. For practical reasons, these groups are deliberately designed in a uniform way, so that the data can be entered as swiftly and smoothly as possible: it is after all possible to describe all three types of object mentioned above by means of very similar structures – $< number >< identifier >< value >$, for example. Obviously such a basic structure can easily be converted into a $\kappa\lambda\epsilon\iota\omega$ structure[1]:

```
...
part name=p;write="property";
    position=totalvalue;
    part=null
part name=null;
    position=number,identifier,value
part name=f;write="provisions";
    position=totalvalue;
    part=null
part name=c;write="clothing";
    position=totalvalue;
    part=null
element name=number;type=number
element name=value;type=number;number=money
...
```

Or, perhaps even more important for practical work in a project, an instruction like this:

for each of the lists of properties, provisions, clothing etc., begin a new line with one of the key letters P, F, C ... *followed by a dollar sign and the calculated total value. This is followed by each object on the list, on its own line, beginning with a dollar sign. After this, the quantity or size of each object is specified, followed by a slash and the object's identifier, followed by another slash and the value of the object).*

is also quite easy to follow.

[1] In the following examples, we will continue to reproduce only those parts of the structure definition which are required in order to understand this section. In order to test the following examples, you should therefore complete the definitions of the logical objects.

It results in something like this (for example):

```
...
i$2400fl.
$2acre/meadow/300fl.
$20acre/woods/1500fl.
$3acre/field/600fl.
v$27fl.
$2hundredweight/grain/8fl. 12x.
$1hundredweight/oats/3fl. 7x.
...
k$35fl.
$4/shirts/2fl. 6x.
$1/skirt/4fl. 8x.
...
```

This list looks as if it would be very easy to type in. Unfortunately, it is hiding an awkward problem. Properties have an area, food supplies have weight and clothing is expressed as a number of items: so in reality, we should define the element `number` differently in each of these lists, because a codebook which managed every unit of measurement would very soon result in chaos.

So in fact the following model would be more suitable:

```
...
part name=p;write="property";
    position=totalvalue;
    part=p-item
part name=p-item;
    position=area,identifier,value
part name=f;write="provisions";
    position=totalvalue;
    part=f-item
part name=f-item;
    position=weight,identifier,value
part name=c;write="clothing";
    position=totalvalue;
    part=c-item
part name=c-item;
    position=number,identifier,value
element name=number;type=number
element name=area;type=number;number=area
element name=weight;type=number;number=weight
element name=value;type=number;number=money
...
```

From the point of view of the person writing this structural statement, the difference is marginal. However, the input data for three lists of this kind would be rather more complicated:

```
...
i$2400fl.
p-item$2acre/meadow/300fl.
p-item$20acre/woods/1500fl.
p-item$3acre/field/600fl.
f$27fl.
f-item$2hundredweight/grain/8fl.  12x.
f-item$1hundredweight/oats/3fl.  7x.
...
c$35fl.
c-item$4/shirts/2fl.  6x.
c-item$1/skirt/4fl.  8x.
...
```

and if you had twenty or thirty different lists, like the ones you find in typical inventory databases, it would be significantly more difficult to process the data structure.

5.9.3.1 The `usage=` parameter

So ideally we are looking for an option which will allow us to use the data as it first appeared, while simultaneously mapping it onto the structure we demonstrated later, in such a way that *κλειω* knows that a line beginning with a dollar sign should be linked to different terms, depending on whether it follows a line beginning with `p$` or a line beginning with `c$`. The `part` directive's `usage=` parameter is used to tell *κλειω* that when it encounters a particular group, it should process the following – or some of the following – groups with a different name than the one which was actually encountered in the data. This parameter defines the name of a logical object, which specifies the name to be used instead of the name which is encountered when a particular group is encountered.

```
...
part name=p;write="property";
    position=totalvalue;
    part=p-item;
    usage=plist
part name=p-item;
    position=area,identifier,value
part name=f;write="provisions";
    position=totalvalue;
    part=f-item;
    usage=flist
part name=f-item;
    position=weight,identifier,value
part name=c;write="clothing";
    position=total value;
    part=c-item;
    usage=clist
part name=c-item;
    position=number,identifier,value
element name=number;type=number
element name=area;type=number;number=area
element name=weight;type=number;number=weight
element name=value;type=number;number=money
...
```

5.9.3.2 `connection` definitions

The logical object in this case is a `connection`-class object. These definitions are described in more detail in section 7.3.1.16. They allow you to replace a name encountered in the input data depending on the name's position in the input job stream. Without wishing to repeat the description provided in the section mentioned, we would like to emphasize that the following definition is required if you want all lines beginning with a dollar sign to be interpreted as `p-item$`'s from a given point in the input data:

```
item name=plist;usage=connection
connection read=null;name=p-item
exit name=plist
```

In order to resolve our problem – "what we are looking for is a $\kappa\lambda\epsilon\iota\omega$ structure definition which will allow us to process the data which appeared in the first example as if they had been entered in the format illustrated in the second example" – we do in fact need to include the following statements in our structure definition:

```
item name=plist;usage=connection
connection read=null;name=p-item
exit name=plist
item name=flist;usage=connection
connection read=null;name=f-item
exit name=flist
item name=clist;usage=connection
connection read=null;name=c-item
exit name=clist
...
part name=p;write="property"; position=totalvalue; part=p-item;
    usage=plist
part name=p-item;
    position=area,identifier,value
part name=f;write="provision"; position=totalvalue; part=f-item;
    usage=flist
part name=f-item;
    position=weight,identifier,value
part name=c;write="clothing"; position=totalvalue;
    part=c-item;
    usage=clist
part name=c-item;
    position=number,identifier,value
element name=number;type=number
element name=area;type=number;number=area
element name=weight;type=number;number=weight
element name=value;type=number;number=money
...
```

5.9.3.3 The `more=` parameter

For reasons of efficiency, normally only one `connection` definition is active at any one time. That is to say, when the system encounters a group in the input data in which the `usage=` parameter has been used, it replaces the last-activated definition with the `connection` definition assigned to this parameter.

This does result in problems in very deeply nested data structures if, as from a given group, additional replacement statements become necessary, while at the same time the system must continue to implement statements which have already been activated. You can cause the system to behave in this way by declaring the `more=yes` parameter. If two simultaneously activated `connection` definitions contain different instructions for the same name encountered in the input data, the definition last activated always takes precedence.

If the system encounters a group on the same logical level as the one on which the second `connection` definition was previously activated, only the latter is cancelled in the first instance. The `connection` definition which was previously activated by a superior group remains in force on its own, until a group is encountered on the same logical level as this superior group.

As soon as you set the `more=yes` parameter, you can activate an unlimited number of `connection` definitions simultaneously.

5.9.3.4 Closing remarks

Such a `connection` definition can of course include any number of directives. So in theory it would be possible to input two totally different structures using the same group identifiers and to specify a `connection` definition at document level which, depending on the type of document concerned, would replace the identifiers you entered with completely new ones. However, we would advise you to use this feature sparingly, otherwise you may find that it becomes very difficult to establish which structure is actually in use.

Please note: the `usage=` parameter remains in force until *the next group on the same logical level* is encountered. If, in our example, you had *not* introduced a `usage=` parameter for `part name=c`, lines beginning with `i$` would have been interpreted as they appear in the data: that is, *not* according to the nature of the last-encountered group for which a `usage=` parameter had been defined.

On rare occasions, it may also be desirable to apply a similar principle at the level of element names, so that for instance a `value=` label appearing after `i$` is converted into `area=`. To do this, we follow exactly the same procedure. In this case, the `part` directive parameter we need is called `substitution=`; it also expects the name of a `connection` definition as its parameter value. If we experience the same problem with elements as we did with groups (as discussed in section 5.8.3.3), we can resort to an analogous solution, based on the `cumulate=yes` parameter.

6. Converting raw data into a database

Data are always read into a *κλειω* data base from an ASCII file. Such ASCII files can either be prepared by the menu driven input facilities described in volume B10 of the "Halbgraue Reihe" or with any text editor capable of storing pure ASCII files.

6.1 General

You use the `read` command to convert input data into a database. As soon as the system identifies this command, it automatically switches into another input mode. For this reason the system does not interpret the first line with an empty first column as the end of the command, but the first line containing data signal character 1.

If you have issued the `read` command, and it has been diagnosed as error-free, the system expects you to enter data immediately afterwards. Consequently the system will misinterpret any commands issued or definitions made after a `read` command.

`read` expects input data to start with a new document. If this is not the case, the system skips over a given number of other groups (the exact number depends on your particular installation) and then aborts the *κλειω* program if it is still unable to find a document.

A `read` task is terminated by one of three possible situations:

- If the system comes to the end of the data after the `read` command, it exits from the database and returns control to the operating system.
- If the system encounters a

  ```
  stop$
  ```

  group in the input data after the `read` command, it exits from the database at this point, ignores any other data in the currently active input file and returns control to the operating system.
- If the system encounters a

  ```
  continue$
  ```

  group in the input data after the `read` command, it exits from the database at this point and awaits any further instructions from the currently active input device. The user may, for example, wish to redefine a logical object in order to use a different currency conversion rate for a different part of the source, and then instruct the system to continue to create the database by entering another `read` command. Or the user may decide to move straight on to an analysis routine for the database which has been created.

Instead of using the `stop` or `continue` symbols you may also use the various resources described in section 5.5.

6.2 The `read` command

The general form of the `read` command is:

NAME= <filename>	**(MUST be specified!)**
REPEat= No | Yes	**(default: No)**
LINEs= <number>	**(default: value of the same parameter in the currently active Options)**
WRITe= Names | Parts | Structure | Generic	**(default: no report; can be specified several times over)**
CONTinuation= Clio | Null | <special character>	**(default: Clio)**
CUMUlate=<special character>	**(default: none)**
SUBStitution=<special character>	**(default: none)**
MORE=<special character>	**(default: none)**
NULL=<special character>	**(default: none)**
FORM=No | Yes	**(default: No)**

6.2.1 Referring to a database

Every `read` task must refer explicitly to an existing database. This database may be the result of a preceding structure definition (hence empty) or an earlier `read` task.

6.2.1.1 The `name=` parameter

The `name=` parameter expects the name of an existing database as a value.

6.2.2 Controlling the scope of the printed output

By default, while *κλειω* is converting the data into its own format, it only prints out the processed data if this appears to be necessary in order to illustrate an error. It also prints out a brief set of statistics on the total number of information units contained in the database, once the conversion procedure is complete.

6.2.2.1 The `repeat=` parameter

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

- `no` instructs the system not to print out the processed data, i.e. to behave as it would by default.
- `yes` instructs the system to print out the processed data, with no space lines between each line of data.

6.2.2.2 The `lines=` parameter

The `lines=` parameter expects a number between zero and four as a value. This parameter tells the system how many space lines to insert between every two lines of the imported and printed input data. If you specify this parameter, the currently active value of the `options` command's `lines=` parameter is restored once this particular `read` task has been completed.

6.2.2.3 The `write=` parameter

The `write=` parameter accepts the keywords `names`, `parts`, `structure` or `generic` as values, all of which can be abbreviated to their respective first character. You can specify this parameter as often as you like, using a different keyword every time. (This should be in the format `write=names;write=parts` *not* `write=names,parts`). This allows you to define any extra printed output with considerable precision.

- `names` instructs the system to print out a list of descriptions of all names in the database which would be valid element names,
- `parts` instructs the system to print out a list of descriptions of all valid names of groups in the database,
- `structure` instructs the system to print out a description of the structural relationships between all non-generic identifiers within the database, and at the same time to specify any structural restrictions (minimum / maximum permitted frequencies) and
- `generic` instructs the system to print out a list of descriptions of all names in the database which would be valid generic names (for elements (*and* groups).

6.2.3 Defining input control characters

$\kappa\lambda\epsilon\iota\omega$ processes the imported data as a set of information fields structured by data signal characters. Within these fields individual characters are assigned a meaning according to the data type involved.

You can modify this behaviour so that individual characters are assigned a meaning which does not depend on data type and which controls the way the imported data is converted before it is broken down into separate information units and analysed in greater depth.

6.2.3.1 The `continue=` parameter

The `continue=` parameter accepts the keywords `clio`, `null` or any other individual character which is not a letter or a number.

- `clio` instructs the system to ignore blank spaces at the end of every line of input and to join the next line to the previous one automatically in such a way that the last character on the first line (which is not an empty space) appears immediately in front of the first character on the second line. This is how the system behaves by default.
- `null` instructs the system to ignore blank spaces at the end of every line of input and to join the first character on the second line (which is not an empty space) to the last character on the first line after inserting a single blank space between the two characters.
- If you define a special character (a *line continuation character*), you can instruct $\kappa\lambda\epsilon\iota\omega$ to check whether the last character at the end of every line is the same as this special character. If it is, the system removes this special character and any preceding spaces, and the character appearing immediately in front of it is linked directly to the first character on the next line. If it is not, the system behaves as it would if you had specified the keyword `null`.

6.2.3.2 One way to simplify input

In order to process a list-based source laid out as follows:
Hubert Müller, 35 years old, a carpenter from Neuberg
Crescentia, his wife, 33 years old
Their son Hubert, a 14-year-old carpenter's apprentice
Their daughter Anna, 11 years old
Their daughter Walpurga, 8 years old
Friedrich Hübner, Crescentia's uncle, in retirement in his 64th year

any details transferred from one person (= group) to the next must be explicitly repeated, in accordance with what we have said above. As follows, for example:

```
head-of-household$Hubert/Müller/35/carpenter/Neuberg
wife$Crescentia/Müller/33
son$Hubert/Müller/14/carpenter's apprentice
daughter$Anna/Müller/11
daughter$Walpurga/Müller/8
relative$Friedrich/Hübner/64/retired/relationship=uncle/
    centre-of-family=wife
```

The following format would have corresponded more closely to normal orthographical conventions:

```
head-of-household$Hubert/Müller/35/carpenter/Neuberg
wife$Crescentia/"/33
son$Hubert/"/14/carpenter's apprentice
daughter$Anna/"/11
daughter$Walpurga/"/8
relative$Friedrich/Hübner/64/retired/relationship=uncle/
    centre-of-family=wife
```

You can declare this abbreviated form of input using the `cumulate=` and `substitution=` parameters in the `read` command.

There are no default settings for either of these parameters. They are interpreted as follows:

6.2.3.2.1 The `cumulate=` parameter

If, at the input stage, an element contains the special character which you have assigned to the `cumulate=` parameter, and only this character, it is replaced by the contents of the *element* with the same name which was last encountered in the database. Thus all entries of all aspects are repeated. If an element starts with this character but it is then followed by any other character the system suspects an error and issues a message. If an element contains this particular character in any other position than the initial one, it is treated and processed like any other character.

Thus this parameter allows you to input data just as illustrated above.

However, it always relates to *entire elements*. Hence the following attempt to repeat an item of conceptual information while also appending an additional commentary, would *not* be possible using the `read` command

```
read name=example;cumulate="

head-of-household$Hubert/Müller/35/carpenter/Neuberg
wife$Crescentia/"/33
son$Hubert/"#son by first marriage with ?/14/carpenter's apprentice
daughter$Anna/"/11
daughter$Walpurga/"/8
relative$Friedrich/Hübner/64/retired/relationship=uncle/
    centre-of-family=wife
```

6.2.3.2.2 The `substitution=` parameter

The `substitution=` parameter is used for this purpose.

This behaviour of this parameter depends on whether the entries in the element concerned are subordinated to the aspects or vice versa.

6.2.3.2.2.1 `substitution=` in "simple" entries

If no option has been specified, or if the user used a `element` directive with the parameter `order=simple` in the element concerned, κλειω regards the entries as a further subdivision of the aspects, as we have already seen. Each element then has exactly three aspects, each one of which can be subdivided into as many entries as you like. The following definition applies to elements of this type:

If one aspect consists of a special character which was previously assigned to the `read` command's `substitution=` parameter, this character is replaced by all entries in the aspect which has just been processed in the element of the same name which was last encountered in the input data. If another character follows it within the same aspect, the system suspects an error and issues an error message. If an element contains the particular character in a position other than the first position of an entry it is treated and processed just like any other character.

If the last specified element of the same name does *not* contain the aspect which is to be repeated, the repetition character is ignored and an error message displayed. So if we specify the special character assigned to `substitution=` after the comment character "`#`" in an element, but there was no comment in the last-encountered element of the same name, no attempt is made to find a comment in an element even further back; instead no repetition is made.

The following solution is appropriate for the example just quoted:

```
read name=example;cumulate=";substitution=&
head-of-household$Hubert/Müller/35/carpenter/Neuberg
wife$Crescentia/"/33
son$Hubert/&#son by first marriage with ?/14/carpenter's apprentice
daughter$Anna/&/11
daughter$Walpurga/"/8
relative$Friedrich/Hübner/64/retired/relationship=uncle/
    centre-of-family=wife
```

If from the preceding definitions, it is *not* clear to you why the following solution would have an undesirable side effect (pay particular attention to `Anna Müller`), we would suggest that you enter both examples into your computer and compare the results of a `write` printout *before* attempting to work with the feature.

```
read name=example;cumulate=";substitution=&
head-of-household$Hubert/Müller/35/carpenter/Neuberg
wife$Crescentia/"/33
son$Hubert/&#son by first marriage with ?/14/carpenter's apprentice
daughter$Anna/"/11
daughter$Walpurga/"/8
relative$Friedrich/Hübner/64/retired/relationship=uncle/
    centre-of-family=wife
```

6.2.3.2.2.2 `substitution=` in “multiple” elements

If you have used an element containing a `element` directive with an `order=multiple` parameter, the entries will divide up the aspects, i.e. each one of these elements consists of any number of entries, each one of which includes up to three aspects. The following definition will then apply to `substitution=`:

If an entry consists of a special character which was assigned to the `read` command's `substitution=` parameter, it is replaced by all aspects of the entry with the same ordinal number in the last element of the same name encountered in the input data. If another character follows it within the same aspect, the system suspects an error and displays an error message. If an element contains this particular character in any other than the first position of an entry it is treated and processed just like any other character. If the last-specified element of the same name does *not* contain an entry with the corresponding ordinal number the repetition character is ignored and an error message displayed. Thus if an element containing two entries is followed by an element containing three entries followed by a repetition character this character is ignored. No attempt is made to find an earlier element containing a fourth entry.

6.2.3.3 Support for fixed-format fields

In order to support the option for defining “fixed-format” input fields described in more detail in section 4.5, you can use the `read` command to define two characters which allow you to input and manage such data more efficiently.

The following parameters have no effect on the processing of elements which have not been assigned a fixed length: if the characters defined in this way are encountered in another element they are also processed just like any other character.

6.2.3.3.1 The `null=` parameter

The `null=` parameter expects any character which has not already been reserved as a data signal character as a parameter value.

If an element which has been assigned a fixed length consists exclusively of this character it is treated as if it did not exist.

6.2.3.3.2 The `more=` parameter

The `more=` parameter expects a parameter value consisting of any character which has not already been reserved as a data signal character.

If this character is encountered in an element for which a fixed length has been declared, as the first character in the first entry, it is regarded as introducing a repetition instruction as described in section 4.5.3.1.

6.2.4 Complex redefinitions of input and output

As a rule, it makes sense to preserve input data in separate files. The data can then be read from these files by means of a redirecting `options` command without further ado. However, the disadvantage of this approach is that the `read` command must be written at the beginning of each file containing input data. In order better to resolve this and a range of other problems, you can arrange things so that once the `read` command's vitally necessary parameters have been analysed – for the time being these include the `name=` parameter – and any other parameters required by the user have been processed, the `read` command's remaining specification is processed as if it was an `options` command. You do this by specifying a `form=` parameter.

The `form=` parameter accepts either of the keywords `yes` or `no` as a specification. They can be abbreviated to their respective first character.

- `no` causes the system to treat the remainder of the specification as if it was part of the `read` command; that is, prompts the system to behave as it would by default.
- `yes` causes all parameters following this particular parameter to be interpreted as if they were part of an `options` command (cf. chapter 7 in this manual). Please note that parameters with the same name in a `read` command are in certain circumstances interpreted differently from the way they are interpreted in an `options` command. Thus the value of the `lines=` parameter in a `read` command is cancelled out once this command has been processed. In contrast the value of this parameter is preserved if it is encountered after `form=yes`, hence interpreted as part of an `options` command.

6.3 Examples

The following are examples of correctly written `read` commands:

```
read name=source

read name=source;repeat=yes;form=yes;
      source="data";first=1;limit=17;part="dat"
```

7. The environment of a *κλειω* program

When you activate *κλειω* 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 *κλειω*. 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 *κλειω* program is executed.

7.1.1 General system behaviour

κλειω'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

- κλειω 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 κλειω 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 κλειω 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, κλειω 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 κλειω 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 κλειω 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.

κλειω 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 *κλειω* 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. *κλειω* 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 κλειω 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 κλειω 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 κλειω 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 κλειω 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 κλειω 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, *κλειω* 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, *κλειω* 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 15th 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 15th 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, $\kappa\lambda\epsilon\iota\omega$ 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 1st, 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 κλειω 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 $\kappa\lambda\epsilon\iota\omega$ 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 $\kappa\lambda\epsilon\iota\omega$ 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.

7.3.1.7.5.1 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 preprocess 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 κλειω, 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 *κλειω*.

You can test the `skeleton` 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.

7.3.1.10 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 *κλειω* 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 *κλειω*'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.

7.3.1.13 The `location` definition

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

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

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

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

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

7.3.1.13.1 The `location` directive

This directive has the general form:

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

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

7.3.1.13.1.1 Declaring a topographical object

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

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

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

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

7.3.1.13.1.1.1 The `first=` parameter

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

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

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

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

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

7.3.1.13.1.1.2 The `order=` parameter

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

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

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

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

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

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

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

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

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

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

7.3.1.13.1.1.3 The `second=` parameter

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

7.3.1.13.1.2 Displaying a topographical object

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

7.3.1.13.1.2.1 The `colour=` parameter

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

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

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

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

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

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

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

7.3.1.13.1.2.2 The `lines=` parameter

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

It defines the line width of the contours.

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

7.3.1.13.1.2.3 The `usage=` parameter

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

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

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

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

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

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

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

7.3.1.13.1.2.4 The `sign=` parameter

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

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

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

7.3.1.13.1.2.5 The `write=` parameter

This parameter expects a constant as a parameter value.

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

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

7.3.1.13.1.2.6 The `symbol=` parameter

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

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

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

7.3.1.13.1.2.7 The `size=` parameter

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

7.3.1.13.1.3 A topographical object's topological properties

7.3.1.13.1.3.1 The `connected=` parameter

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

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

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

7.3.1.13.1.4 Inputting a topographical object

7.3.1.13.1.4.1 The `form=` parameter

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

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

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

7.3.1.13.1.4.1.1 The `digipad` format

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

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

Hence the `location` directive:

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

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

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

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

7.3.1.13.1.4.1.2 The `arcinfo` format

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

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

In this format the example given above would therefore read:

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

7.3.1.13.1.5 The ownership of a topographical object

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

7.3.1.13.1.5.1 The `always=` parameter

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

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

7.3.1.13.1.5.2 The `name=` parameter

3.3.5) The `name=` parameter.

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

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

7.3.1.13.2 The `type` directive

This directive has the general form:

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

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

7.3.1.13.2.1 The `type=` parameter

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

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

7.3.1.13.2.2 The `colour=` parameter

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

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

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

7.3.1.13.2.3 The `lines=` parameter

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

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

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

7.3.1.13.2.4 The `usage=` parameter

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

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

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

7.3.1.13.2.5 The `sign=` parameter

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

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

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

7.3.1.13.2.6 The `form=` parameter

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

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

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

7.3.1.13.3 The `source` directive

This directive has the general form:

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

This directive is used to link up parts of a map which have been digitized separately. It provides a number of methods to do so. Which method is choosen, depends on the subset of parameters specified.

7.3.1.13.3.1 Simple offset method

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

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

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

7.3.1.13.3.2 Shifting static origin

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

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

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

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

7.3.1.13.3.3 Shifting variable origin

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

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

7.3.1.13.3.4 Shifting scales

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

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

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

7.3.1.14 Defining sort sequences

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

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

7.3.1.14.1 The default sort sequence

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

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

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

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

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

7.3.1.14.2 The `order` definition

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

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

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

7.3.1.14.2.1 The `sign` directive

This directive has the general form:

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

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

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

7.3.1.14.2.1.1 The `sign=` parameter

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

7.3.1.14.2.1.2 The `usage=` parameter

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

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

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

7.3.1.14.2.1.3 The `after=` parameter

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

```
sign sign=ä;after=a
```

7.3.1.14.2.1.4 The `before=` parameter

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

```
sign sign=a;before=ä
```

7.3.1.14.2.1.5 The `order=` parameter

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

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

7.3.1.14.2.1.6 Interaction between the parameters

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

7.3.1.14.2.2 The `conversion` directive

This directive has the general form:

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

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

7.3.1.14.2.2.1 The `before=` parameter

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

7.3.1.14.2.2.2 The `without=` parameter

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

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

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

7.3.1.14.2.2.3 The `form=` parameter

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

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

```
+1792+seventeenhundredandninetytwo+
```

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

7.3.1.14.2.2.4 Sorting older character conventions

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

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

where the $< modifier >$ is normally numeric.

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

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

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

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

7.3.1.14.2.2.4.1 The `sign=` parameter

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

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

7.3.1.14.2.2.4.2 The `usage=` parameter

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

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

7.3.1.14.2.3 The `training` directive

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

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

7.3.1.15 The `catalogue` definition

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

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

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

7.3.1.15.1 The `keyword` directive

This directive has the general form:

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

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

7.3.1.15.1.1 The `name=` parameter

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

7.3.1.15.1.2 The `form=` parameter

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

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

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

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

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

7.3.1.15.1.2.1 Terms

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

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

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

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

7.3.1.15.1.2.2 Built-in functions

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

7.3.1.15.1.2.2.1 The `name[]` built-in function

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

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

7.3.1.15.1.2.2.2 The `lemma[]` built-in function

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

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

7.3.1.15.1.2.3 Linkage operators

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

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

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

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

7.3.1.15.1.2.4 Logical brackets

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

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

7.3.1.15.1.3 The `bridge=` parameter

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

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

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

7.3.1.15.1.4 The `overwrite=` parameter

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

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

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

7.3.1.16 The `connection` definition

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

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

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

7.3.1.16.1 The `usage` directive

This directive has the general form:

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

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

7.3.1.16.1.1 The `read=` parameter

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

7.3.1.16.1.2 The `name=` parameter

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

7.3.1.17 The `image` definition

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

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

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

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

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

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

To administer images, κλειω uses a two-level abstraction of the equipment which is actually available on a given machine.[2]

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

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

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

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

7.3.1.17.1 The `part` directive

This directive has the general form:

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

It defines the conditions for all subsequent directives.

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

7.3.1.17.1.1 The `type=` parameter

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

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

7.3.1.17.1.2 The `device=` parameter

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

7.3.1.17.1.3 The `volume=` parameter

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

7.3.1.17.2 The `device` directive

This directive has the general form:

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

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

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

7.3.1.17.2.1 The `name=` parameter

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

7.3.1.17.2.2 The `target=` parameter

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

7.3.1.17.2.3 The `usage=` parameter

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

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

7.3.1.17.2.4 The `type=` parameter

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

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

7.3.1.17.3 The `volume` directive

This directive has the general form:

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

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

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

7.3.1.17.3.1 The `name=` parameter

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

7.3.1.17.3.2 The `target=` parameter

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

7.3.1.17.3.3 The `type=` parameter

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

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

7.3.1.17.3.4 The `device=` parameter

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

7.3.1.17.4 The `image` directive

This directive has the general form:

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

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

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

7.3.1.17.4.1 The `name=` parameter

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

7.3.1.17.4.2 The `source=` parameter

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

7.3.1.17.4.3 The `type=` parameter

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

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

7.3.1.17.4.4 The `device=` parameter

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

7.3.1.17.4.5 The `volume=` parameter

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

7.3.1.17.4.6 The `write=` parameter

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

7.3.1.17.5 The `visibility` directive

This directive has the general form:

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

This directive changes the visibility of

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

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

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

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

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

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

7.3.1.18 The `chronology` definition

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

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

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

7.3.1.18.1 Syntax required for saints' day dates

7.3.1.18.1.1 Simple calendar terms

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

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

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

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

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

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

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

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

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

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

7.3.1.18.1.2 Composite calendar terms

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

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

$$< Day\,Count > \quad = \quad [< number >] \begin{Bmatrix} < Weekday > \\ < Day > \end{Bmatrix}$$

This composite calendar term corresponds to expressions such as:

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

$< Feast >$ is defined as:

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

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

This composite calendar term corresponds to expressions like:

```
Walpurgis octave
Walpurgis Eve
Walpurgis
```

7.3.1.18.1.3 Dates of religious festivals (feast days)

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

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

Hence a feast day date corresponds to expressions like:

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

7.3.1.18.1.4 Periods of time

According to the rules for the `date` data type, every κλειω date potentially consists of a *terminus ante quem* and a *terminus post quem*. Expressions written like this:

```
Walpurgis 1567 - Johannis 1567
```

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

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

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

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

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

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

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

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

7.3.1.18.2 Individual directives for the `chronology` definition.

7.3.1.18.2.1 The `name` directive

The general form of the `name` directive is:

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

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

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

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

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

7.3.1.18.2.1.1.1 Defining fixed feast day dates

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

Example:

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

7.3.1.18.2.1.1.2 Defining feast days in relation to Easter

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

Examples:

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

7.3.1.18.2.1.1.3 Defining movable feast day dates

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

Examples:

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

7.3.1.18.2.1.1.4 Defining an unknown < *Feast Day* >

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

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

Example:

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

7.3.1.18.2.1.2 Calendar terms of the < *Weekday* > type

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

Examples:

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

7.3.1.18.2.1.2.1 Defining an unknown < *Weekday* >

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

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

Example:

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

7.3.1.18.2.1.3 Calendar terms of the $< Proximity >$ **type**

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

Example:

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

7.3.1.18.2.1.3.1 Defining an unknown $< Proximity >$

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

Example:

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

7.3.1.18.2.1.4 Calendar terms of the $< Day >$ **type**

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

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

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

7.3.1.18.2.1.5 Calendar terms of the $< Octave >$ **type**

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

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

Example:

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

7.3.1.18.2.1.6 Calendar terms of the < *Eve* > type

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

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

Example:

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

7.3.1.18.2.1.7 Calendar terms of the < *Period* > type

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

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

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

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

Example:

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

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

```
week after Martinmas
```

would be interpreted as:

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

7.3.1.18.2.2 The conversion directive

The general form of the conversion directive is:

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

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

7.3.1.18.2.2.1 The preparation= parameter

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

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

7.3.1.18.2.3 The training directive

The general form of the training directive is:

EXPLain=No | Yes   (default: No)

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

7.3.1.18.2.3.1 The explain= parameter

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

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

7.3.1.19 The `classification` definition

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

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

7.3.1.19.1 The purpose of embedded classifications

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

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

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

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

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

Let us assume the following user defined symbols:

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

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

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

While this example is not complete, it shows the basic mechanism:
Rule 1: for a multiple-valued subclassification we announce which of the classes is applicable by specifying the indicator for the appropriate class.

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

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

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

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

This can be done by the following markup:

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

which applies the following rules:

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

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

As a result we would get:

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

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

7.3.1.19.2 Individual directives

7.3.1.19.2.1 The `part` directive

The general form of the `part` directive is:

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

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

7.3.1.19.2.1.1 The `name=` parameter

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

7.3.1.19.2.1.2 The `more=` parameter

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

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

7.3.1.19.2.1.3 The `without=` parameter

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

7.3.1.19.2.1.4 The `write=` parameter

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

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

7.3.1.19.2.2 The `sign` directive

The general form of the `sign` directive is:

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

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

7.3.1.19.2.2.1 The `sign=` parameter

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

7.3.1.19.2.2.2 The `write=` parameter

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

7.3.1.19.2.3 Example

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

```
item name=example;usage=classification

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

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

exit name=example
```

7.3.2 The `delete` command

The general form of the `delete` command is:

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

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

7.3.2.1 The `name=` parameter

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

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

7.3.2.2 The `usage=` parameter

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

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

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

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

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

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

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

7.3.2.3 The `type=` parameter

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

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

7.3.2.4 The `source=` parameter

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

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

7.3.2.5 The `catalogue=` parameter

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

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

7.3.2.6 The `each=` parameter

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

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

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

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

7.3.3 The `describe` command

The general form of the `describe` command is:

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

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

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

7.3.3.1 The `name=` parameter

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

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

7.3.3.2 The `usage=` parameter

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

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

- `codebook` displays a codebook, regardless of whether it was generated by `item` or `create`.
- `catalogue` displays a catalogue generated by the `catalogue` command. If you want to do this, you should also specify `type=permanent`.
- `keyword` displays a list of references stored in a catalogue's reference library. If you have not yet used the `catalogue[]` built-in function to allocate a catalogue, you must explicitly specify a catalogue in the `catalogue=` parameter.
  The list of references is reproduced in the syntax of the `form=` parameter of the `catalogue` definition's `keyword` directive (cf. section 7.3.1.15.1.2). If you redirect the

output from this command to a file it can be used as a quick and efficient way of restoring earlier lists of references should you decide to recreate a catalogue and/or a database.

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

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

7.3.3.3 The `type=` parameter

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

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

7.3.3.4 The `source=` parameter

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

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

7.3.3.5 The `write=` parameter

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

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

- `usage=source`
  1 Here `write=` defines the amount of information about the database to be displayed. You can specify this parameter as often as you like, using a different keyword each time in order to obtain the printout you require.
  - `names` instructs the system to print out a list of descriptions of all valid element names in the database,
  - `parts` instructs the system to print out a list of descriptions of all valid group names in the database,
  - `structure` instructs the system to print out a description of the structural relationships between all non-generic identifiers in the database, specifying any structural restrictions in force (e.g. minimum/maximum permissible frequencies) and

  - `generic` instructs the system to print out a list of descriptions of all names valid as generic names (for elements *and* groups) in the database.

- `usage=location`
  Here `write=` defines the way in which the contents of a `location` definition should be reproduced.
  - `text` describes the prevailing settings for each individual topographical object.
  - `mapping` instructs the system to produce a map, containing all the topographical objects defined in the `location` definition, together with their standard attributes.

- `usage=image`
  Here `write=` defines the way in which the various types of items within a `image` definition are listed.
- `image` instructs the system to give a complete list of all the images described by the definition. If no other `write` parameter has been specified all the images are listed in one list. If any of the following parameter values are specified also, the images are listed under the device and/or the volume on which the reside.
- `volume` instructs the system to give a complete list of all the volumes in the definition. If `write=device` is not specified all volumes are described in one list, otherwise the volumes are listed under the devices on which the can be mounted.
- `device` instructs the system to give a complete list of all the devices in the definition.

7.3.3.6 The `catalogue=` parameter

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

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

7.3.4 Defining environment-related default settings

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

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

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

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

7.3.4.0 The durability of logical objects

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

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

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

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

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

Thus in the following κλειω task:

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

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

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

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

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

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

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

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

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

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

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

7.3.4.1 The `text` command

The general form of the `text` command is:

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

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

By default, the system assumes that:

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

7.3.4.1.1 The `name=` parameter

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

7.3.4.2 The `date` command

The general form of the `date` command is:

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

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

By default, the system assumes that:

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

7.3.4.2.1 The `name=` parameter

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

7.3.4.2.2 The `text=` parameter

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

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

7.3.4.2.3 The `minimum=` parameter

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

This date becomes the earliest valid date in the database.

* These defaults can be changed with the configuration program.

7.3.4.2.4 The `maximum=` parameter

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

This date becomes the latest valid date in the database.

7.3.4.2.5 The `first=` parameter

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

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

7.3.4.3 The `number` command

The general form of the `number` command is:

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

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

By default, the system assumes that:

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

7.3.4.3.1 The `name=` parameter

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

7.3.4.3.2 The `text=` parameter

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

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

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

7.3.4.3.3 The `minimum=` parameter

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

This number becomes the lowest valid number in the database.

7.3.4.3.4 The `maximum=` parameter

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

This number becomes the highest valid number in the database.

7.3.4.4 The `category` command

The general form of the `category` command is:

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

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

By default, the system assumes that:

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

7.3.4.4.1 The `name=` parameter

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

7.3.4.5 The `relation` command

The general form of the `relation` command is:

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

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

By default, the system assumes that:

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

7.3.4.5.1 The `name=` parameter

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

7.3.4.5.2 The `text=` parameter

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

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

7.3.4.5.3 The `cumulate=` parameter

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

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

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

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

7.3.4.5.4 The `source=` parameter

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

7.3.4.5.5 Choosing visibility and view

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

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

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

7.3.4.6 The `location` command

The general form of the `location` command is:

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

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

By default, the system assumes that:

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

7.3.4.6.1 The `name=` parameter

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

7.3.4.6.2 The `usage=` parameter

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

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

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

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

7.3.4.7 The `image` command

The general form of the `image` command is:

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

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

By default, the system assumes that:

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

7.3.4.7.1 The `name=` parameter

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

7.3.4.7.2 The `usage=` parameter

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

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

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

8. Working with individual databases

In order to work with a database, you must first specify:

- which database you want to work with,
- which parts of the database the system should select for processing and
- what you want the system to do with the information you extract.

8.1 General constituents of commands

In addition to the parameter values described in section 2.1.1.3.1.1, which apply to all system components, the parts of the system documented in this chapter accept a series of additional parameter values which are used to extract the set of information you want to process from a database.

8.1.1 Paths

Information you want to extract from a database for a particular purpose is described by a *path*. A path consists of a series of specifications which *κλειω* uses to address a sequence of entries. Each entry in turn is subjected to the required processing. When the system reaches the last entry, it proceeds down the next path specified in the command. Once it reaches the last path the task is terminated, and the extracted information as a whole is subjected to a final post-processing operation – e.g. a sort – if appropriate.

Paths consist of the following:

- group identifiers,
- group functions,
- element identifiers,
- element functions,
- expressions,
- network transitions and
- conditions.

8.1.1.1 Group identifiers

A group identifier consists of the name of a group, which is included in the data structure of the relevant database. It represents the set of all those groups which can be addressed by this name. The system recognises a group identifier in a parameter by the fact that

- it appears at the head of a path or
- a data signal character 2 appears in front of it.

8.1.1.1.1 Absolute group identifiers

A group identifier is absolute if it appears at the head of a path down which the system can proceed without being dependent on any other path or paths. The following are examples of absolute group identifiers:

- `child`
- `/child`

The system interprets both identifiers as the set of all groups called “child” contained in a given database.

8.1.1.1.2 Relative group identifiers

A group identifier is relative if it is used in a path which is introduced by another group identifier or which depends on another path in the context of a given task.

For example, the group identifier "child" in the following paths is relative in each case:

- `person/child`
- `marriage/child`

The system interprets the first of these identifiers as the set of all those groups called "child" which depend directly on a group called "person" in a given database; the second as the set of all those groups called "child" which depend directly on the group "marriage".

8.1.1.2 Group functions

A group function is a built-in function which occupies the same sort of position in a path in which you might also expect to find a group identifier. Its value consists of a set of groups defined by its arguments.

8.1.1.2.1 Group function arguments

A group function's arguments define a set of groups. Where group functions are nested, the system applies each outer function in turn to every element in the set represented by the inner function.

8.1.1.2.1.1 Keywords

Keywords in group functions can always be written in abbreviated form, consisting simply of the first character in the keyword.

8.1.1.2.1.2 Group specifications

A group specification consists of a group identifier or group function.

8.1.1.2.1.3 Number ranges

A number range consists of a list of comma-delimited interval specifications.

An interval specification comprises:

- a positive whole number (integer),
- two positive integers linked by a hyphen,
- an integer preceded by a hyphen, which the system interprets as "1-n", or
- an integer followed by a hyphen, which the system interprets as "n-1000000".

If a number range is missing from a function which accepts number ranges, the system substitutes the number range "1".

A number falls within a number range if it corresponds to a number contained in that range or occurs between two of the numbers describing an interval specification.

Intervals in a number range do not have to appear in any particular order, and may overlap.

8.1.1.2.1.4 Label ranges

A label range consists of a list of comma-delimited label intervals.

A label interval comprises:

- a constant which the system interprets as a group identification,
- two constants joined by a hyphen,
- a constant preceded by a hyphen, which the system interprets as $< first\ group\ identification\ to\ appear > - < constant >$ or
- a constant followed by a hyphen, which the system interprets as $< constant > - < last\ group\ identification\ to\ appear >$.

If a label range is missing from a function which requires such a range, this is interpreted as an error.

A group is included in a label range if it has a group identification identical to one of the identifications in the range or if it is logically positioned between two groups with group identifications which form a label interval.

Intervals in a label range do not have to appear in any particular order and may overlap.

8.1.1.2.1.5 Numbers

A number is a positive whole number (integer).

8.1.1.2.1.6 Element specifications

An element specification consists of an element identifier or element function.

8.1.1.2.2 Individual group functions

Unless you specify an argument in the following functions, no character other than a number (any number) of spaces should appear between the enclosing parentheses.

Any surplus arguments are treated as errors.

8.1.1.2.2.1 The group function `query[]`

The group function `query[]` is written in one of two ways:

```
query[]
```

or

```
query[always | temporary]
```

The system interprets this function as the last group reached by the previous `query` command. The last group reached by `query` is the group at the end of the path in front of the *first* equals sign which introduces a condition. The system takes no account of groups preceded by `and` or `or` or which form comparands.

- If the function is invoked without an argument or with `always`, *all* following paths are relative to the group expressed by `query[]`.
- If it is invoked with `temporary` it instructs the system that the *path immediately adjoining it* is relative to this group, but that it should restore the status prevailing prior to `query[]` for all subsequent paths.
  We recommend that you use `temporary` in built-in functions by preference. Built-in functions in which `query[]` together with `always` would have known and undesirable effects (such as `continue[]`) always imply this argument in any case.

8.1.1.2.2.2 The group function `sign[]`

The group function `sign[]` is written as follows:

```
sign[group specification, label range]
```

The system interprets it as the set of groups defined by *group specification* located within the *label range*.

8.1.1.2.2.3 The group function `catalogue[]`

The group function `catalogue[]` is written as follows:

```
catalogue[name,complete | start | limit | algorithm,
        element specification | constant]
```

This function accepts arguments which can only be understood in the context of processing catalogues. (cf. section 8.3.2 of this manual for more information on this topic). *Name* specifies the name of a catalogue generated by a `catalogue[]` command.

The second argument specifies which access path the system should choose inside the catalogue:

- `complete` instructs the system to choose the access path which allows it to search for a term as a whole.
- `start` instructs the system to choose the access path which allows it to search for a term beginning with a known character string.
- `limit` instructs the system to choose the access path which allows it to search for a term ending in a known character string.

- `algorithm` instructs the system to choose the access path which demands that the search term should first be converted into a derived form (a lemma, a soundex code or a skeletonising code). You should not (and cannot) tell the system which procedure to use for this purpose: the system ensures that it uses the same procedure for this derivation as it used to implement this access path when the catalogue was first created.

If the second argument is missing, the system assumes that it is `complete`: if you want to specify the third argument explicitly, however, you must ensure that the second argument is written.

If the third argument is missing, the system interprets the function as the set of all groups which caused at least *one* entry to be made in the *name* catalogue. Each group appears in this set the same number of times as it caused entries to be made in the *name* catalogue. These groups are processed in the same order as the entries in the specified access path for the catalogue *name*.

If the third argument is a *constant* the system interprets the function as the set of all groups which caused at least one entry to be made in the *name* catalogue, where the entry can be reached via this *constant* in the specified access path, i.e. where the entry is identical to, starts or ends with this constant, or can be brought into the same derivable form as this constant.

The system examines the third argument to see whether it can be used as a search pattern in the sense defined in section 7.3.1.15.1.2 once the outer delimiting characters have been stripped off. If it can, then the function is defined as the set of all groups which can be reached via this pattern. It is important to remember that the third argument as a whole should be enclosed by a pair of delimiters. Thus in order to access the set of all those groups in which contexts containing the word `church` as well as the word `school` appear, you would need to make the following function call:

```
catalogue[example,complete,"'school' and 'church'"]
```

In this case, the system ignores the second argument, because the access path to be used for the pattern's separate components is evident from the pattern definition. In the interests of consistency, however, you must specify a valid keyword for the second argument, even in this formulation.

If the third argument is an *element specification*, the system interprets the function as the set of all groups which have caused at least one entry to be made in the *name* catalogue, where the entry can be reached via an element in the set of entries of the `text` data type described by the *element specification* in the specified access path, i.e. where the entry is identical to such an entry, or begins or ends with it, or can brought into the same derivable form as this entry.

Each group appears in this set the same number of times as it caused entries of this type to be made in the *name* catalogue. The groups are processed in the same order as the entries in the *name* catalogue.

The only place in a path where any form of this function should appear is at a position where an absolute group name would be valid, i.e. as the first component of a path which does not depend on any other path.

8.1.1.2.2.4 The group function `keyword[]`

The group function `keyword[]` is written as follows:

```
keyword[name-1,name-2]
```

This function accepts non-standard arguments which can only be understood in the context of catalogue processing (cf. section 8.3.2 of this manual) and interactive information retrieval (cf. Chapter 7 in the Tutorial volume and section 8 of volume B10 of the "Halbgraue Reihe". *Name-1* specifies the name of a catalogue generated by a `catalogue` command. *Name-2* specifies a list of references stored for later use with the help of the `save` option on the main menu for interactive processing of catalogues (cf. section 8.1.2.3 of volume B10 of the "Halbgraue Reihe").

The system interprets this function as the set of all groups which have contributed at least one reference to the list of references *name-2* contained in the catalogue *name-1*. Each group appears in the set the same number of times as it contributed references to the list of references *name-2*. The groups are processed in the same order as the entries in the catalogue *name-1*.

The only place in a path where this function should appear is at a position where an absolute group name would be valid, i.e. as the first component of a path which does not depend on any other path.

8.1.1.2.2.5 The group function `brother[]`

The group function `brother[]` is written as follows:

```
brother[group specification,left | right,number range]
```

The system interprets it as the set of groups

- with the same name as an element in the set expressed by *group specification* (henceforth called *group* for short),
- (if the second argument is `left`) which in the same group as the one containing the *group*, have lower ordinal numbers than the *group* (i.e. appear "in front of" the latter),
- (if the second argument is `right`) which in the same group as the one containing the *group*, have higher ordinal numbers than the *group* (i.e. appear "after" the latter) and
- which are positioned at a distance from the *group* which is still within the *number range*.

8.1.1.2.2.6 The group function `father[]`

The group function `father[]` is written as follows:

```
father[element specification]
```

The system interprets it as the set of groups containing the elements expressed by *element specification*.

8.1.1.2.2.7 The group function `same[]`

The group function `same[]` is written as follows:

`same[group specification,left | right,number range]`

The system interprets it as the set of groups which

- have the same name as an element in the set expressed by *group specification* (henceforth referred to as *group*),
- (if the second argument is `left`), appear "in front of" the *group* in the database,
- (if the second argument is `right`), appear "after" the *group* in the database and
- are located at a distance from the *group* which is still within the *number range*.

8.1.1.2.2.8 The group function `part[]`

The group function `part[]` is written as follows:

`part[group specification,number range]`

The system interprets it as the set of groups which form an element in the set defined by *group specification* and have an ordinal number within the *number range*.

8.1.1.2.2.9 The group function `forward[]`

The group function `forward[]` is written as follows:

`forward[group specification,number range]`

The system interprets it as the set of groups which are logically immediately subordinated to the elements in the set defined by *group specification* (henceforth referred to as *group* for short) and appear at positions in the sequence of types of groups contained in the *group* which are within the *number range*.

8.1.1.2.2.10 The group function `total[]`

The group function `total[]` is written as follows:

`total[group specification]`

The system interprets it as the set of groups which are logically directly or indirectly subordinated to the elements in the set defined by *group specification*.

8.1.1.2.2.11 The group function `root[]`

The group function `root[]` is written as follows:

`root[number]`

This is defined as the group on logical level *number* which logically contains the group which the system last activated. (Documents appear on logical level zero; groups which are logically immediately subordinated to them on logical level one, etc.). A group located on a logical level with an ordinal number equal to or less than *number* is regarded as its own `root[]`.

8.1.1.2.2.12 The group function `back[]`

The group function `back[]` is written in one of two ways:

`back[number]`

or

`back[group identifier]`

The system interprets the first of these two forms as the group which appears *number* logical levels above the last group to be activated.

If the last group to be activated is less than *number* logical levels from its parent document, the system does not select a group for processing.

The system interprets the second of these two forms as the first group called *group identifier* which logically appears “above” the last group to be activated. If no such group is located anywhere in the hierarchy up to and including the document which contains the present group, the system does not select a group for processing.

This interpretation of the parameter values is best illustrated with the help of a diagram:

If we assume the following data structure:

```
Document
    |
    |
    | → Group A
    |     |
    |     |
    |     | → Group B
    |     |     |
    |     |     |
    |     |     | → Group C
```

In the current group	corresponds	to group
Group C	back[2]	Group A
Group C	back[Group A]	Group A
Group B	back[2]	document
Group B	back[Group A]	Group A
Group A	back[2]	not defined
Group A	back[Group A]	not defined

8.1.1.2.2.13 The group function `continue[]`

The group function `continue[]` is written as follows:

```
continue[group specification,element specification,condition,
        number,self]
```

The system interprets it a set of groups which is derived dynamically from the data. For a better understanding of how this works, we must describe its behaviour in a path. In the path `path_section-1 continue[...]  path_section-2`, `continue[]` is defined by the set of groups reached by the following operation:

a) Once *path_section-1* has been processed, `continue` proceeds along the path defined by *group specification* and then processes *path_section-2*.
b) Now the system attempts to find the *element specification* of the `relation` data type in the group which it has reached.
c) If you specified the third argument, the system determines whether the entries made accessible by the argument satisfy the *condition*, which must be a relational search pattern.
d) Where an entry does fulfil this condition, the system sets up a transition to the group to which the entry refers. (If you did not specify a *condition*, the system does this for all entries).
e) Finally, the system processes *path_section-2* again, and the process proceeds to step b.

In the normal way, the system terminates this process if step b) or c) fails to unearth an element which can be traced any further. However, since it is quite possible to formulate the individual paths concerned in such a way that a group refers back to itself by various roundabout ways, we have included a number of safeguards. By default, the following rules apply:

- The system stops processing step b) if has already been successful 10 times.
- The system stops processing step d) if it returns you to a group which you reached earlier while the system was processing steps a) or d).

You can control both default settings as follows:

- If you specify *number*, the system refrains from processing step b) once it has been successful *number* times. (“Maximum depth of recursion”)
- If you define `self` as the fifth argument, the system does not check to see whether it has already processed one and the same group. You can only specify the fifth argument if you also specify the fourth; in this case, we would strongly advise you to set the latter to a low value.

8.1.1.2.2.14 The group function `last[]`

The group function `last[]` is written as follows:

```
last[relative group specification]
```

It is defined as the set of the last occurences of the last part of the group specification within the previous part of that specification.

That is:

```
last[group/subgroup]
```

will for each `group` in the database select the last occurence of `subgroup` within this occurence of group. Or, to give a more meaningful example:

```
last[marriage/child]
```

will access all the last born `children` of all `marriages`, provided they occur within the respective marriages in the order of their births.

All considerations of command relativity apply. That is, the example

```
query name=example;part=marriage
write part=last[child]:each[]
```

will work exactly the same as

```
query name=example
write part=last[marriage/child]:each[]
```

8.1.1.2.2.15 The group function `selection[]`

The group function `selection[]` is written as follows:

```
selection[first | last,relative group specification,
        element specification,condition]
```

It is defined as the set of the *first* or *last* occurences of the last part of the group specification within the previous part of that specification which contains an *element specification* fulfilling *condition*. All considerations of command relativity apply. That is

```
query name=example;part=marriage
write part=selection[last,child,:birthdate,
        before "1 january 1700"]:each[]
```

will display each element of the *last* `child` in each `marriage` with a `birthdate` prior to the 1^{st} of January 1700;

```
query name=example
write part=selection[first,marriage/child,:birthdate,
        after "1 january 1700"]:each[]
```

will display each element of the *first* `child` in each `marriage` with a `birthdate` after the 1^{st} of January 1700.

8.1.1.2.2.16 The group function `none[]`

The group function `none[]` is written as follows:

```
none[group specification,relative element specification,condition]
```

It is defined as the set of all groups out of *group specification* which do *not* contain any *relative element specification* that fulfills *condition.*

That is

```
write part=none[marriage,child:birthdate,before "1.1.1600"]:each[]
```

will display information about all `marriages` where for *all* `children` the `birthdate` is either unknown or later than the 1^{st} of january 1600. Each such `marriage` is displayed once.

8.1.1.3 Element identifiers

An element identifier consists of the name of an element, which is included in the data structure of the relevant database. It represents the set of all those elements which can be addressed by this name. The system recognises an element in a parameter by the fact that it appears after a data signal character 9.

8.1.1.3.1 Absolute element identifiers

An element identifier is absolute if it appears at the head of the name of a path down which the system can proceed without being dependent on any other path or paths. The following is an example of an absolute element identifier:

- `:surname`

The system interprets this identifier as the set of all elements called “surname” contained in a given database.

8.1.1.3.2 Relative element identifiers

An element identifier is relative if it is used in a path which is introduced by a group identifier or which depends on another path in the context of a given task. For example, the element identifier “surname” in the following path names is relative in each case:

- `person/child:surname`
- `marriage/child:surname`

The system interprets the first of these identifiers as the set of all elements called “surname” which depend on a group called “child” in a given database, which in turn depends on a group called“person”; it interprets the second identifier as the set of all elements called “surname” which depend on a group called “child” in a given database, which in turn depends directly on a group called “marriage”.

8.1.1.4 Element functions

An element function is an built-in function which occupies the same sort of position in a path in which you might also expect to find an element identifier. Its value consists of a set of elements defined by its arguments.

8.1.1.4.1 Element function arguments

Element function arguments define a set of elements, which – depending on the context – the system may also interpret as a set of the entries contained in their basic information or one of their other aspects, or as a set of entries calculated from other components relating to them. Where element functions are nested, the system applies each outer function in turn to every element in the set represented by the inner function.

8.1.1.4.1.1 Keywords

Keywords in element functions can always be written in abbreviated form, consisting simply of the first character in the keyword.

8.1.1.4.1.2 Element specifications

An element specification consists of an element identifier or element function.

8.1.1.4.1.3 Number ranges

A number range consists of a list of comma-delimited interval specifications.

An interval specification comprises:

- a positive whole number (integer),
- two positive integers linked by a hyphen,
- an integer preceded by a hyphen, which the system interprets as “1-n”, or
- an integer followed by a hyphen, which the system interprets as “n-1000000”.

If a number range is missing from a function which accepts number ranges, the system substitutes the number range “1”.

A number falls within a number range if it corresponds to a number contained in that range or occurs between two of the numbers describing an interval specification.

Intervals in a number range do not have to appear in any particular order and may overlap.

8.1.1.4.1.4 Group specifications

A group specification consists of a group identifier or group function.

8.1.1.4.1.5 Conversion specifications

Conversion specifications tell the system which rules to use in order to convert an entry of the `text` data type into a required target data type before processing it. This language element has not yet been implemented.

8.1.1.4.1.6 Constants

A constant is always a string of characters framed by delimiters, as described in section 2.1.1.3.1.1.1.

8.1.1.4.1.7 Definition names

Definition names are the names of logical objects, defined by the `item` command (cf. section 7.3.1.1).

8.1.1.4.1.8 Names of variables

Names of variables are names which select one of several options in a logical object, e.g. a codebook.

8.1.1.4.2 Individual element functions

Unless you specify an argument in the following functions, no characters other than a number (any number) of spaces are allowed to appear between the enclosing parentheses.

Any surplus arguments are treated as errors.

8.1.1.4.2.1 The element function `name[]`

The element function `name[]` is written in one of the two following ways:

```
name[]
```

or

```
name[element specification]
```

- In the first case, it is defined as an entry of the `text` data type containing the name of the last group to be activated.
- In the second case, it is defined as the set of all entries of the `text` data type, which can be calculated from the names of the set of elements defined by *element specification*.

8.1.1.4.2.2 The element function `sign[]`

The element function `sign[]` is written as follows:

```
sign[]
```

It is defined as an entry of the `text` data type containing the group identification of the last group to be activated.

8.1.1.4.2.3 The element function `usage[]`

The element function `usage[]` is written as follows:

```
usage[element specification]
```

It is defined as a set of entries of the `text` data type which represents the data type of all entries defined by the *element specification*.

8.1.1.4.2.4 The element function `brother[]`

The element function `brother[]` is written as follows:

```
brother[element specification,left | right,number range]
```

The system interprets it as the set of entries which

- appear in the same element as the last entry to be activated by the *element specification* (and henceforth referred to as *entry*),
- (if the second argument is `left`), appears in the same element “in front of” the entry,
- (if the second argument is `right`), appears in the same element “after” the entry and
- is positioned at distance away from the *entry* which is still within the *number range*.

8.1.1.4.2.5 The element function `cousin[]`

The element function `cousin[]` is written as follows:

```
cousin[element specification,first | comment | original]
```

It is defined as the set of entries which are located in the aspect defined by the *keyword* of the set of entries defined by the *element specification*.

At the present time this element function can only be used for element functions where the `element` directive has been assigned the parameter `order=multiple`. As a rule, it is used to process databases in which the equivalent entry from another aspect is to be processed for a given entry. This is necessary, for example, if you choose an input convention where an element's basic information includes the names of goods and the associated original text field contains the price quoted in the source, as demonstrated below:

```
commodity=name-1%2.1;name-2%1.3;name-3%2.5;name-4%3.2
```

In this example, the following task would cause the system to print out the prices for a commodity forming the object of a search:

```
query part=:commodity="identifier (forming object of search)"
write part=:cousin[:query[],original]
```

8.1.1.4.2.6 The element function `status[]`

The element function `status[]` is written as follows:

```
status[element specification]
```

It is defined as a set of entries of the `text` data type which represents the status of all entries defined by the *element specification*.

8.1.1.4.2.7 The element function `first[]`

The element function `first[]` is written as follows:

```
first[element specification]
```

It is defined by the set of all entries in the "basic information" aspect of the set of all elements defined by the *element specification*.

8.1.1.4.2.8 The element function `original[]`

The element function `original[]` is written as follows:

```
original[element specification]
```

It is defined by the set of all entries in the "original text" aspect of the set of all elements defined by the *element specification*.

8.1.1.4.2.9 The element function `comment[]`

The element function `comment[]` is written as follows:

```
comment[element specification]
```

It is defined by the set of all entries in the "comment" aspect of the set of all elements defined by the *element specification*.

8.1.1.4.2.10 The element function `same[]`

The element function `same[]` is written as follows:

`same[element specification, left | right, number range]`

The system interprets it as the set of elements which

- have the same name as an element in the set expressed by *element specification* (henceforth referred to as *element*),
- (if the second argument is `left`), appear "in front of" the *element* in the database,
- (if the second argument is `right`), appear "after" the *element* and
- are located at distance from the *element* which is still within the *number range*.

8.1.1.4.2.11 The element function `part[]`

The element function `part[]` is written as follows:

`part[element specification,number range]`

It is defined as the set of all entries defined by *element specification* which appear in their respective aspects in positions within the sequence of entries in the *number range*.

8.1.1.4.2.12 The element function `offspring[]`

The element function `offspring[]` is written as follows:

`offspring[group specification,number range]`

The system interprets it as the set of elements which are contained in the elements of the set defined by the *group specification* (henceforth referred to as *group*) and appear in the sequence of types of element contained in the *group*, in positions within the *number range*.

8.1.1.4.2.13 The element function `each[]`

The element function `each[]` is written as follows:

`each[]`

It is defined as the set of all elements contained in the last group to be activated.

8.1.1.4.2.14 The element function `total[]`

The element function `total[]` is written as follows:

`total[]`

It is defined as the set of all elements which are contained in the last group to be activated and in all groups logically subordinated to the latter.

8.1.1.4.2.15 The element function `core[]`

The element function `core[]` is written as follows:

`core[element specification]`

It is defined by the set of core information items in the entries defined by *element specification*.

8.1.1.4.2.16 The element function `text[]`

The element function `text[]` is written as follows:

```
text[element specification,number | date | category | relation |
        location | image | others]
```

- If only the first argument is specified, this function is defined as the set of all entries of the `text` data type defined by *element specification*.
- If the second argument is `others`, this function is defined as the result of converting each entry within *element specification* to the data type `text`.
- If the second argument is any of the other keywords, this function is the set of the entries of the data type `text` which is the result of converting all entries of the selected data type within *element specification* to `text`.

8.1.1.4.2.17 The element function `date[]`

The element function `date[]` is written as follows:

```
date[element specification,text,definition name]
```

- If only the first argument is specified, this function is defined as the set of all entries of the `date` data type defined by *element specification*.
- Otherwise the function is defined as the set of all entries of the `date` data type which is the result of converting all entries of data type `text` within *element specification* with the help of *definition name*, which must be the name of a `date` definition.

8.1.1.4.2.18 The element function `number[]`

The element function `number[]` is written as follows:

```
number[element specification,text,definition name]
```

- If only the first argument is specified, this function is defined as the set of all entries of the `number` data type defined by *element specification*.
- Otherwise the function is defined as the set of all entries of the `number` data type which is the result of converting all entries of data type `text` within *element specification* with the help of *definition name*, which must be the name of a `number` definition.

8.1.1.4.2.19 The element function `category[]`

The element function `category[]` is written as follows:

```
category[element specification,text,definition name]
```

- If only the first argument is specified, this function is defined as the set of all entries of the `category` data type defined by *element specification*.
- Otherwise the function is defined as the set of all entries of the `category` data type which is the result of converting all entries of data type `text` within *element specification* with the help of *definition name*, which must be the name of a `category` definition.

8.1.1.4.2.20 The element function `relation[]`

The element function `relation[]` is written as follows:

```
relation[element specification]
```

This function is defined as the set of all entries of the `relation` data type defined by *element specification*.

8.1.1.4.2.21 The element function `location[]`

The element function `location[]` is written as follows:

```
location[element specification,text,definition name]
```

- If only the first argument is specified, this function is defined as the set of all entries of the `location` data type defined by *element specification*.
- Otherwise the function is defined as the set of all entries of the `location` data type which is the result of converting all entries of data type `text` within *element specification* with the help of *definition name*, which must be the name of a `location` definition.

8.1.1.4.2.22 The element function `image[]`

The element function `image[]` is written as follows:

```
image[element specification,text,definition name]
```

- If only the first argument is specified, this function is defined as the set of all entries of the `image` data type defined by *element specification*.
- Otherwise the function is defined as the set of all entries of the `image` data type which is the result of converting all entries of data type `text` within *element specification* with the help of *definition name*, which must be the name of an `image` definition.

8.1.1.4.2.23 The element function `lines[]`

The element function `lines[]` is written as follows:

```
lines[number]
```

It is defined as an entry of the `text` data type which causes the system to advance by *number* lines when you are using commands such as `write` or `index` to transfer data to an output medium or device.

8.1.1.4.2.24 The element function `page[]`

The element function `page[]` is written as follows:

```
page[constant]
```

It is defined as an entry of the `text` data type which causes the system to advance by one page, at the top of which *constant* appears on a separate line, when you are using commands such as `write` or `index` to transfer data to an output medium or device.

8.1.1.4.2.25 The element function `cumulation[]`

The element function `cumulation[]` is written as follows:

```
cumulation[element specification | :query[],
    small | large | mean | deviation | number | total,
    without | deviation | generation, without | deviation | generation]
```

It is defined as a entry of the `number` data type, which represents a parameter for the statistical distribution of all entries of the `number` data type covered by the *element specification*.

- If `:query[]` appears as the first argument the system invokes the element specification in the last `cumulation[]` call. If `cumulation[]` is invoked with `:query[]` before a call with *element specification* was made, the system displays an error message.
- You can use the keywords `small`, `large`, `mean`, `deviation`, `number` and `total` to select the minimum, maximum, mean value, standard deviation, number of cases included and the sum of the corresponding distribution respectively.
- You may omit the third and fourth parameters; if you leave them in, they must be consistent. The optional keywords for these parameters have the following effects:
  - `without` instructs the system to exclude all `number` entries expressed as an interval between a minimum and a maximum value from the calculation of statistical classification numbers. (This is how the system proceeds by default.)
  - `deviation`, on the other hand, instructs the system to calculate the mean value from minimum and maximum in such events, and to weight its inclusion in the statistical classification numbers using the following expression:

$$1 - \frac{(maximum - minimum)}{maximum}$$

.

  - `generation` only has consequences if calculations within a network are involved. It evaluates the distances of the individual entries included in the calculation from the group in the database in which the path forming the *element specification* started. In fact, it divides each entry by the square of its distance along the path. This means that an entry appearing in the same group as the original information item is divided by 1; an entry which was reached by making *one* transition via a entry of the `relation` data type is divided by 2; an entry which could only be reached by making *two* such transitions is divided by 4, etc. etc. This option was implemented primarily for the purpose of weighting statistics within genealogical networks.

8.1.1.4.2.26 The element function `codebook[]`

The element function `codebook[]` is written as follows:

```
codebook[element specification-1 | constant | :query[],
        name of variable | element specification-2, definition name]
```

If *name of variable* or *element specification-2* do not contain any further additions, this function is defined as follows:

The set of entries of the `number` data type which are created when the system searches out each entry in turn which can be converted into the `text` data type and which corresponds to *element specification-1* or *constant*, as an identifier in the codebook called *definition name*, and then specifies the numerical value of one entry from each variable which can be derived from *variable name* or the contents – converted into the `text` data type – of each entry in *element specification-2*.

If the *variable name* is followed by one of the two keywords `number` or `text` after a *data signal character 2* (slash) or if *element specification-2* is followed by one of these keywords after the same data signal character, the following definitions apply:

For `number`: the same as if you had not specified a keyword.

For `text`: the set of entries of the `text` data type resulting from the following operation: the system starts by searching out each entry in turn which can be converted into the `text` data type and which corresponds to *element specification-1* or *constant*, as an identifier in the codebook called *definition name* and then retains the numerical value of one entry from each variable which can be derived from *variable name* or the contents – converted into the `text` data type – of each entry in *element specification-2*. Finally, the system searches for the tag corresponding to each of these numerical values in the codebook called *definition name*.

If `:query[]` appears as the first argument, the system attempts to find additional variables in the most recently read specifications for the same codebook. In this case, any attempt to specify the *definition name* again is treated as an error; it is also an error to invoke this function using `:query[]` unless it has first been called by *element specification-1* or *constant*.

8.1.1.4.2.27 The element function `expectation[]`

The element function `expectation[]` is written as follows:

```
expectation[element specification-1 | constant-1,
        element specification-2 | constant-2,
        element specification-3 | constant-3, definition name,
        name of variable | element specification-4, each[] | partial]
```

This built-in function [1] is used to estimate the gap between two points in time about which you only have inadequate information. It is intended primarily as a way of estimating the age of a person at a given moment in time. Because the function can be explained most plausibly with the help of this example, we use a version of it below. However it

[1] This built-in function was developed in collaboration with the Anthropological Institute at Göttingen University, and was defined by Dr. Eckart Voland.

should be noted that this function can also be used with other sequences of interrelated `date` data.

The function always provides you with an entry of the `number` data type which expresses the probable distance in time between *element specification-1* (or *constant-1*) and *element specification-2* (or *constant-2*) in years.

In the following description of the way the function operates, we assume that:

- *element specification-1* or *constant-1* represent a given individual's date of birth, and that
- *element specification-2* or *constant-2* represent the date on which the same individual died.
- *Element specification 3* or *constant 3* represent the date of a "target event", e.g. the date on which this individual was married.
- *Definition name* refers to a codebook which lists the life expectancies of a population classified by sex.
- *Variable name* or *element specification-4* represents an individual's sex. The function calculates the probable age of the individual at the time of the target event: if the date on which the individual was married is unknown, the system prepares a "missing value"; it also prepares a "missing value" if the date of death precedes the target date.

If you only know a *terminus post* or *ante quem* for the target date, the system uses the term you know; if you know both terms for a given interval the system uses the interval's midpoint.

The system completes the two other dates using the following procedures for separate combinations of incomplete information:

date of birth	date of death				
	exactly known	only *terminus ante quem* known	only *terminus post quem* known	both terms known	date unknown
exactly known	1	2	3	8	4
only *terminus ante quem* known	5	6	7	9	6
only *terminus post quem* known	5	7	6	9	6
both terms known	8	2	3	8	4
date unknown	6	6	6	6	6

1) No missing information about dates.
2) The system assumes that the date of death is the lower value of:
   - the indivividual's life expectancy at birth according to sex,
   - the *terminus ante quem* of the individual's date of death.
3) The system assumes that the individual's date of death is equivalent to the life ex-

pectancy according to sex for the age attained by the *terminus post quem* of the individual's date of death.

4) The system assumes that the individual's date of death is equivalent to the individual's life expectancy at birth according to sex.
5) The system uses the known term as the date of birth.
6) The system is unable to supply the missing information for these combinations.
7) In both cases, the system regards the known term as an exact date.
8) The system assumes that the imprecisely known date is the midpoint between both terms.
9) Where the system only knows one term for a date, it uses the latter; if it knows both terms it uses the midpoint between them.

The function's last argument determines the type of result reproduced as a `number` entry:

- `complete` instructs the system to use the exact result of the numerical calculation as an entry.
- `partial` instructs the system to round up the exact result to the nearest whole year.

8.1.1.4.2.28 The element function `order[]`

The element function `order[]` is written as follows:

```
order[]
```

It is defined as an entry of the `number` data type, which represents the ordinal number of the relevant group.

8.1.1.4.2.29 The element function `days[]`

The element function `days[]` is written as follows:

```
days[element specification]
```

It is defined as a set of entries of the `number` data type, which represent the day specifications for all entries of the `date` data type contained in the *element specification*. The system uses a value of zero for entries in which an interval of time is expressed as calendar dates from two different months.

8.1.1.4.2.30 The element function `month[]`

The element function `month[]` is written as follows:

```
month[element specification]
```

It is defined as a set of entries of the `number` data type which represent the month specifications for all entries of the `date` data type contained in the *element specification*. The system uses a value of zero for entries in which an interval of time is expressed as calendar dates from two different years.

8.1.1.4.2.31 The element function `year[]`

The element function `year[]` is written as follows:

```
year[element specification]
```

It is defined as the set of entries of the `number` data type which represent the year specifications for all entries of the `date` data type contained in the *element specification*.

8.1.1.4.2.32 The element function `weekday[]`
The element function `weekday[]` is written as follows:

```
weekday[element specification]
```

It is defined as a set of entries of the `text` data type which represent the weekdays for all entries of the `date` data type contained in the *element specification*. In terms of intervals of time, the weekday is treated as unknown. Entries generated by this built-in function are language-specific. In the English language version, they are reproduced as: `Unknown`, `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday` and `Sunday`. In the German language version, they are reproduced as: `Unbekannt`, `Montag`, `Dienstag`, `Mittwoch`, `Donnerstag`, `Freitag`, `Samstag` and `Sonntag`.

8.1.1.4.2.33 The element function `position[]`
The element function `position[]` is written as follows:

```
position[]
```

It is defined as an entry of the `number` data type, which reproduces the hierarchical distance of the relevant group from document level. The group opening the document has a hierarchical distance of zero; all groups directly contained in this group have a hierarchical distance of one, etc.

8.1.1.4.2.34 The element function `visibility[]`
The element function `visibility[]` is written as follows:

```
visibility[]
```

It is defined as an entry of the `number` data type, which reproduces the visibility (cf. section 4.3.3) of the given group as a numerical value.

8.1.1.4.2.35 The element function `form[]`
The element function `form[]` is written as follows:

```
form[constant, text | date | number | category, definition name]
```

It is defined as an entry appearing at the point in the database reached by the path being processed when the built-in function is activated. This entry's data type corresponds to the one specified by the second argument. The *constant* is converted into the relevant internal format according to the following rules:

- Using the logical object called in by default for the relevant data type if the third argument is missing.
- Using the logical object *definition name* from the class of logical objects corresponding to the data type. The system will generate an error message if such a logical object is not found in the local environment and cannot be loaded from the database being processed when this function is called.

Hint: *so you can prepare* `number` *constants which can be used with* `date` *specifications in joint expressions, the system makes the* `number`*-class logical object* `date` *available by default.* This logical object provides you with language-specific qualifiers with which you can formulate suitable constants.

In the German version, these are:

- `Jahr` or `J` with a numerical value of 365.25,
- `Monat` or `M` with a numerical value of 30.5,
- `Woche` or `W` with a numerical value of 7 and
- `Tag` or `T` with a numerical value of 1.

In the English version, they are:

- `Year` or `Y` with a numerical value of 365.25,
- `Month` or `M` with a numerical value of 30.5,
- `Week` or `W` with a numerical value of 7 and
- `Day` or `D` with a numerical value of 1.

8.1.1.4.2.36 The element function `collect[]`

The element function `collect[]` is written as follows:

```
collect[element specification]
```

It is defined as an entry of the data type required in the given context. This entry contains the specifications of all entries described by *element specification*.

The way the individual entries are combined depends on data type.

- `text` In this case the system strings the entries together, each separated from the next by data signal character 8 (by default: ";").
- `date` In this case, the system generates an interval of time, with the two exact dates or interval boundaries which are furthest apart in all relevant entries, as interval boundaries.
- `number` In this case, the system generates a numerical range, with the two exact numerical data or interval boundaries which are furthest apart in all relevant entries, as interval boundaries
- `category` In this case, the system generates an entry which contains the logical sum of all entries combined. Thus every character held for a particular category appears exactly once.

8.1.1.4.2.37 The element function `target[]`

The element function `target[]` is written as follows:

```
target[element specification,constant]
```

This built-in function is defined as the set of all network transitions from the entries in the *element specification* which contain references to a destination which fulfils the condition for the `relation` data type (cf. section 8.1.2.3.5.1) *constant*. At present, it is *not* possible to write another path instead of *constant*.

8.1.1.4.2.38 The element function `conversion[]`

The element function `conversion[]` is written as follows:

```
conversion[element specification,definition name]
```

It is defined as the set of entries of the `text` data type in the *element specification*, after they have been converted by the conversion definition *definition name*. Cf. section 7.3.1.7 for more information about this class of logical objects.

8.1.1.4.2.39 The element function `before[]`

The element function `before[]` is written as follows:

```
before[element specification]
```

It is defined by a set of entries of the `date` data type which are *always* exact dates.

- Where any `date` entries contained in *element specification* already represent exact dates the system incorporates these dates as they are.
- Where entries which contained intervals of time the system incorporates their respective *termini ante quem*.

8.1.1.4.2.40 The element function `after[]`

The element function `after[]` is written as follows:

```
after[element specification]
```

It is defined by a set of entries of the `date` data type which are *always* exact dates.

- Where any `date` entries contained in *element specification* already represent exact dates the system incorporates these dates as they are.
- Where entries which contained intervals of time the system incorporates their respective *termini ante quem*.

8.1.1.4.2.41 The element function `difference[]`

The element function `difference[]` is written as follows:

```
difference[element specification]
```

It is defined by a set of entries of the `number` data type.

- From all `date` entries contained in *element specification* which represent exact dates the system incorporates zero.
- Where any entries contain intervals of time the system uses the gap in days between the two terms.

8.1.1.4.2.42 The element function `lemma[]`

The element function `lemma[]` is written as follows:

```
lemma[element specification,latin]
```

It is defined by the set of entries of the `text` data type which is created if the set of entries of the same data type specified by *element specification* is lemmatised using the lemmatising procedure described by the second argument.

At the present time, the user can only invoke the lemmatising system used by the Istituto Linguistica Computazionale by specifying the keyword `latin`. This system was developed by Andrea Bozzi; an earlier version has been implemented as a stand-alone program by Giuseppe Cappelli. A fuller description is available on request.

In order to use this system it is important to know:

- the lemma returned by the system is a basic form of the word (nominative singular, first person singular in the present tense, or a non-inflectable form).
- the algorithm supports lemmatising operations, i.e. one original entry may result in several entries, each containing a potential lemma.

8.1.1.4.2.43 The element function `soundex[]`

The element function `soundex[]` is written as follows:

```
soundex[element specification,definition name]
```

It is defined as the set of entries of the `text` data type which are expressed by *element specification* after they have been converted by the soundex definition *definition name*. Cf. section 7.3.1.8 for more information on this class of objects.

8.1.1.4.2.44 The element function `skeleton[]`

The element function `skeleton[]` is written as follows:

```
skeleton[element specification,definition name]
```

It is defined as the set of entries of the `text` data type which are expressed by *element specification* after they have been converted by the skeletonising definition *definition name*. Cf. section 7.3.1.9 for more information on this class of objects.

8.1.1.4.2.45 The element function `query[]`

The element function `query[]` is written as follows:

```
query[]
```

It is defined as the last entry to be reached by `query`.

This function plays a particularly important role when you are processing data from which you are intending to select a very specific entry using `query`. An example follows:

The task

```
query name=example;part=:placename="Bavaria"
write part=:placename
```

is interpreted as follows, in accordance with the general system logic:

“Search the current database for all `placenames` where *at least one entry* contains the character string `Bavaria`. Print out the contents of the *element* `placename` for all such groups.”

The interpretation of your instruction described above is normally perfectly acceptable, but when your data is presented as follows:

```
placename=A-village, Bavaria; B-town, Swabia
```

you will find that you cannot produce a list which only contains locations in Bavaria. The system also interprets the following task:

```
query name=example;part=:placename="Bavaria"
index part=:placename
```

according to the same logic, so that you obtain an alphabetical list of all locations which appear in elements containing the character string `Bavaria` *in at least one entry*: and in this case, also including “`B-town, Swabia`”.

The element `:query[]` provides the solution to this problem:

```
query name=example;part=:placename="Bavaria"
write part=:query[]
```

is interpreted as follows:

"Search the current database for all `placenames` in which *at least one entry* contains the character string `Bavaria`. Print out the contents of all such *entries*."

Accordingly, the following instruction:

```
query name=example;part=:placename="Bavaria"
index part=:query[]
```

will provide you with an index of all *entries* containing the character string `Bavaria`. "`B-town, Swabia`" will *not* appear in this list.

8.1.1.4.2.46 The element function `keyword[]`

The element function `keyword[]` is written as follows:

```
keyword[name,element specification]
```

It is defined as a set of entries of the `text` data type, each containing the name of one catchword to which the *element specification* in the catalogue *name* is linked. Catchwords are the names of lists of references which were generated by the menu option `Link` in the main menu for interactive processing of a catalogue (cf. "Halbgraue Reihe" volume B10, section 2.2.3) or by a `keyword[]` directive with the parameter `bridge=yes` (cf. section 7.3.1.15.1.3) specified in the context of a catalogue definition.

8.1.1.4.2.47 The element function `also[]`

The element function `also[]` is written in one of the two following ways:

```
also[]
```

or

```
also[element specification]
```

- In the first case, it is defined as a set of entries of the `number` data type, each of which represents a view in which the most recently activated group exists.
- In the second case, it is defined as a set of entries of the `number` data type, each of which represents a view in which at least one element of the set of elements defined by *element specification* exists.

8.1.1.4.2.48 The element function `words[]`

The element function `words[]` is written as follows:

```
words[element specification]
```

It is defined as an entry of datatype `number` which represents the number of words (separated by one or more blank characters) contained in *element specification*.

8.1.1.4.2.49 The element function `last[]`

The element function `last[]` is written as follows:

```
last[element specification]
```

It is defined as the last entry within *element specification*.

8.1.1.5 Expressions

Expressions come into being when you use the *expression operator* to append a second path ending in an element specification to the element specification closing a first path. You should also insert a *link operator* between the expression operator and the following path.

This means that expressions have the general form:

$< path - 1 >< expression\ operator >< link\ operator >$
$< path - 2 >$

Expression operators are the "ampersand" (`&`) and the "vertical line" (|). Link operators are variously defined and interpreted for each data type.

The system processes the expression for each entry in *path-1* in turn; it combines the entries in *path-2* to form *a single* entry before performing the requested operation in accordance with the rules described in section 8.1.1.4.2.36.

In practice, of course, you would hardly ever use expression operators to process constants as we do in the following examples: we are using them in this way simply so that it is easy to check the results. The following task illustrates how to use expressions in practice (cf. also section 8.1.1.5.2.1). It instructs the system to output the age of the individuals concerned:

```
query name=source;part=person
write part=:name,:first_name,:date&-:date_of_birth
stop
```

No link operators have been defined for the `relation`, `location` and `image` data types to date.

8.1.1.5.0 Expression operators

The operators defining the way in which two elements shall be combined consist of two parts. The second part, the link operator, defines how the entries shall be connected. The first, the expression operator, defines how missing information shall be handled in the evaluation of an expression.

In both cases expressions are strictly evaluated from left to right. All link operators have the same precedence, bracketing by "`(`" and "`)`" is possible, however.

8.1.1.5.0.1 Expression operator &

When this expression operator is used any expression which contains at least one missing information is considered missing as a whole. If within a given group the element `name` contains "`Smith`", the element `first_name` is missing and the element `occupation` contains "`carpenter`", the expression

```
:name&+:first_name&+:occupation
```

would therefore be considered as missing.

8.1.1.5.0.2 Expression operator |

When this expression operator is used, a missing element is ignored, together with the operator which links it to the previous part of the expression. With the data described in the preceding section, the expression

```
:name|+:first_name|+:occupation
```

would therefore be evaluated as "`Smithcarpenter`".

8.1.1.5.1 Expressions using the `text` data type

If *path-1* is of the `text` data type, *path-2* must also be of the `text` data type.

8.1.1.5.1.1 The `text` link operator '+'

This operator appends the contents of the second path directly to the contents of the first path.

```
:form["Neo-Classical ",text]&+
    :form["architectural style",text]
```

would therefore result in the character string `"Neo-Classical architectural style"`.

8.1.1.5.1.2 The `text` link operator '-'

This operator subtracts the contents of the second path from the contents of the first path.

```
:form["Neo-Classical architectural style",text]&-
    :form["architectural ",text]
```

would therefore result in the character string `"Neo-Classical style"`.

8.1.1.5.1.3 The `text` link operator '*'

This operator retains in the contents of the first path the character string which starts with the first occurrence of the contents of the second path.

```
:form["Neo-Classical architectural style",text]&*
    :form["architectural ",text]
```

would therefore result in the character string `"architectural style"`.

8.1.1.5.1.4 The `text` link operator ':'

This operator retains in the contents of the first path the character string leading up to the start of the string contained in the second path.

```
:form["Neo-Classical architectural style",text]&:
    :form["architectural",text]
```

would therefore result in the character string `"Neo-Classical"`.

8.1.1.5.2 Expressions using the `date` data type

If *path-1* is of the `date` data type, *path-2* must also be of the `date` or `number` data types. In the second case, the system expects *path-2* to represent a number of days.

8.1.1.5.2.1 The `date`–`date` link operator '-'

This operator converts the first path's entries into the `number` data type and reproduces in them the interval in days between the two `date` paths. If one or both of the paths represents an interval of time, the system produces a numerical range which represents the intervals of time between the corresponding interval boundaries. In order to make sure that the minimum limit of the range is less than its maximum limit, the system swaps the two parts round if necessary.

```
:form["13.4.1721",date]&-
    :form["15.4.1721",date]
```

thus results in the numerical value `-2.00`.

```
:form["13.4.1721-16.4.1721",date]&-
    :form["14.4.1721-15.4.1721",date]
```

results in the numerical value `-1.00 - 1.00`.

8.1.1.5.2.2 The `date`–`number` link operator '+'

This operator converts the contents of *path-1* into the `date` specification occurring *path-2* days after the original `date` specification.

```
:form["13.4.1721",date]&+
    :form["1week",number,date]
```

thus results in the calendar date `20.4.1721`.

```
:form["13.4.1721-15.4.1721",date]&+
    :form["1day-1week",number,nodefault]
```

results[2] in the calendar date `14.4.1721 - 22.4.1721`.

8.1.1.5.2.3 The `date`–`number` link operator '-'

This operator converts the contents of *path-1* into the `date` specification occurring *path-2* days before the original `date` specification.

```
:form["13.4.1721",date]&-
    :form["1week",number,date]
```

thus results in the calendar date `6.4.1721`.

```
:form["13.4.1721-15.4.1721",date]&-
    :form["1day-1week",number,nodefault]
```

thus results[3] in the calendar date `6.4.1721 - 14.4.1721`.

[2] This is an *example*. There is no standard `number` definition which calculates intervals in days.

[3] See previous remark.

8.1.1.5.3 Expressions using the `number` data type

If *path-1* is of the `number` data type, *path-2* must also be of the `number` data type.

8.1.1.5.3.1 The `number` link operator '+'

This operator corresponds to the arithmetical operation addition.

```
:form["3",number]&+
    :form["2",number]
```

thus corresponds to the numerical value `5.00`.

```
:form["3-5",number]&+
    :form["2-3",number]
```

corresponds to the numerical value `5.00 - 8.00`.

8.1.1.5.3.2 The `number` link operator '-'

This operator corresponds to the arithmetical operation subtraction.

```
:form["3",number]&-
    :form["2",number]
```

thus corresponds to the numerical value `1.00`.

```
:form["3-5",number]&-
    :form["2-3",number]
```

corresponds to the numerical value `0.00 - 3.00`.

8.1.1.5.3.3 The `number` link operator '*'

This operator corresponds to the arithmetical operation multiplication.

```
:form["3",number]&*
    :form["2",number]
```

thus corresponds to the numerical value `6.00`.

```
:form["3-5",number]&*
    :form["2-3",number]
```

corresponds to the numerical value `6.00 - 15.00`.

8.1.1.5.3.4 The `number` link operator ':'

This operator corresponds to the arithmetical operation division.

```
:form["3",number]&:
    :form["2",number]
```

thus corresponds to the numerical value `1.50`.

```
:form["3-5",number]&:
    :form["2-3",number]
```

corresponds to the numerical value `1.00 - 2.50`.

8.1.1.5.4 Expressions with the `category` data type

If *path-1* is of the `category` data type, *path-2* must also be of the `category` data type.

8.1.1.5.4.1 The `category` link operator '+'

This operator represents the logical total.

```
:form["ab",category]&+
    :form["cd",category]
```

thus corresponds to the series of abbreviations `a, b, c, d`.

8.1.1.5.4.2 The `category` link operator '-'

This operator represents the logical difference.

```
:form["abcd",category]&-
    :form["cd",category]
```

thus corresponds to the series of abbreviations `a, b`.

8.1.1.6 Network transitions

A network transition is expressed by the fact that immediately after an element specification containing entries with the preferred data type `relation`, the path is continued by additional group and/or path definitions.

8.1.1.6.1 Unconditional network transitions

If you note down this element specification without any additions the system proceeds down the section of the path following the group referred to by the network identifiers in the element specification's entries.

Hence the path `person:related:name` starts from the group `person` and goes on to the element `name` in *every* group referred to by a network identifier in one of the entries in the element `related`.

8.1.1.6.2 Conditional network transitions

If, on the other hand, you mention the element specification as the first argument in the element function `target[]` (cf. section 8.1.1.4.2.37), the system only proceeds down the remainder of this path if the element and the group to which the elements refer fulfil certain conditions.

`person:target[:relationship,"debtor"]:name` instructs the system to trace only those network identifiers in the entries in the element `related` which refer to groups named `debtor`.

8.1.1.6.3 Transitions to other databases

In principle, every element of the `relation` data type can refer to any number of different databases. References of this type are declared using the resources described in section 10.

In principle, network transitions which link several databases together have the same syntax as the transitions described above. In order to be able to assess whether the part of a path noted after the transition to another database is correct, $\kappa\lambda\epsilon\iota\omega$ must know the target database for this transition. For this reason, the system always interprets a network transition of the kind described above as an instruction to select only those entries from the total number of available entries in an element which remain in the original database.

If you wish to select transitions to *another* database, you must specify the name of the database to which the transition should be made in a pair of angled brackets immediately after the name of the element in which the transitions are being selected.

To preserve formal consistency you must still do this even if the element being processed only contains transitions to one database.

So if, starting from a database named `source`, you wanted to output information contained in this database as well as information contained in another database named `taxlist`, which can be accessed via an element with the same name, you would set the system the following task:

```
query name=source
write part=:each[],:taxlist<taxlist>:each[],
         :addresses<addresses>:each[]
```

Transitions between databases correspond in every other way to transitions within a database. This means you can also use the conditional form, as shown in the following example.

```
query name=source
write part=:each[],
         :target[:taxlist<taxlist>,"person"]:each[],
         :addresses<addresses>:each[]
```

In order to be selected for processing, the reference in this instance must indicate the group `person` in the database `taxlist`.

8.1.2 Conditions

You can append a condition, prefixed by a data signal character 3, as the very last component at the end of every path; i.e. after the element specification.

8.1.2.1 General

Every condition in $\kappa\lambda\epsilon\iota\omega$ can be reduced to the following form:

$< path >< data\ signal\ character\ 3 >< reference\ pattern >$

Relational operators are in principle constituents of the reference pattern, hence appear to the right of the data signal character.

Every such condition of the named variety can be translated as follows:
"At least one element in the set *path* must fulfil the condition specified by *reference pattern*".

8.1.2.1.1 Logical relationships between conditions

Instead of a condition of the named variety, you can use a *compound condition*, the separate subconditions of which are connected by *logical operators.*

Subconditions may themselves be compound conditions.
The relationship can be expressed in one of the two following ways:

- $< path >< data\ signal\ character\ 3 >< reference\ pattern\ a >$
  $< logical\ operator >$
  $< path >< data\ signal\ character\ 3 >< reference\ pattern\ b >$
  (Type 1)
- $< path >< data\ signal\ character\ 3 >< reference\ pattern\ a >$
  $< logical\ operator >< reference\ pattern\ b >$
  (Type 2)

These two forms are not equivalent.

Type 1 is interpreted as follows: "The condition as a whole is fulfilled if this *logical relationship* between the following subconditions is true: 'At least one element in the set *path* should fulfil the condition laid down by *reference pattern a*' and 'At least one element in the set *path* should fulfil the condition laid down in *reference pattern b*'".

Type 2, on the other hand, is interpreted as follows: "The condition as a whole is fulfilled if at least one element in the set *path* fulfils the condition defined by the *logical relationship* between *reference pattern a* and *reference pattern b.*"

There should be at least one blank space before and after every logical relationship.

8.1.2.1.1.1 Logical operator `and`

If two subconditions are linked together by the keyword `and`, both must be true in order to satisfy the compound condition.

8.1.2.1.1.2 Logical operator `or`

If two subconditions are linked together by the keyword `or`, it is sufficient for one of them to be true in order to satisfy the compound condition.

8.1.2.1.2 Negation of conditions

If the keyword `not` appears in front of a subcondition, this condition is treated as true as long as it is *not* fulfilled. There should be one or more spaces before and after the keyword.

8.1.2.1.3 Bracketing

The following applies by default:

- `or` divides a compound condition if it appears before `and`.
- `not` negates the immediately following subcondition, until the next logical condition.

You can modify both procedures by inserting a pair of parentheses. Parentheses combine compound conditions to form a single subcondition, which can then be logically linked to another subcondition or negated.

8.1.2.2 Reference patterns which do not depend on data type

Certain relation conditions do not take the type of information to be treated into consideration. They relate to one property of a set expressed by a path.

8.1.2.2.1 The reference pattern `null`

If a reference pattern consists of the keyword `null`, no element of the set of elements specified by the path should exist, if the system is to regard the condition for an element in the set of containing groups as fulfilled. It also does so if one of the groups in the path which lead to the element specification, does not exist.

If this reference pattern is negated, at least one element in the set of elements defined by the last component in the path must exist.

8.1.2.3 Reference patterns which depend on data type

Most reference patterns relate to a particular data type. In order to process them, the system only calls those entries from the set defined by the path which possess the required data type.

8.1.2.3.0 Reference patterns

Reference patterns consist of

- one and only one *comparand* and
- (optionally) a number of *comparison modifiers*.

You can specify these components in any order, each separated from the others by one or more spaces.

8.1.2.3.0.1 Comparands

Comparands consist of data of the specified data type. The data either appears in a user-defined constant, or it is taken from other entries.

8.1.2.3.0.1.1 Comparisons using constants

These comparisons have the general form:

$< path - 1 >=< constant >$

If the comparand is expressed as a constant, at least one element in the set defined by the path must fulfil the condition set by the constant.

8.1.2.3.0.1.2 Comparisons using the contents of other entries

Comparisons of this type have the general form:

$< path - 1 >=< path - 2 >$

If you want the comparison to be successful, at least one of the entries contained in *path-1* must fulfil the condition set by at least one of the entries contained in *path-2*.

The system always checks all the entries contained in *path-1*; thus *path-1* can fulfil a condition as many times as it contains entries.

The comparison with the entries in *path-2*, on the other hand, terminates as soon as it is successful: hence each entry in *path-1* can only fulfil the condition at most once.

8.1.2.3.0.2 Comparison modifiers

Comparison modifiers are keywords which must be written in the prevailing upper or lower case. They are specific to the data type in each instance, and modify the standard form of the comparison which is being performed.

8.1.2.3.1 The `text` data type

Reference patterns for the `text` data type check two character strings, to see whether one is contained in the other in the manner specified.

8.1.2.3.1.1 Comparison modifiers

If no comparison modifier is present, the system expects the entries appearing in *path-1* to contain the comparand in any position.

8.1.2.3.1.1.1 Comparison modifier `start`

If the reference pattern contains the keyword `start`, the comparand should appear at the beginning of one of the entries in *path-1*.

8.1.2.3.1.1.2 Comparison modifier `limit`

If the reference pattern contains the keyword `limit`, the comparand should appear at the end of one of the entries in *path-1*.

8.1.2.3.1.1.3 Comparison modifier `equal`

If the reference pattern contains the keyword `equal`, the comparand should exactly match one of the entries in *path-1*. You can formulate this in another way by specifying the keyword `start` together with the keyword `limit`.

8.1.2.3.2 The `date` data type

Reference patterns for the `date` data type check two calendar dates, to see if one is related to the other in the manner specified.

8.1.2.3.2.1 Comparison modifiers

If no comparison modifier is present, the system requires the calendar dates in the entries in *path-1* to match the comparand exactly. However this match only applies to the day, month and year as displayed in the system's own internal format, after conversion.

If the comparand is an interval of time, the condition can only be fulfilled by calendar dates which are also intervals of time with exactly the same *termini post* and *ante quem*.

8.1.2.3.2.1.1 Comparison modifier `equal`

The comparison modifier `equal` demands that in addition to matching, as they should if no comparison modifier were present, all other parts of the calendar date should also match; i.e. the two calendar dates should be written in an identical style, and neither or both of them should be distinguished by question marks or exclamation marks, etc.

8.1.2.3.2.1.2 Comparison modifier `before`

In order to fulfil a condition which includes this comparison modifier[4] , the exact date (or *terminus ante quem*) of an entry from *path-1* must occur before the exact date (or *terminus post quem*) of the comparand.

8.1.2.3.2.1.3 Comparison modifier `after`

In order to fulfil a condition which includes this comparison modifier[5] , the exact date (or *terminus post quem*) of an entry from *path-1* must occur after the exact date (or *terminus ante quem*) of the comparand.

8.1.2.3.2.1.4 Comparison modifiers `before equal`

In order to fulfil a condition which includes these two comparison modifiers[6] , the exact date (or *terminus ante quem*) of an entry from *path-1* must occur before the exact date (or *terminus post quem*) of the comparand, or else be identical to the latter.

8.1.2.3.2.1.5 Comparison modifiers `after equal`

In order to fulfil a condition which includes these two comparison modifiers[7] , the exact date (or *terminus post quem*) of an entry from *path-1* must occur after the exact date (or *terminus ante quem*) of the comparand, or else be identical to the latter.

[4] The system accepts `less` as a synonym.
[5] The system accepts `greater` as a synonym.
[6] The system accepts `less equal` as a synonym.
[7] The system accepts `greater equal` as a synonym.

8.1.2.3.3 The `number` data type

Reference patterns for the `number` data type check two items of numerical data to see whether one relates to the other in the manner specified.

8.1.2.3.3.1 Comparison modifiers

If no comparison modifier is present, the system demands that the numerical data in the entries in *path-1* should exactly match the comparand. However, this match only applies to the numerical value resulting from the conversion into the system's own internal format.

If the comparand is a numerical range the condition can only be fulfilled by data which is equal to the same numerical range, with exactly the same minimum and maximum values.

Hint: *It is almost entirely true to say that the more complex and less common of the following comparison modifiers only ever affect numerical ranges specified in terms of minimum and maximum values. In order make it easier to understand the following definitions, we are going to base them on such numerical values. An exact number may be understood as a numerical range in which the minimum and maximum values are equal.*

8.1.2.3.3.1.1 Comparison modifier `equal`

The comparison modifier `equal` demands that in addition to matching, as they should if no comparison modifier were present, all other components of the numerical data should also match: i.e. both numbers should have the same uncertainty operators and neither or both of them should be distinguished by question marks or exclamation marks, etc.

8.1.2.3.3.1.2 Comparison modifier `circa`

In order to fulfil a condition which includes this comparison modifier, an entry from *path-1* should intersect the comparand by a bare minimum. To do so it is enough if either

- the entry's minimum value is less than or equal to the comparand's maximum value, and simultaneously the entry's maximum value is greater than or equal to the comparand's minimum value,

  or
- the comparand's minimum value is less than or equal to the entry's maximum value and simultaneously the comparand's maximum value is greater than or equal to the entry's minimum value.

8.1.2.3.3.1.3 Comparison modifier `less`

In order to fulfil a condition which includes this comparison modifier, an entry from *path-1* must have a maximum value which is less than the comparand's minimum value.

8.1.2.3.3.1.4 Comparison modifier `greater`

In order to fulfil a condition which includes this comparison modifier, an entry from *path-1* should have a minimum value which is greater than the comparand's maximum value.

8.1.2.3.3.1.5 Comparison modifiers `equal circa`

In order to fulfil a condition which includes these two comparison modifiers, an entry from *path-1* should intersect the comparand rather more narrowly than would be the case if you defined this condition with the `circa` comparison modifier alone. To fulfil this condition, it is enough if either

- the entry's minimum value is less than or equal to the comparand's minimum value and simultaneously the entry's maximum value is greater than or equal to the comparand's maximum value,

  or
- the comparand's minimum value is less than or equal to the entry's minimum value and simultaneously the comparand's maximum value is greater than or equal to the entry's maximum value.

8.1.2.3.3.1.6 Comparison modifiers `equal greater`

In order to fulfil a condition which includes these two comparison modifiers, an entry from *path-1* should have a minimum value greater than or equal to the comparand's maximum value.

8.1.2.3.3.1.7 Comparison modifiers `equal less`

In order to fulfil a condition which includes these two comparison modifiers, an entry from *path-1* should have a maximum value less than or equal to the comparand's minimum value.

8.1.2.3.3.1.8 Comparison modifiers `circa greater`

In order to fulfil a condition which includes these two comparison modifiers, an entry from *path-1* should have a maximum value greater than or equal to the comparand's minimum value.

8.1.2.3.3.1.9 Comparison modifiers `circa less`

In order to fulfil a condition which includes these two comparison modifiers, an entry from *path-1* should have a minimum value less than or equal to the comparand's maximum value.

8.1.2.3.3.1.10 Comparison modifiers `equal circa greater`

In order to fulfil a condition which includes these three comparison modifiers, an entry from *path-1* should have a maximum value greater than or equal to the comparand's maximum value.

8.1.2.3.3.1.11 Comparison modifiers `equal circa less`

In order to fulfil a condition which includes these three comparison modifiers, an entry from *path-1* should have a minimum value less than or equal to the comparand's minimum value.

8.1.2.3.4 The `category` data type

Reference patterns for the `category` data type check two combinations of categorical abbreviations to see whether one contains the other.

8.1.2.3.4.1 Comparison modifiers

If no comparison modifier is present, the system expects the entries which appear in *path-1* to contain the comparand; it ignores the presence or absence of additional categorical abbreviations.

8.1.2.3.4.1.1 Comparison modifier `equal`

This comparison modifier specifies that except for the comparand, no other categorical abbreviations should be included in entries in *path-1*.

8.1.2.3.4.1.2 Comparison modifier `others`

This comparison modifier specifies that in addition to the comparand, at least one other categoric abbreviation should be included in the entries in *path-1*.

8.1.2.3.5 The `relation` data type

Reference patterns for the `relation` data type differ widely, depending on whether you are comparing the data with a *constant* or with *path-2*.

8.1.2.3.5.1 `relation` comparisons with constants

If you are making a comparison with constants, reference patterns for the `relation` data type do not allow you to specify comparison modifiers. In this type of comparison a constant consists of a path in which built-in functions and the names of substitution definitions are illegal (cf. section 7.3.1.11 for more on this), and which ends with the name of an element.

The condition is fulfilled by entries from *path-1* which contain at least one network identifier referring to an element which is correctly described by the given path.

The path given as the constant can be as long as you like. The shortest possible constant of this type thus consists of the name of an element: this means that the exact whereabouts of this element within the database structure is of no importance with respect to fulfilling the condition. The longest possible constant of this type represents the entire path between the document and this particular element.

8.1.2.3.5.2 `relation` comparisons with another path

Comparisons of this type expect a match between two subnetworks in one of the networks of the `relation` data type. (cf. section 4.1.1.1.5 for more about these concepts).

8.1.2.3.5.2.1 Comparison modifiers

The system treats a condition without a comparison modifier as fulfilled if the comparand's subnetwork is completely contained in a subnetwork (described by a entry) in *path-1.* [8]

8.1.2.3.5.2.1.1 Comparison modifier `start`

In order to fulfil a condition which includes this comparison modifier, the comparand's subnetwork must match the beginning of one of subnetworks in *path-1.*

8.1.2.3.5.2.1.2 Comparison modifier `limit`

In order to fulfil a condition which includes this comparison modifier, the comparand's subnetwork must match the end of one of the subnetworks in *path-1.*

8.1.2.3.5.2.1.3 Comparison modifier `equal`

A condition which includes this comparison modifier is fulfilled if the comparand's subnetwork is completely identical to a subnetwork in *path-1.*

You can formulate this condition in another way by specifying `start` and `limit` simultaneously.

8.1.2.3.6 The `location` data type

For purposes of comparison, the system currently treats data of this data type as if it was `text` data. The definitions in section 8.1.2.3.1 also apply in this case.

8.1.2.3.7 The `image` data type

For purposes of comparison, the system currently treats data of this data type as if it was `text` data. The definitions in section 8.1.2.3.1 also apply in this case.

[8] When you set up a database, the subnetworks (assuming they overlap) are also identical by definition. Comparison modifiers only make sense in databases where you have made changes to individual entries deleting single relationships between the elements defining the network. This can be done only with the database editor, deleting individual entries of such an element. This editor is at the time of printing available only with version 6 of $\kappa\lambda\epsilon\iota\omega$. More

8.1.3 Examples

Examples of paths without conditions:

```
person

name

person/father/migration:date

person/father/part[migration,3-4]:comment[:date]

property:each[]

:each[]
```

Examples of paths with conditions:

```
:name="Smith"

:each[]=("Red" or "red") and not "cloth" limit

person:first_name1="Mary" or :first_name2="Anne"

food_supplies:identifier="oats" and query[]/creditor:name="Smith"
```

8.2 Selecting part of a database for processing

Whenever you set κλειω the task of processing a database, this task must relate to a portion of the data held in the database. This means that κλειω must know which database to process.

Once you make a database known to the system, it applies all subsequent analysis instructions to this database until you specify a new one. Unless you explicitly tell the system which part of the database to process, it processes all documents in the order in which they were imported into the database.

8.2.1 The `query` command

The general form of the `query` command is:

NAME=<filename>	**(default: none)**
PART=<pathlist>	**(default: none)**

It describes the parts of the database which the system should process. For this purpose, it specifies a series of paths, each of which is processed in turn. As it proceeds down each path, it remembers the position of each element it finds in the set defined by that path, and starts to execute all the instructions which specify which items of information should be processed. Hence all paths specified in these commands are relative to the paths contained in the `query` command.

If you do not issue an explicit `query` command, the paths of all analysis instructions are relative to the group defining each document.

If a `query` command's paths contain conditions, the system only processes those parts of the database which fulfil these conditions.

8.2.1.1 The `name=` parameter

The `name=` parameter accepts the name of a previously encountered structure definition as a value. This name is used to set up relations with all parts of the database which are required during the various phases of the processing.

If a `query` command does not contain this parameter, the system continues to process the last-named database.

The first task in every κλειω program which is intended for processing a database should therefore include a `query` command with the name of the database in its `name=` parameter.

8.2.1.2 The `part=` parameter

The `part=` parameter expects a list of paths as a value.

The system processes these paths *one after the other*.

8.2.2 Block-structured κλειω tasks

Normally, κλειω tasks consist of a (sometimes implicit) `query` command and one or more analysis instructions.

All analysis instructions are executed for all parts of the database which have been selected by `query`.

In this case we say that the κλειω program consists of an *unnamed block* of commands.

In addition, you have the option of issuing several `query` commands in a *κλειω* program, each of which processes a different part of the database using different analysis instructions.

Every time *κλειω* decides whether a particular part of a database fulfils a condition set within a `query` command, the system checks to see whether an instruction block exists which should only be executed once a positive (or negative) decision has been made. If it finds a suitable block, the system proceeds to process it.

Blocks consists of

- a *block initialisation command*,
  - `confirm` or
  - `negate`,
- any number of *κλειω* instructions and
- an `exit` command.

Blocks may contain other blocks – hence also `query` commands.

The `query` command in each block should appear before any analysis instructions.

8.2.2.1 The `confirm` command

The general form of the `confirm` command is:

NAME=<name of block>	**(MUST be specified!)**
TYPE=Permanent | Temporary	**(Default: Permanent)**

This command appears (a) after a `query` command which includes a condition and before all subsequent analysis commands or (b) after an `exit` command which was used to close a block initialised by `negate` for which no parallel block initialised by `confirm` has yet been declared.

It initialises an instruction block which is only executed if the preceding `query` command's conditions are fulfilled.

8.2.2.1.1 The `name=` parameter

The `name=` parameter accepts any user-defined name as a value, by which the block is then known. This name must be different from the names of all other blocks which have not yet been closed by `exit`.

8.2.2.1.2 The `type=` parameter

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

- `permanent` instructs the system to consider any `query` command contained within the block defined by this `confirm` command to become the target for any `query[]` built-in function encountered thereafter. This is the default behaviour of the system.
- `temporary` asks *κλειω* to let following `query[]` built-in functions refer to the last `query` command encountered before the `confirm` block in question.

8.2.2.2 The `negate` command

The general form of the `negate` command is:

NAME=<name of block> **(MUST be specified!)**
TYPE=Permanent | Temporary **(Default: Permanent)**

The command appears (a) after a `query` command which includes a condition, but in front of any subsequent analysis instruction or (b) after an `exit` command closing a block initialised by `confirm` for which no parallel block initialised by `negate` has yet been declared.

It initialises an instruction block which the system only executes if the preceding `query` command's conditions are not fulfilled.

8.2.2.2.1 The `name=` parameter

The `name=` parameter accepts any user-defined name as a value, by which the block is then known.This name must be different from the names of all other blocks which have not yet been closed by `exit`.

8.2.2.2.2 The `type=` parameter

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

- `permanent` instructs the system to consider any `query` command contained within the block defined by this `negate` command to become the target for any `query[]` built-in function encountered thereafter. This is the default behaviour of the system.
- `temporary` asks *κλειω* to let following `query` built in functions still refer to the last `query` command encountered before the `negate` block in question.

8.2.2.3 The `exit` command

The general form of the `exit` command is:

NAME=<name of block> **(MUST be specified!)**

This instruction closes one or more blocks which have been previously defined by `confirm` or `negate`.

8.2.2.3.1 The `name=` parameter

The `name=` parameter accepts any user-defined name as a value, by which a block which has not yet been closed by `exit` is then known.

`exit` is used to close all blocks which were previously opened by `confirm` or `negate`, in the opposite order to that in which they were initialised, up to and including the block with the same name as the one in the `exit` command's `name=` parameter.

If the system cannot find such a block, the task is invalidated, but any blocks which were closed in error are not opened again.

8.2.3 Examples

Examples of how to select information using simple `query` commands:

```
query name=source

query name=source;part=:identifier

query name=source;part=commodity:identifier

query name=source;part=commodity:identifier="oats" or "barley"
```

Example of a block structure:

```
query name=source;part=person:origin="Aberdeen"
confirm name=aberdonian
          |
    analysis instructions
          |
exit name=aberdonian
negate name=others
query part=:origin="Scotland"
confirm name=scotsmen
          |
    analysis instructions
          |
exit name=scotsmen
negate name=remainder
          |
    analysis instructions
          |
exit name=remainder
exit name=others
```

8.3 Processing the selected information

You process the data selected by `query` by issuing appropriate analysis instructions.

All paths specified in these instructions are relative to the position last reached by `query` within the database. The system finishes processing all these paths before it starts to process the paths specified in `query`.

8.3.1 The `write` command

The general form of the `write` command is:

PART=<pathlist>	**(default: query[]:total[])**
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)**
BRIDge=Yes | No	**(default: Yes)**
IMAGe=Yes | No	**(default: Yes)**
TARGet=Human | Machine	**(default: Human)**

The `write` command is used to print out selected information.

8.3.1.1 The `part=` parameter

The `part=` parameter accepts a list of paths as a value. If it is missing, the system substitutes the path `query[]:total[]`.

8.3.1.2 Controlling the scope of the printout

By default, `write` prints out the maximum possible amount of information extracted from each unit of information addressed by the path, i.e.:

- From each group addressed by `each[]` or `total[]`:
  - the group's name,
  - ordinal number and
  - identifier.
- From each element (every one of which begins on a new line):
  - all entries in all aspects. Every entry begins on a new line.
- From each entry:
  - the core information and
  - the entry's status.
- If the query leads through networks linked by `relation`, the system always issues a message whenever it makes the transition to another group.
- If the data contains information of the `image` data type, this is output to the specified output device. If you issue the `write` command in interactive mode, $\kappa\lambda\epsilon\iota\omega$ enables a platform-specific image processing dialogue.

  This dialogue is available only in version 6 of the system. More specifically, its existence creates the difference between version 5 and 6. It is documented in volume A 22 of this series.

You can limit the scope of the printouts described above by invoking the parameters listed below *either* in a `write` command – whereupon they apply to *all* `write` commands in the task being processed – or in an `options` command (cf. section 7.1.6), in which case they will remain in force until you cancel them explicitly or until you terminate the *κλειω* program.

8.3.1.2.1 The `continue=` parameter

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

- `yes` instructs the system to log every transition to a group using a `relation` information item, hence explicitly restores the system's default setting.
- `no` suppresses this logging function.

8.3.1.2.2 The `more=` parameter

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

- `no` ensures that not more than one entry is printed on each line, and thus restores the system's default setting.
- `yes` instructs the system to join up as many entries as possible on one printed line.

8.3.1.2.3 The `second=` parameter

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

- `yes` instructs the system to print the name of the aspect in front of every secondary aspect ("comment", "original text"), thus explicitly restoring the system's default setting.
- `no` suppresses these identifiers. (So that you can continue to distinguish these aspects from the basic information, they are delimited by the same data signal characters as they were when they were entered – as long as you specify `more=yes` and `cumulate=yes`. Otherwise, the printout displays a series of indentations which do not make a particularly attractive impression.)

8.3.1.2.4 The `cumulate=` 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.

- `no` instructs the system to begin each element on a new line, thus restoring the system's default setting.
- `yes` instructs the system to combine as many elements as possible on each line.

8.3.1.2.5 The `self=` 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.

- `yes` instructs the system to print out each element's name, thus explicitly restoring the system's default setting.
- `no` suppresses this output.

8.3.1.2.6 The `position=` 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.

- `yes` instructs the system to print out the ordinal number and group identifier for every group which is output, thus restoring the system's default setting.
- `no` suppresses this output.

8.3.1.2.7 The `start=` 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.

- `yes` instructs the system to print out the standard information concerning a group, i.e. the group's name at the very least. Whether any additional information is printed out depends on the value assigned to `position=`. Thus this parameter value explicitly restores the system's default setting.
- `no` suppresses all standard information concerning groups, regardless of the value assigned to `position=`.

8.3.1.2.8 The `image=` 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.

- `yes` instructs the system to output any `image` information it encounters to the specified output device, thus explicitly restoring the system's default setting.
- `no` suppresses this output.

8.3.1.2.9 The `bridge=` 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.

- `yes` requests a detailed address to be printed for every entry which points into another database.
- `no` abbreviates this to the name of the database into which the entry points.

8.3.1.3 Dumping contents of a data base into an ASCII file

By default *κλειω* prepares the output from a `write` command to be read by a human. It can also be used, however, to dump a complete database into an ASCII file as a backup or to transfer the data to another computer system. Furthermore, portions of a database can be dumped into a file to be reorganised according to a different structure.

8.3.1.3.1 The `target=` parameter

This parameter expects one of the two keywords `human` or `machine` as a parameter value. Both of them can be abbreviated to their respective first character.

- `human` asks for output to be formated for a human reader, the default action of the system.
- `machine` asks for output formatted as *κλειω* input data.

Irrespective of the value of this parameter the output is directed to the current output unit. In almost all cases it will be advisable, therefore, to specify the `target=` parameter of the `continue` or `stop` command whensoever `target=` ist set to `machine` with the scribe command. On these parameters see chapter 3 of this manual.

8.3.1.4 Examples

Below are some examples of correctly written `write` commands:

```
write

write part=family/child:each[]

write part=:name,:date_of_birth,:birthplace

write part=:name,creditor:each[],query[]:age

write part=poem:verse;self=no
```

8.3.2 The `catalogue` command

The general form of the `catalogue` command is:

NAME= <filename>	**(MUST be specified!)**
PART= <list of paths>	**(MUST be specified!)**
LIMIt=<list of characters>	**(default: ”.,;:?!()’”” ”)**
SIGNs=<list of characters>	**(default: ”.?!”)**
NULL=<list of characters>	**(default: none)**
TYPE=Words | Terms	**(default: Words)**
POSItion=Words | Sentences | Part | Element	**(default: none; can be specified more than once)**
OVERwrite=No | Yes	**(default: No)**
FIRSt=Yes | No	**(default: Yes)**
SECOnd=No | Yes	**(default: No)**
LEMMa=Latin	**(default: none)**
SOUNdex=<definition name>	**(default: none)**
SKELeton=<definition name>	**(default: none)**
ALSO=<character>	**(dfault: none)**
SELF=No | Yes	**(default: No)**
FORM=No | Yes	**(default: No)**
REPEat=Sentences | Part | Element | Back | Root	**(default: none)**
MORE=<number>	**(default: zero)**
CLASsification=<definition name>	**(default: none)**
WITHout=Yes | No	**(default: Yes)**

This command is used to set up an index of locations. As a rule, an index of this type contains a list of all words converted to lower case which are contained in the elements specified by the `part=` parameter. *κλειω* can access any entry on this list instantaneously, without perceptible delay. Lists of this kind are useful for accessing a subset of a database directly, using the `catalogue[]` and `keyword[]` built-in functions (cf. sections 8.1.1.2.2.3 and 8.1.1.2.2.4 of this manual), and also for processing a database interactively, as described in German in “Halbgraue Reihe” volume B10.

8.3.2.1 The `name=` parameter

The `name=` parameter expects a valid filename as a value. *κλειω* sets up a catalogue under this filename and simultaneously enters this name in the library of logical objects assigned to the currently active database. By default, the system assumes that a catalogue with this name does not yet exist. If one does exist the system will refuse to execute the instruction. You must specify this parameter.

8.3.2.2 The `part=` parameter

The `part=` parameter expects a list of paths as a value. You must specify this parameter.

By default, `catalogue` processes the core information of every entry in the “basic information” aspect of every element addressed by the list of paths.

8.3.2.3 The `limit=` parameter

The `limit=` parameter expects a constant as a value. This parameter defines the characters used to signal the end of a word. Here you should also specify those characters which are not to be included in the catalogue. By specifying the relevant numbers as (sub)values of this parameter you can, for example, instruct *κλειω not* to include numbers in the index of locations.

Regardless of how you define the blank space character in this parameter, the system *always* interprets it as a word separator.

8.3.2.4 The `signs=` parameter

The `signs=` parameter expects a constant as a value. It specifies the characters which are used to mark the end of a "sentence". However, this parameter is only effective if you also specify the parameter `position=sentences`.

8.3.2.5 The `type=` parameter

This parameter expects one of the keywords `words` or `element` as a value. Both of them can be abbreviated to their respective first character. It tells the system what kind of entries to make in the catalogue.

- `words` instructs the system to build up the catalogue from individual words, separated by the separator specified in `limit=`. The maximum word length is an installation option; in the system as delivered, it is set to 40 characters. This is how the system creates catalogues by default.
- `terms` instructs the system to include the contents of entry in question, i.e. including any blank spaces in the entry, as an entry in the catalogue. The maximum number of significant characters to be included is an installation option. In the system as delivered, the limit is set to 40 characters.

8.3.2.6 The `position=` parameter

The `position=` parameter expects one of the keywords `words`, `sentences`, `part` or `element` as a parameter, all of which can all be shortened to their respective first character. You can use it to instruct *κλειω* to count down to the position in which the respective catalogue entry is held in the original database with the help of a series of scales, while the system is actually setting up the catalogue. Specifying this parameter means that when you are processing the catalogue in interactive mode you can define the entry's context for printing or search operations very much more precisely. Above all, however, you should be aware that if you specify this parameter several times, you may increase the amount of memory space required to store the catalogue by a very large margin.

The system interprets this parameter's values as follows:

- `words` instructs the system to remember exactly to which word in the text specified by the list of paths a given catalogue entry corresponds. The system always interprets the beginning of a new entry as an integral boundary.
- `sentences` instructs the system to remember exactly to which sentence in the text specified by the list of paths a given catalogue entry corresponds. The system always interprets the beginning of a new entry as an integral boundary.
- `part` instructs the system to remember exactly to which entry in a series of entries described by the list of paths a given catalogue entry corresponds.

- `element` instructs the system to remember exactly to which element in a series of elements described by the list of paths a given catalogue entry corresponds. This system always interprets this kind of context by reference to the original database. Thus "in front of" an element, you will always find the identically named element in the database – even if it was not included in the catalogue.

8.3.2.7 The `overwrite=` parameter

This parameter expects one of the two keywords `yes` or `no` as a parameter value. Both of them can be abbreviated to their respective first character. It determines how the system should proceed if a catalogue with the name specified in the `name=` parameter already exists.

- `no` instructs the system not to execute the instruction if this is the case, i.e. to behave as it would by default.
- `yes` instructs the system to delete all parts of the existing catalogue with this name and then proceed to execute the instruction. This means that all the reference lists which depended on this particular catalogue are also lost.

8.3.2.8 Defining access paths

You can set up four access paths in each catalogue:

a) One access path via complete word forms or terms,
b) one access path via the beginnings of words,
c) one access path via word endings and
d) one access path via forms derived from words or terms obtained from the database.

Of these access paths, a) is created implicitly as soon as you set up access paths b) or c). You must set up at least one of access paths b), c) or d).

By default, access path b) – and hence also implicitly access path a) – are both implemented; the system only sets up the other access paths if you explicitly request it to do so.

8.3.2.8.1 The `first=` parameter

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

- `yes` enables the system to access complete word forms and also the beginnings of word forms in the catalogue. This is the system's default setting.
- `no` means that the system cannot access the beginnings of words.

8.3.2.8.2 The `second=` parameter

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

- `no` means that the system cannot access word endings. This is the default setting.
- `yes` enables the system to access word forms and also the endings of word forms in the catalogue.

8.3.2.8.3 Access paths via derived forms

Because the system assumes that depending on the type of material of which your data is formed, only one derived form of the corresponding entries should be included in a catalogue, you can only specify *one* of the following parameters for each catalogue.

All procedures used to derive forms are transparent to the user, i.e. you are never confronted by the derived forms. However, whenever you instruct the system to access a given word form, it also obtains for you all word forms from which it is possible to derive the same basic form which can be derived from the word form you specified.

8.3.2.8.3.1 The `lemma=` parameter

Currently, the only keyword this parameter accepts is `latin`. This places at your disposal the lemmatising system of the Istituto Linguistica Computazionale, as developed by Dr. Andrea Bozzi, an earlier version of which had been implemented as a stand-alone program by Giuseppe Cappelli. A more detailed description of this lemmatising procedure is available on request.

Every time the system encounters this parameter with this parameter value, it lemmatises the word forms contained in the catalogue. This means that when you are using the catalogue, you can access the lemmata directly. This way of working with catalogues makes particularly good sense when full-text databases containing Latin texts are involved.

8.3.2.8.3.2 The `soundex=` parameter

This parameter accepts as a parameter value a `soundex` definition, which you must define before the next `continue` command at the latest.

We would advise you to set up this directive with the parameter `type=permanent`, so that you can access it at a later date (cf. section 7.3.1.1.3 in this part of the manual).

If you use this parameter while you are creating catalogues, the system will process the material using the specified `soundex` definition, after which it will make it available in the form of a `soundex` code. This means you can also access the respective code values (cf. section 7.3.1.8 in this part of the manual).

8.3.2.8.3.3 The `skeleton=` parameter

This parameter accepts the valid name of a `skeleton` definition as a parameter value, which you must define the next `continue` command at the latest.

You should always set up the `skeleton` definition with the `type=permanent` parameter, so that the system can access it at a later date (cf. section 7.3.1.1.3 in this part of the manual).

This parameter works in much the same way as the `soundex=` parameter, except that the system processes the material using a `skeleton` definition and then makes it available in this form (cf. section 7.3.1.9 in this part of the manual).

8.3.2.8.3.4 Support for external lemmatization

κλειω can also create catalogues from data which have been entered into the database already in a lemmatized form. In such cases another software system has typically been used during data preparation. The system expects such data to have the form

$$< form >< separator >< lemma >$$

where the *separator* is an arbitrary not reserved character, as in the example

```
Quicunque@quicumque in@in Societate@societas nostra@mio-nuestro,
quam@quam Jesu@nomen-proprium ...
```

The separator may be surrounded by space characters; the lemmata have to consist exclusively of characters which are legal within words, they must therefore not contain any character which has been assigned to the `limit=` or `signs=` parameters. By default every form which is not followed by a separator is not considered for building the access path for derived forms. (It is entered into the primary access path, though.)

8.3.2.8.3.4.1 The `also=` parameter

This parameter expects as parameter value a single character which is used as separator. The use of this parameter and the parameters `soundex=`, `skeleton=` and `lemma=` is mutually exclusive.

The parameter defines which character shall be used as separator between form and lemma.

8.3.2.8.3.4.2 The `self=` parameter

This parameter expects one of the two keywords `no` or `yes` as a parameter value. Both of them can be abbreviated to their respective first character. It decides whether forms without explicit lemma shall be considered their own lemma.

- `no` orders the system explicitly not to consider any form without lemma when constructing the access path for derived forms. This is the system default.
- `yes` orders the system to consider a word without lemma to be itself its own lemma. With this parameter value the following example would therefore be equivalent to the one given above:

```
Quicunque@quicumque in Societate@societas nostra@mio-nuestro,
quam Jesu@nomen-proprium ...
```

8.3.2.9 The `form=` parameter

This parameter expects one of the two keywords `yes` or `no` as a parameter value. Both of them can be abbreviated to their respective first character. It tells the system whether to differentiate between upper-case and lower-case characters in the catalogue. This parameter's value establishes the long-term "sensitivity" of the catalogues: it is *not* possible retrospectively to decrease the sensitivity of a catalogue which has been defined on the basis of this decision. (Obviously it is also not possible to modify a catalogue which you set up to ignore this distinction so that it does start to take it into account .)

- `no` instructs the system to ignore the distinction between upper-case and lower-case characters, i.e. to behave as it would by default.
- `yes` instructs the system to preserve the distinction between upper-case and lower-case characters in the catalogue.

8.3.2.10 The scope of the references included in the catalogue

By default, the system includes a reference to either every word or every term in the processed data in the catalogue. Consequently, if one and the same word appears several times in an entry, this entry is represented several times in the relevant catalogue. Also as a consequence, when you are setting up lists of references interactively (cf. "Halgraue Reihe" volume B10, section 2.3.2.2), you may assume that a list of locations contains as many references to a particular entry as that entry contains terms which qualify for inclusion in your list of locations. So if you set up a list of locations defined as *all contexts containing "a" or "b"*, it will contain an entry in which "a" and "b" both appear, twice over.

You can of course modify the way the system behaves when a criterion appears several times in a defined context.

8.3.2.10.1 The `repeat=` parameter

This parameter expects as a parameter value one of the keywords `sentences`, `part`, `element`, `back` or `root`. All of these can be shortened to their respective first character.

It defines which context(s) in a catalogue, or in a list of references derived from that catalogue, should only appear once.

- `sentences` tells the system that only one instance of a word appearing several times in a single sentence should be included in the catalogue. The system includes the first instance of this word in the sentence in the catalogue. In all calculations with respect to comparisons and display output (cf. "Halbrgraue Reihe" volume B10, section 2.3.2.4), the system ignores any further instances of this word in the sentence. When setting up a list of references, the only reference the system takes into consideration for each sentence is the one which was included as the first in the list.
- `part` tells the system that only one instance of a word appearing several times in each entry should be included in the catalogue, or conversely, that every list of references derived from the catalogue should only contain each entry once. The consequences of using this parameter are similar to those described for `sentences`.
- `element` tells the system that only one instance of a word appearing several times in each element should be included in the catalogue, or conversely, that every list of references derived from the catalogue should only contain each element once. The consequences of using this parameter are similar to those described for `sentences`.
- `back` tells the system that only one instance of a word appearing several times in each group should be included in the catalogue, or conversely, that every list of references derived from the catalogue should only contain each group once. The consequences of using this parameter are similar to those described for `sentences`.
- `root` tells the system that only one instance of a word appearing several times in each document should be included in the catalogue, or conversely, that every list of references derived from the catalogue should only contain each document once. The consequences of using this parameter are similar to those described for `sentences`.

8.3.2.10.2 The `more=` parameter

This parameter expects a number as a parameter value. It allows you to define the meaning of the `back` and `root` parameter values for the `repeat=` parameter more precisely.

If it appears in conjunction with `back`, it specifies the distance of the group which the system should use as the context from the group containing the element which is currently being processed. Hence the combination

```
repeat=back;more=3
```

means that each word in every element at a distance of three logical levels or less from a group containing them all together should only be included once in a catalogue. If the document is less than <number> levels away from the element being processed, the system treats the document as the context.

If this parameter appears in conjunction with `root`, it specifies the distance of the group which should be used as the context from the document containing the element which is currently being processed. Hence the combination

```
repeat=root;more=3
```

means that each word in the group which contains the element currently being processed, and which is three levels away from the document containing this element, should only be included once in the catalogue. If the group directly containing the element is itself less than three levels away from the containing document, the system treats the group as the context.

8.3.2.11 The `null=` parameter

The `null=` parameter expects a constant as a value. It specifies characters which are to be completely ignored during the creation of the catalogue. It is typically used to exclude bracketing symbols, editorial question marks and the like.

8.3.2.12 "Classified" catalogues

A catalogue can be created from elements which contain "classified" text, as described in section 7.3.1.19 of this manual. As a result:

- When the catalogue is used with the full text system, the classifications for the selected term will be displayed.
- Within the full text menus additional submenus can be activated, which allow to restrict text searches to words with a specified combination of classification codes.

8.3.2.12.1 The `classification=` parameter

This parameter accepts the valid name of a `classification` definition as a parameter value, which you must define the next `continue` command at the latest.

You should always set up the `classification` definition with the `type=permanent` parameter, so that the system can access it at a later date (cf. section 7.3.1.1.3 in this part of the manual). Once the definition has been created, the system will not allow its deletion or modification as long as a catalogue exists which uses it.

The catalogue will be created in such a way that for each entry within it a record of its class in all independent subclassifications is kept.

8.3.2.12.2 The `without=` parameter

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

- `no` tells the system to accept every word into the catalogue, even if none of the subclassifications in the applicable classification have been specified. This is the default behaviour of the system.
- `yes` restricts the catalogue to those words for which at least *one* class in one of the subclassifications has been selected. Usually this reduces the size of a catalogue dramatically. For *display* all of the text is still available.

8.3.2.13 Examples

Below are some examples of correctly written `catalogue` instructions:

```
catalogue name=vocabulary;part=letter:text

catalogue name=vocabulary;part=letter:text;
        overwrite=yes;signs=".?!;";
        position=words;position=sentences

catalogue name=people;part=:family_name;
        type=terms
catalogue name=vocabulary; part=letter:text;
        first=yes;second=yes;lemma=latin
catalogue name=people;part=:family_name,
        type=terms;first=yes;second=yes;soundex=famname
catalogue name=people;part=:family_name;
        first=yes;second=yes;skeleton=famskeleton
```

8.3.3 The `index` command

The general form of the `index` command is:

PART=<path>	**(MUST be specified!)**
IDENtification=<path>	**(default: root[0]:sign[])**
POSItion=No | Yes	**(default: No)**
FIRSt=Start | Limit	**(default: Start)**
AFTEr=<definition name>	**(default: none)**
LIMIt=<print constant>	**(default: " ")**
SIGNs=<number>	**(default: dependent on data type)**
FORM=Left | Right	**(default: dependent on data type)**
ORDEr=<definition name>	**(default: Order)**
USAGe=<definition name>	**(default: none)**
WRITe=Yes | No	**(default: Yes)**
WITHout=No | Yes	**(default: No)**
PREParation=<definition name>	**(default: none)**
TYPE=Index | Count | Table	**(default: Index)**
SELF=<character string>	**(default: none)**
MAXImum=<number>	**(default: none)**
MINImum=<number>	**(default: none)**
SUBStitution=<character string>	**(default: none)**
CUMUlate=No | Yes	**(default: No)**
MORE=Last | First | Null	**(default: Last)**
REPEat=<character string>	**(default: none)**
CONNect=<comparison filter>	**(default: none)**
SEQUence=<comparison filter>	**(default: none)**
CONTinue=<comparison filter>	**(default: none)**
LINEs=No | Yes | <character>	**(default: No)**
ALSO=<character string>	**(default: none)**
SOURce=<number>	**(default: none)**
ALWAys=<character>	**(default: none)**
ONLY=<character>	**(default: none)**

You can use this command to create sorted lists of characteristics or combinations of characteristics. You can then use these lists in your project, or in order to create indices for publishing. For this reason, we have provided a series of parameters which you can use for the first purpose if you wish, but which are less convincing in that role than they are as tools for creating formally faultless indices.

Unlike most other commands, this command accepts several `part=` parameters. Each of these `part=` parameters – and every `identification=` parameter – defines one *logical column* in the index. Additionly, unless you use `identification=` explicitly to define another location specification or `type=number` to request such a specification for counting purposes –an extra logical column is included in the index, in which the identifer of the document from which the contents of the first logical column were derived, is displayed as a location specification.

The system starts by combining these logical columns into *logical lines* which are then sorted.

The user may use various parameters to ensure that logical columns colliding with logical lines in the index are combined before they are printed out.

The system sorts the index according to the entries requested by the first `part=` parameter. If these are identical for several lines, the system sorts these lines according to the criteria in the second `part=` parameter. If there is still no difference between certain entries, the system calls in additional information until it has been able to sort each entry uniquely.

8.3.3.1 Defining logical columns

You can use the following parameters to define the logical columns into which the index is divided. On every logical line of the index, each logical column contains exactly one entry from the database. If several entries are available for the paths described below, the system constructs one logical line for every possible combination of these entries.

8.3.3.1.1 The `part=` parameter

The `part=` parameter expects a path as a parameter value. The entries in this path are all combined in one logical column in the index.

The system uses the following default settings in order to prepare each separate data type for sorting:

- `text`: the entire contents of every entry are included in the logical column. It is then left-justified and sorted.
- `date`: every entry is sorted in the format *year.month.day*, where the year (with leading zero if necessary) is always four-digit, and the day and month (with leading zeros if necessary) are always two-digit. Calendar dates defined as intervals of time are sorted into exact date specifications according to their *terminus post quem*.
- `number`: the entries – in the standard `number` print format, i.e. to six decimal places – are incorporated in right-justified form into a twelve-digit field. This is a way of ensuring that the system correctly processes small numerical values of the type frequently found in historical sources. If larger numbers occur or uncertainty indicators and/or numerical ranges need to be sorted, you should define a new field width using the `signs=` parameter. If you do so, you should also *explicitly* specify `form=right` (right justification).
- `category`: the system left-justifies these entries, delimits them with commas and sorts them according to the print format defined in the relevant `category` definition.
- `relation`, `location` and `image`: the system converts data items of this data type into their `text`-type representation before inserting them in the index.

The first `part=` parameter in each `index` command has a special significance: the element specified in this parameter *must* exist so that the system can generate an index line. This means that in the case of a database with a widely varied information content, the creation of indices where the element which should appear in the first column only rarely exists, is limited from the start to index lines which are as complete as possible.

There are many occasions, however, on which you will want to include index lines in the index even though the first-mentioned element is missing. If you do want to do this, you should assign the element function `form[]` to the first `part` parameter, with a dummy constant as its argument if necessary (i.e. `:form[""]`). This represents an element specification which – formally speaking – exists in that group, which in turn means that the system can create an index line for every group selected by `query`.

8.3.3.1.2 The `identification=` parameter

This parameter also expects a path as a parameter value. The entries in this path are all combined in one logical column in the index. The only difference between this parameter and the preceding one is that when it comes to defining such logical columns more precisely, a larger number of parameters are available to the system than is the case for columns generated for the `part=` parameter.

The system assumes that the entries specified in this parameter are identifiers suitable for identifying locations, i.e. that as a general rule each entry is not more than twelve characters long.

8.3.3.2 Parameters relating to the index as a whole

Each of the following parameters relates to the appearance of the index as a whole. This means that you can specify them before the first column-defining parameters, and that you only have to do so once.

8.3.3.2.1 The `position=` parameter

The `position=` parameter expects one of the keywords `yes` or `no` as a value.

- `no` instructs the system to output all logical columns for which you did not specify a `signs=` parameter one after the other, separated by a blank space of the character string you assigned to the `limit=` parameter. This is the system's default setting.
- `yes` instructs the system to set up each index line in tabular form. This means that the system attempts to transfer every logical column into a printed column of the same width, beginning at the same print position on every line. The width of these columns is calculated as follows:
  - First, the system reserves thirteen characters for every logical column set up by the `identification=` parameter, or for the system's default identifier.
  - The sum of these column widths is subtracted from the width of the current output line. (defaults: screen output 80 characters per line, printed output 132 characters per line. You can redefine these settings using the `options` command's `signs=` parameter: cf. section 7.1.3.1.2.)
  - The system divides the remainder of the line by the number of logical columns defined in `part=`.
  - Finally, the system subtracts one from this column width, in order to have one print position available as a column separator.

8.3.3.2.2 The `first=` parameter

This parameter expects as a parameter value one of the keywords `start` or `limit`. You can use this parameter to control the order in which the index is sorted.

- `start` explicitly instructs the system to behave as it would be default. The index is sorted in ascending order.
- `limit` instructs the system to sort the index in descending order.

8.3.3.2.3 The `after=` parameter

This parameter expects the name of a `conversion` definition as a parameter value (cf. section 7.3.1.7.)

If you are using complex sets of rules to replace values appearing more than once, or to combine successive lines, or to link together successive but sometimes missing logical columns – in short, if you are using complex index definitions, you may sometimes find that the resultant index lines do not fully live up to your expectations, inasmuch as they contain a construction which results in an inelegant printout, such as lines containing several commas in a row, because they are continuing to display column separators even though there are no values in these columns.

This parameter is designed to help you deal with precisely this situation. The system applies the `conversion` definition you specify to the completed index line, *after* building up the latter in its entirety from the various instructions for the individual logical columns, but immediately *before* printing it out or displaying it.

8.3.3.3 Parameters relating to individual columns

Each of the following parameters helps you to process a single logical column. For this reason, they should only appear after the first `part=` or `identification=` parameter, and each refer to the last parameter of this type to appear.

8.3.3.3.1 Parameters which can be used for any logical column

8.3.3.3.1.1 The `limit=` parameter

This parameter expects a print constant as a parameter value. This constant is output as a piece of text bridging the gap between the logical column for which the parameter is specified and the next logical column. If you specify this parameter for the last logical column in the command, but you did not specify either an `identification=` or a `type=number`, the text is used to introduce the location reference; otherwise it is simply ignored.

8.3.3.3.1.1.1 Print constants

Print constants are a system concept which "has not yet been implemented". In principle, they comprise alphanumeric constants containing sequences which control the output from your printer. In the case of this one parameter the system recognises the following control sequences for the time being:

- `\n` Go to the beginning of the next line.
- `\f` Go to the beginning of the next page.
- `\t` Go to the printer's next tab position.

8.3.3.3.1.2 The `signs=` parameter

This parameter expects a number as a parameter value. This instructs the system to fit the output for this logical column into a printed column with a fixed length of *number*. If an entry is more than *number* characters long, the extra characters are ignored.

If you specify `form=right` *without* specifying `signs=` the system automatically assumes a value for `signs=` of `18`.

8.3.3.3.1.3 The `form=` parameter

This parameter expects one of the keywords `left` or `right` as a value.

- `left` means that the entries are sorted left-justified.
- `right` means that the entries are sorted right-justified and sorted.

8.3.3.3.1.4 The `order=` parameter

This parameter expects as a parameter value the name of a sorting order defined by an `order` definition (cf. section 7.3.1.14).

It instructs the system to sort the logical column specified by the last-encountered `part=` or `identification=` parameter according to the rules of this sorting order.

8.3.3.3.1.5 The `usage=` parameter

This parameter has not yet been implemented.

8.3.3.3.1.6 The `write=` parameter

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

- `yes` instructs the system to output the logical column in the index line. This is the system's default setting.
- `no` instructs the system *not* to output the logical column. This means you can sort the index by information which does not then appear in the output.

8.3.3.3.1.7 The `without=` parameter

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

- `no` instructs the system not only to output the logical column, but to use it for sorting purposes – according to the position in which it appears in the index line. This is the system's default setting.
- `yes` instructs the system to output the information but *not* to use it for sorting.

8.3.3.3.1.8 The `preparation=` parameter

This parameter expects as a parameter value the name of a `conversion` definition.

It instructs the system to apply this `conversion` definition to the contents of the logical column *before* proceeding to sort the latter. This means that this conversion affects both the order in which the printed lines are sorted, and their appearance, whereas the parameter of the same name in the `order` definition's `conversion` directive relates to the sort operation exclusively.

8.3.3.3.2 Parameters for logical columns generated by `part=`

8.3.3.3.2.1 Parameters affecting single logical columns

8.3.3.3.2.1.1 The `type=` parameter

This parameter expects one of the keywords `index`, `count` or `table` as a parameter value. It is used to define the form of the output.

- `index` instructs the system to generate an index by outputting every line for which entries exist in the logical columns – even if this means that several totally identical lines are printed out one after the other. At the same time, if you have not specified an `identification=` parameter, this parameter instructs the system to output the identification of the document which originated the entry in the first logical column on each index line, by default. This is also the system's default setting.
- `count` instructs the system to print out a single line for each combination of values in the preceding logical columns. This line represents the frequency of these combinations. No identifier is output unless requested by `identification=`.
- `table` has not yet been implemented.

8.3.3.3.2.1.2 The `self=` parameter

This parameter expects an alphanumeric constant as a parameter value. This string is entered in the logical column if no entry exists for it, but the system prepares an index line in any case due to the existence of other logical columns. This parameter only affects the output, not the sort. "Empty" logical columns are *always* sorted before any columns for which information does exist in the database.

8.3.3.3.2.1.3 The `maximum=` parameter

This parameter expects as parameter value a number restricting the number of lines of output. If you did not specify a `type=` – or you specified `type=index` – the system outputs a maximum of *number* lines with the same content for each entry in the relevant logical column; if you specified `type=count`, the only lines printed are ones where the frequency of the entry in the logical column does not exceed *number*.

You can use this parameter to prevent the system from producing a list containing several hundred occurrences of *Smith John* when you really want to concentrate on the "exotic" aspects of your material.

8.3.3.3.2.1.4 The `minimum=` parameter

This parameter also expects as a parameter value a number limiting the number of lines of output. In `type=number`, the system only outputs lines where the frequency of the entry in this logical column does not exceed *number*. You can use this parameter to prevent the system from producing a list of articles of clothing and their associated values comprising several hundred index lines when the ones which really interest you are the ones occurring more frequently, hence more typical of their time.

8.3.3.3.2.1.5 Exclusion of text from sorting

In some applications it is necessary to exclude portions of the text from sorting. For the most typical applications there exists the possibility of preparing the data by a `conversion` directive (cf. section 8.3.3.3.1.8 above) or explicitly marking text to be ignored for sorting purposes (cf. section 7.3.1.14.2.2). Beyond that it can be useful to consider only that part of a “logical column” for sorting purposes, which is is in front of some character (e.g. the first full stop). For such purposes two parameters are available.

8.3.3.3.2.1.5.1 The `always=` parameter

This parameter expects a single character as value. What is in front of this character within a logical column will *always* be used for sorting purposes. The remainder of the logical column will be ignored in sorting, if the full stop is immediately followed by one or more space characters.

8.3.3.3.2.1.5.2 The `only=` parameter

This parameter expects a single character as value. It is meaningful only, if specified together with a `always=` parameter. The character specified with `always=` will signal the end of the text to be considered for sorting only then, if it occurs in front of the character assigned to the `only=` parameter.

8.3.3.3.2.2 Parameters affecting logical columns and lines

The following parameters instruct the system to combine the contents of adjacent logical columns to form a single column of output in certain circumstances, or else to combine a series of logical lines to form a single line of output.

8.3.3.3.2.2.1 The `substitution=` parameter

This parameter expects an alphanumeric constant as a parameter value. If the logical column for which you set this parameter displays the same contents on two or more output lines in succession, these contents are replaced by this character string from the second output line onwards.

8.3.3.3.2.2.2 The `cumulate=` parameter

This parameter expects one of the keywords `yes` or `no` as a parameter value. You should only use it if you used the `cumulate=` or `substitution=` parameters for the previous logical column.

- `no` means that the value of the last-defined `substitution=` parameter relates exclusively to the logical column for which it was specified. This is the default setting.
- `yes` means that the `substitution=` parameter affects each column from the one for which it was set to the last column for which `cumulate=` was specified in unbroken sequence in a cumulative fashion: you can use it, for example, for an index to tell the system that where adjacent logical columns include first and family names, both logical columns should be replaced by a single substitution symbol.

8.3.3.3.2.2.3 The `more=` parameter

This parameter expects one of the keywords `last`, `first` or `null` as a parameter value. It is used to specify which of the constants specified in the `limit=` parameter should be output if a `cumulate=` parameter results in the replacement of several logical columns for which `limit=` was specified.

- `last` instructs the system to output the `limit=` parameter for the last logical column – affected by `cumulate=` – on an index line. (This is the system's default setting.)
- `first` instructs the system to output the `limit=` parameter for the first logical column – affected by `cumulate=` – on an index line.
- `null` instructs the system to ignore all `limit=` parameters for the affected logical columns, and to output a blank space instead.

8.3.3.3.2.2.4 The `repeat=` parameter

This parameter expects a constant as a parameter value.

It instructs the system to combine index lines where the preceding logical columns match to form a single index line. This means that the differing contents of the logical columns for which `repeat=` was defined are separated by the constant specified as the parameter value and then appended to each other, but in such a way that each value appearing in this logical column is only used once.

This parameter remains in force, with the constant you specified, until:

a) the system encounters another `repeat=` parameter,
b) the system encounters a `substitution=` parameter which it treats like an extra `repeat=` parameter after the first `repeat=` parameter,
c) the system encounters a `sequence=` parameter.

Following these rules, the parameter `repeat=" and "` will cause the five index lines below:

```
Huber, Julius:  123-12/4; farmer
Huber, Julius:  123-12/4; weaver
Huber, Julius:  321-9/3; innkeeper
Huber, Julius:  321-9/3; weaver
Huber, Julius:  321-9/3; farmer
```

to turn into two index lines:

```
Huber, Julius:  123-12/4; farmer and weaver
Huber, Julius:  321-9/3; innkeeper and weaver and farmer
```

8.3.3.3.3 Parameters for logical columns defined by `identification=`

Identifications shall frequently be sorted in a more complex way than the alphabetic or numeric one: particularly if they contain shelfmarks like “123-4a” which shall be sorted between “123-5” and “123-4”. Additionally it is often desirable to combine lines which are identical except for their identifications into single lines, like merging the two lines

```
Huber, Julius:  123-12/4
Huber, Julius:  123-12/5
```

into the index line:

```
Huber, Julius:  123-12/4 f.
```

This is accomplished by *comparison filters* which serve as parameter value for three parameters (`sequence=`, `connected=` and `continue=`) which provide the services described initially with different degrees of thoroughness. Only *one* of these parameters may be used with a specific `identification=` parameter, as they are mutually exclusive.

Comparison filters comprise a series of format identifiers, i.e. individual characters which specify what kind of count to expect in an identifier, together with separators, i.e. any characters which have not been defined as format identifiers.

The system recognises the following format identifiers:

`0` Count by Arabic ‘page’ numbers; `2` appears before `11`.

`1` Count by Arabic ‘section’ numbers; `2` appears after `11`.

`x` Count by Roman numerals.

`9` Count by Arabic and Roman ‘page’ numbers; Roman numerals appear before Arabic numbers.

`a` Count by letters. Each letter is treated as a section identifier: hence `ab` appears before `c`. The distinction between upper and lower case is ignored.

`A` Count by letters. Each letter is treated as a section identifier: hence `ab` appears before `c`. Upper-case letters are treated as higher-order sections: hence `b` appears before `Aa`.

`b` Count by letters. Each letter is treated as a number: hence `z` appears before `aa`. The distinction between upper and lower case is ignored.

`B` Count by letters. Each letter is treated as a number: hence `z` appears before `aa`. Upper-case letters are treated as higher-order sections: hence `b` appears before `Aa`.

`z` Alphabetical suffix: you can use optional letters to define a preceding numerical element more precisely if required.

If several format identifiers appear in a row (with separators between them if you wish), the system treats them as being in descending order of importance from left to right. If an identifier does not match the comparison filter, the system does not compare it to the preceding one unless it matches the latter precisely. If the system reaches the end of an identifier before it reaches the last format item in the comparison filter, any items which are not present are treated as zero. The system ignores any parts of the identifier which appear after the end of the comparison filter.

Examples:

Counting filter	Applicable to	Not applicable to
`sequence="0-0/0"`	`123-18/4; 123`	`123a-18/4`
`sequence="0z-0a/0"`	`123-18c/4; 123`	`123a-18/4`
`sequence="0z-0z/0"`	`123-18c/4; 123; 123a-18/4`	
`sequence="x 0 a"`	`xvii 12 a; x; x 13`	`24`
`sequence="9"`	`xvii; 23; 125`	`19a`

The system treats matching index lines according to the following rules. In each of the following examples, we use the comparison filter "0-0/0".

8.3.3.3.3.1 The `sequence=` parameter

This parameter is used most oftenly for the purposes indicated at the beginning. Its operation is described by the following rules:

Rule 1: *If two index lines in front of the identifier match, and if both identifiers are also the same, the system only outputs one index line.*

This means that

```
Huber, Julius:  123-12/4
Huber, Julius:  123-12/4
```

turns into the following index line:

```
Huber, Julius:  123-12/4
```

Rule 2: *If two index lines in front of the identifier match, but the two identifiers are not sequential, the system outputs one index line in which the two identifiers are separated by a comma.*

This means that

```
Huber, Julius:  123-12/4
Huber, Julius:  242-3/1
```

turns into the index line:

```
Huber, Julius:  123-12/4, 242-3/1
```

Rule 3: *If two index lines in front of the identifier match and the two identifiers are in sequence, the system outputs one index line with an* `f.` *appearing after the first identifier.*

This means that

```
Huber, Julius:  123-12/4
Huber, Julius:  123-12/5
```

turns into the index line:

```
Huber, Julius:  123-12/4 f.
```

Rule 4: *If several index lines in front of the identifier match and the identifiers are in sequence, the system outputs one index line in which the first identifier is followed by a hyphen, followed in turn by the last identifier in the sequence.*

This means that

```
Huber, Julius:  123-12/4
Huber, Julius:  123-12/5
Huber, Julius:  242-3/1
Huber, Julius:  321-9/3
Huber, Julius:  321-9/4
Huber, Julius:  321-9/5
```

turns into the index lines:

```
Huber, Julius:  123-12/4 f., 242-3/1, 321-9/3 - 321-9/5
Huber, Julius:  321-9/3 - 321-9/5
```

Rule 5: *If several index lines in front of the identifier match but the identifiers do not follow any particular order, the system outputs one index line in which the separate identifiers are linked together by applying the preceding rules, one after the other.*

This means that

```
Huber, Julius:  123-12/4
Huber, Julius:  123-12/5
Huber, Julius:  242-3/1
Huber, Julius:  321-9/3
Huber, Julius:  321-9/4
Huber, Julius:  321-9/5
```

turns into the index line:

```
Huber, Julius:  123-12/4 f., 242-3/1, 321-9/3 - 321-9/5
```

Rule 6: *If several index lines in front of the identifier match and additional logical columns follow after the identifier, these columns are appended to the end of the index line, separated by commas. In addition, the rules described for the* `repeat=` *parameter (cf. section 8.3.3.3.2.2.4) also apply. The separator between each field is a comma, or the value of a* `repeat=` *parameter if you have specified one.*

This means that

```
Huber, Julius:  123-12/4; farmer
Huber, Julius:  123-12/5; weaver
Huber, Julius:  242-3/1; innkeeper
Huber, Julius:  321-9/3; innkeeper
Huber, Julius:  321-9/4; weaver
Huber, Julius:  321-9/5; farmer
```

becomes the index line:

```
Huber, Julius:  123-12/4 f., 242-3/1, 321-9/3 - 321-9/5; farmer,
weaver, innkeeper
```

8.3.3.3.3.2 The `connected=` parameter

This parameter expects a comparison filter, like the `sequence=` parameter. Its action is weaker, though. Using this parameter the data are sorted according to the criterion defined by the comparison filter, that is, according to the logic of a system of shelfmarks. The resulting index lines are, however, *not* combined to form a single line at the end.

8.3.3.3.3.3 The `continue=` parameter

This parameter condenses the resulting line even more strongly than the `sequence=` parameter. If two successive identifications contain at the beginning format elements which agree completely, only the divergent portions are displayed. With this parameter the lines

```
Huber, Julius:  123-12/4; farmer
Huber, Julius:  123-12/5; weaver
Huber, Julius:  242-3/1; innkeeper
Huber, Julius:  321-9/3; innkeeper
Huber, Julius:  321-9/4; weaver
Huber, Julius:  321-9/5; farmer
```

would therefore be reduced to the index line:

```
Huber, Julius:  123-12/4 f., 242-3/1, 321-9/3 - 9/5; farmer,
weaver, innkeeper
```

8.3.3.4 An index generated by several `index` commands

In most cases, a single `index` command is sufficient to create an index. However, there are certain situations where this is not possible. One example of such is provided by an index where for some groups a family name and for others a maiden name must appear in one and the same logical column. In such cases, you can formulate as many successive `index` commands as you like in one and the same *κλειω* task. Because of the way they work, this does not affect the meanings of the individual parameters, but you should note the two following points:

- You should only specify `part=` and `identification=` parameters for the second and subsequent `index` commands. *κλειω* does not actually prevent you from specifying other parameters for these commands, but in general you should avoid doing so: even at a later stage, only very experienced users should risk using others. There is no harm in simply copying the first command's parameters; in fact, this can be advantageous, since it facilitates the deletion of individual commands at a later date.
- The first `part=` / `identification=` parameter in *every* `index` command places an entry in the first logical column of the index; the second of these parameters in *every* `index` command places an entry in the second column etc. The user is responsible for ensuring that this results in columns containing terms which are easy to sort.

For the creation of such registers additional parameters exist. They have two purposes: (a) to clarify the sorting order between whole identical index lines, which shall be sorted differently, depending on which `index` command created them and (b) to differentiate between identical entries within a logical column, again depending on which `index` command created them.

8.3.3.4.1 Sort order of whole index lines

8.3.3.4.1.1 The `lines=` parameter

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

It only makes sense if you use more than one `index` command to set up your indices; otherwise the system simply ignores it.

Where indices have been generated by more than one `index` command, a sort is not usually affected by which `index` command caused which term to be included in which logical column.

- `no` explicitly confirms this default setting.
- `yes` tells the system that two index lines in which the first logical column is regarded as identical should follow one another in the order corresponding to the ordinal number of the `index` command used to include the values in the logical columns. If, in a task in which you used two `index` commands, the first column on two lines matches and you set this parameter to `yes`, these two lines are printed in the same order as the commands which generated them, regardless of the contents of the following columns. So in this case, an index line generated by the first `index` command would appear before a line generated by the second, and so on.

In addition to `yes` or `no`, the `lines=` parameter accepts another value, consisting of any character which has not already been defined as a data signal character.

If you assign a character to the `lines=` parameter, the system uses it to create sort keys, on the basis of the logic described in the following section. You may only use this character as a parameter value if you define a `also=` parameter at the same time.

8.3.3.4.1.2 The `also=` parameter

The `also=` parameter expects a list of characters as a value.

You use it in conjunction with the `lines=` parameter described in the previous section, in order to perform complex sort operations based on terms. For this purpose, the system treats characters in the list of characters comprising this parameter's value as separators between further subdivisions contained in a logical column.

Its use can best be described with the help of an example.

Let us assume we have data in the form:

```
person$cleric=William (of York)
person$owner=William (of Folkstone)
```

We want to create a *κλειω* program capable of sorting clerics before other people with the same name, regardless of the contents of any designations of origin appearing in brackets.

The solution we adopt

```
index part=:cleric;lines=yes
index part=:owner
```

is not wholly successful, because the order of the two `index` commands only becomes relevant to the sort *after the end* of the logical column defined by the `part=` parameter.

By using the formulation

```
index part=:cleric;lines=+;also="("
index part=:owner
```

we achieve the required result, since the order of the `index` commands, which are now "in the middle of the logical column", becomes relevant to the sort.

This is achieved by writing an appropriate sort note at this point in the data, which the system limits by using the special character assigned to the `lines=` parameter. To enable the system to do this, you must make sure that it sorts the relevant logical column using a sorting order where you have assigned the same special character to the `conversion` directive's `form=` parameter as you assigned to the `lines=` parameter.

8.3.3.4.1.3 The `source=` parameter

This parameter expects a number as its parameter value. It provides for a further refinement of the model introduced by the `lines=` parameter.

In very complex applications it may become neccessary to specify explicitly in which order lines being produced by individual `index` commands shall be sorted, if they are otherwise identical. For this purpose an identification number can be assigned to the individual `index` command with this parameter. More than one `index` command can share one and the same number: lines produced by these `index` commands are than assumed to be of identical position in the sorting order.

If this parameter is used with some `index` commands within a *κλειω* task, it is highly recommended to specify it with each such command within this task.

8.3.3.4.2 Sort order of individual columns

The following mechanism provides for a very subtle difference in sorting individual columns, depending on the `index` command from which they are derived. We will first introduce the parameters which implements this mechanism and than explain it with an example.

The careful reader will notice that the problem solved here is (almost) identical to the one which was discussed with the `also=` parameter in section 8.3.3.4.1.2. The difference is, that with `also=` we have to rely on specific characters already present in the input data, while the present mechanism avoids this condition.

8.3.3.4.2.1 The `second=` parameter

This parameter expects a single character as parameter value.

To declare, that logical columns shall be sensitive for sorting according to the `index` command by which the index line containing it has been created, the `second=` parameter has to be specified excatly once. It should be assigned a character, which never occurs at the beginning of a line. This character will be inserted by *κλειω* for internal purposes at the beginning of some columns, but *not* occur in the output. (The character flags a logical column as "sensitive to explicit sorting".)

8.3.3.4.2.2 The `result=` parameter

This parameter expects a single character as parameter value.

The value of this parameter is inserted at the end of a column – *or replaces the first space character it contains* – before this column is compared for sorting purposes. The character does not appear in print.

8.3.3.4.2.3 Explanatory example

Let us assume we have data in the form:

```
person$cleric=William of York
person$owner=William of Folkstone
```

We want to create a *κλειω* program capable of sorting clerics before other people with the same name, regardless of the content of the field after the initial name.

The solution we adopt is:

```
index part=:cleric;secundum=~
index part=:owner;result=0
```

As a result, when the two names occur, the will be compared as

```
William of York
William0of Folkstone
```

so the cleric is correctly sorted before the secular person. (The zero does *not* occur in print.)

8.3.3.5 Examples

Below are some examples of correctly written `index` commands:

```
index part=:identifier;signs=10;
        part=:value;signs=5
```

```
index part=:identifier;type=count;
        part=:value;type=count
```

```
index part=:identifier;cumulate=yes;substitution='---"---';
        limit=" Value:  "
        part=:value;cumulate=yes;substitution='---"---';
        limit=" appears in:  "
```

8.3.4 The `create` command

The general form of the `create` command is:

NAME=<definition name>	**(MUST be specified!)**
PART=<list of paths>	**(MUST be specified!)**
OVERwrite=No | Yes	**(default: No)**
RESUlt=Yes | No	**(default: Yes)**
WRITe=No | Yes	**(default: No)**
IDENtification=<number>	**(default: 0)**
REPEat=No | Yes	**(default: No)**

It is used to generate codebooks which reproduce terms contained in a database. The system generally uses such codebooks to convert databases into files which can be subjected to statistical analysis; they can be coded for this purpose, i.e. provided with code numbers using the linguistic resources described in section 7.3.1.12.

When you are using this command to generate a codebook, you can include two variables for every identifier extracted from the database:

- the variable `kleio` numbers the identifiers included in the codebook in steps of ten, in the alphabetical order of the identifiers.
- the variable `system` is assigned the indicator for "missing values".

If you are supplementing an existing codebook by issuing another `create` command:

- no change is made to entries for identifiers which are already in the codebook.
- the following applies to the variable `kleio`:
  - no attempt is made to reintroduce it into the codebook if the user has removed it.
  - otherwise, a count in steps of ten is introduced for every newly introduced identifier. This count begins with the highest entry which was last made in this variable (multiplied by ten) and follows the alphabetical order of the *newly added* identifiers.
- all other variables for the newly added identifiers are set to the indicator for missing values.

8.3.4.1 The `name=` parameter

This parameter expects a user-defined name as a parameter value. It is used to set up a new codebook or to address one which you want to supplement.

8.3.4.2 The `part=` parameter

This parameter expects a list of paths as a value. The contents of all entries in this path which can be converted into the `text` data type are included in the codebook in the form of one identifier per entry.

8.3.4.3 The `overwrite=` parameter

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

- `no` means that when you set up a new codebook, there should be no codebook of the same name already associated with the database. If a codebook with the specified name does exist the system will refuse to execute the task. This is the system's default setting.
- `yes` instructs the system to destroy any existing codebook with the name you specified before setting up the new one.

8.3.4.4 The `result=` parameter

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

- `yes` instructs the system to try and set up a new codebook. This is the system's default setting.
- `no` instructs *κλειω* to add the new identifiers to an existing codebook. If no codebook with the name you specified exists the operation is aborted.

8.3.4.5 The `write=` parameter

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

- `no` instructs the system to include the extracted identifiers in the codebook with no further messages or prompts to the user. This is the default system setting.
- `yes` instructs *κλειω* to print out a list of the locations of all identifiers imported into the codebook. By default, the location of each identifier is designated by the identification of the document from which the identifier was taken.

8.3.4.6 The `identification=` parameter

This parameter expects a number as a parameter value. It tells the system how many levels away from document level it should search for a group identification with which to designate locations. In this case, the document is represented by zero, all groups contained in the document itself by one, and so on. If a suitable identifier is found in a group which is less than *number* levels away from the document, this group's identifier is used to identify the location instead.

Specifying this parameter implies you have also specified `write=yes`.

8.3.4.7 The `repeat=` parameter

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

- `no` instructs the system to terminate the task after entering the identifiers in the codebook. This is the system's default setting.
- `yes` instructs the system to output a description of the codebook contents in the form of a `describe` command once it has finished processing the command.

8.3.5 The `cumulate` command

The general form of the `cumulate` command is:

PART=<list of paths>	**(MUST be specified!)**
WRITe=<constant>	**(default: none)**
WITHout=Yes | No	**(default: Yes)**
RELAtion=No | Yes	**(default: No)**

It is used to create some simple statistics describing the parameters controlling the distribution of the contents of the entries specified by the `part=` parameter. These include the maximum value, minimum value, mean value, standard deviation and sum total of the distribution in question, together with the number of statistical cases included.

Hint: *this command is intended as a general aid to orientation. κλειω is not a statistics package. In almost every case involving more than the very simplest statistics, it is worth generating a file which can be subjected to statistical analysis by other means.*

8.3.5.1 The `part=` parameter

This parameter expects a list of paths as a parameter value. Every entry of the `number` data type in this list of paths is included in the specified calculations as a statistical case.

8.3.5.2 The `write=` parameter

This parameter expects an alphanumeric constant as a parameter value.

By default, *κλειω* prints out – on suitable output devices – the distribution requested in each case on a new page. If you specify this parameter, the character string you specify is printed out on the first line of each page.

8.3.5.3 The `without=` parameter

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

- `yes` means that in all cases where a `number` data item is expressed as an interval between a minimum and a maximum value, this entry is excluded from the calculation of statistical classification figures. (This is the system's default setting.)
- `no`, on the other hand, means that in such cases, the system calculates the mean value from the minimum and maximum values and weights its inclusion in the statistical classification figures using the following expression: $1 - \frac{(maximum - minimum)}{maximum}$.

8.3.5.4 The `relation=` parameter

This parameter expects one of the keywords `yes` or `no` as a parameter value.
It only has an effect if you are performing calculations over a network.

- `no` instructs the system to ignore an entry's position within a network. This is the system's default setting.
- `yes` evaluates the distance of each entry to be included in the calculation from the group in the database from which the first transition initiated by an element of the `relation` data type took place. Each entry is then divided by the square of its distance along the path. This means that an entry appearing in the same group as the original information item is divided by 1; an entry which was reached by making *one* transition via an element of the `relation` type is divided by 2; one which could only be reached by making *two* transitions is divided by 4, and so on. This option was included primarily for the purpose of weighting statistics in genealogical networks.

8.3.6 Creating statistical data records

$\kappa\lambda\epsilon\iota\omega$ makes two commands available so that the user can set up statistical data records.

The `translation` command defines which files the system should direct the results to; the `case` command defines how a subset of the information contained in a database should be transformed into a statistical case.

This enables the system to produce two kinds of result:

- a *data file* containing fixed-format cases, with (for the most part) numeric codes representing the variables.
- a *command file* containing the necessary commands for processing the data file in one of the two statistics software packages SPSS-X or SAS.

Since the way individual statistical cases are constructed is more important to an understanding of the way data is transferred to a statistics file than the way the surrounding conditions are defined, we shall start by explaining how they are constructed.

8.3.6.1 The `case` command

The general form of the `case` command is:

PART=<path> | <subpath>	**(MUST be specified!)**
MAXImum=<number>	**(default: one)**
REPEat=<number>	**(default: none)**
DATE=DAY | MONTh | YEAR | DATE | WEEKday | DIFFerence | NUMBer | AFTEr | BEFOre | DOUBle	**(default: DAY, MONTh, YEAR, DATE. Can be specified more than once.)**
NUMBer=Small | Large | Mean | Deviation | Number | Total	**(default: Number. Can be specified more than once.)**
SIGNs=<number-as-format>	**(default: numbers: 6.3; character strings: 20. Can be specified more than once.)**
NAME=<character string>	**(default: VARnnnn)**
WRITe=<character string>	**(default: path in part=)**
NULL=<number>	**(default: zero)**
WITHout=<number>	**(default: installation-dependent)**

A file for statistical analysis produced by $\kappa\lambda\epsilon\iota\omega$ consists of *cases* defined as one or more *case types*. Each case contains exactly the same number of variables in every case type: these variables can be analysed statistically.

Each `case` command causes exactly one of these case types to be generated. In many instances, one entry in a $\kappa\lambda\epsilon\iota\omega$ element will only generate a single statistical variable; frequently, however, it will also generate a whole group of such variables. Each of these (groups of) variables, which should all appear in the statistics file, is defined by a `part=` parameter. This group of variables can be described in more detail by subsequent parameters, which each relate to the preceding `part=` parameter.

8.3.6.1.1 Structuring the information

8.3.6.1.1.1 Simple cases

The `case` command causes one statistical case to be defined for every element encountered in the command's first `part=` parameter. If this element is missing from a group selected by the `query` command, the system does not generate a statistical case; if any elements specified in subsequent `part=` parameters are also missing, the system substitutes a *missing value* for the groups of variables to be derived from them.

Thus in the following example:

```
query name=source;part=person
case part=:age;
        part=:codebook[:occupation,status,occupations];
        part=:debts
```

a single case is generated for every person *whose* `age` *is known*, regardless of whether his/her `occupation` and/or `debts` are known.

If you want the system to generate a case for *every* person, you must use `form[]` – as for the `index` command – to ensure that the first element to appear is one which *always* appears, thus:

```
query name=source;part=person
case part=:form["0.0",number];
        part=:age;
        part=:codebook[:occupation,status,occupations];
        part=:debts
```

8.3.6.1.1.1.1 The `maximum=` parameter

If you do not add any further specifications to a `part=` parameter, cases are produced according to the following logic:

The first entry from every set of entries defined by the path mentioned in `part=` *is incorporated into the statistical case.*

The `maximum=` parameter means that the number of entries translated into independent variables or (e.g. in the case of `date`) groups of variables for each element selected in the preceding `part=` parameter, should be the same as the number assigned to `maximum=`.

```
case part=:codebook[:occupation,status,occupations];maximum=4;
        part=:date;date=month;date=year;maximum=2
```

consequently means that four variables are derived from `occupation`, each containing just one entry, and four variables are also derived from `date`, the first two containing the first entry's month and year, the second two containing the second entry's month and year, and so on.

8.3.6.1.1.1.2 The `repeat=` parameter

This parameter, which also expects a number as a parameter value, allows you to use another solution for the same problem. If you specify this parameter, the system only generates as many variables for each case as are necessary to process *one* entry; but it generates up to the <*number*> of cases you specified, each representing a single entry.

```
case part=:codebook[:occupation,status,occupations];repeat=4;
        part=:date;date=month;date=year;maximum=2
```

would consequently mean that the system generates one case for each of the first four entries in `occupation`, while deriving four variables for `date`, the first two containing the first entry's month and year, the second two containing the second entry's month and year.

The system repeats these last four variables for each of the cases derived from `occupation`.

If there are fewer entries than specified in `repeat=`, the system generates fewer statistical cases: i.e. if in our example, fewer than four occupational details appeared for a particular person, fewer cases would be generated.

8.3.6.1.1.1.3 Using `maximum=` to `repeat=` in comparisons

While you can combine these two parameters in any way you like, it is a good idea to keep to the following rules, for reasons related to content:

- `repeat=` only generates as many cases as are actually used; you can set as high a value as you like for this parameter.
- `maximum=` *always* generates the specified number of variables. For this reason, you should only request the number of variables you really need for your purposes: it will make statistical analysis very much more difficult if you create eight occupation variables simply because one person in twenty thousand happens to have eight occupations. In such cases, we recommend one of two solutions:
  - Either use the element function `cumulation[]` to create a variable which represents the number of existing occupational details but only includes the first two or three in the statistics file,
  - or combine (using a second codebook if necessary, regardless of the way you coded the individual occupations) the occupational details which appear into a single value using `collect[]`, which is then replaced by a code representing the particular configuration of the occupations in each case.
- If you use `repeat=` more than once in a `case` command, the individual cases are set up – as in the `index` command – so that the system uses every possible combination of the values of the groups of variables to which you assigned `repeat=` to create an case. Statistically speaking, files of this kind can be problematic, and should be avoided by beginners.

8.3.6.1.1.2 Subpaths in `part=`

As a rule, it is a good idea to use the `query` command to activate the group in `case` tasks which will act as the “case generator”. If the case is supposed to contain a variable which is logically superior to the case-generating group, you should then activate this variable using `root[]` or `back[]`, as shown in the following example:

```
query name=source;part=person
case part=:codebook[:occupation,socialstatus,occupations];
        part=:age;
        part=root[0]:house_no
```

If this is not possible, so if a longer path is specified after `part=`, you may experience the following additional difficulty. According to $\kappa\lambda\epsilon\iota\omega$'s default definitions,

```
query name=source;part=house
case part=apartment/person:age
```

is defined as the set of all entries of all `age` specifications of all `persons` in all `apartments`. According to our definition of the `case` command above, only the first entry in this set is chosen. Thus the second and all subsequent apartments are ignored, together with the people living in them.

To avoid this problem, you can divide a path over several `part` parameters *in the* `case` *command only*. These parameters must appear in strict succession, with nothing separating them except for the `maximum=` or `repeat=` parameters used to specify the maximum number of groups in each case which should be brought in for processing. A `part=` parameter containing an element should appear at the end of this series of `part=` parameters with *subpaths*.

8.3.6.1.1.2.1 Using `maximum=` with subpaths

You use the `maximum=` parameter according to the logic described above.

```
case part=apartment;maximum=2;
        part=person;maximum=3;
        part=:codebook[:occupation,status,occupations];maximum=2
```

would accordingly instruct the system to generate the following statistical variables:

```
Variable 01:  Case number
Variable 02:  Apartment 1, Person 1, Occupation:  Entry 1
Variable 03:  Apartment 1, Person 1, Occupation:  Entry 2
Variable 04:  Apartment 1, Person 2, Occupation:  Entry 1
Variable 05:  Apartment 1, Person 2, Occupation:  Entry 2
Variable 06:  Apartment 1, Person 3, Occupation:  Entry 1
Variable 07:  Apartment 1, Person 3, Occupation:  Entry 2
Variable 08:  Apartment 2, Person 1, Occupation:  Entry 1
Variable 09:  Apartment 2, Person 1, Occupation:  Entry 2
Variable 10:  Apartment 2, Person 2, Occupation:  Entry 1
Variable 11:  Apartment 2, Person 2, Occupation:  Entry 2
Variable 12:  Apartment 2, Person 3, Occupation:  Entry 1
Variable 13:  Apartment 2, Person 3, Occupation:  Entry 2
```

8.3.6.1.1.2.2 Using `repeat=` with subpaths

The `repeat=` parameter offers a rather more dynamic solution to this problem. It instructs the system to begin a new statistical case every time it encounters the relevant group.

```
case part=apartment;maximum=2;
        part=person;maximum=3;repeat=yes;
        part=:codebook[:occupation,status,occupations];maximum=2
```

would accordingly result in the following data structure:

```
Case 1 Variable 01:  Case number
Case 1 Variable 02:  Apartment 1, Person 1, Occupation:  Entry 1
Case 1 Variable 03:  Apartment 1, Person 1, Occupation:  Entry 2
Case 2 Variable 01:  Case number
Case 2 Variable 02:  Apartment 1, Person 2, Occupation:  Entry 1
Case 2 Variable 03:  Apartment 1, Person 2, Occupation:  Entry 2
Case 3 Variable 01:  Case number
Case 3 Variable 02:  Apartment 1, Person 3, Occupation:  Entry 1
Case 3 Variable 03:  Apartment 1, Person 3, Occupation:  Entry 2
Case 4 Variable 01:  Case number
Case 4 Variable 02:  Apartment 2, Person 1, Occupation:  Entry 1
Case 4 Variable 03:  Apartment 2, Person 1, Occupation:  Entry 2
Case 5 Variable 01:  Case number
Case 5 Variable 02:  Apartment 2, Person 2, Occupation:  Entry 1
Case 5 Variable 03:  Apartment 2, Person 2, Occupation:  Entry 2
Case 6 Variable 01:  Case number
Case 6 Variable 02:  Apartment 2, Person 3, Occupation:  Entry 1
Case 6 Variable 03:  Apartment 2, Person 3, Occupation:  Entry 2
```

8.3.6.1.1.3 Using several `case` commands

During every task involving translation, the first variable in each case is reserved for a case number, each counted consecutively.

If you use several `case` commands in a task, the first `part=` parameters in all these commands always relate to the `part=` parameter in the preceding `query` command. Every command of this kind generates its own type of case. If $\kappa\lambda\epsilon\iota\omega$ encounters more than one `case` command, the second variable in all case types is reserved for the case type identifier. $\kappa\lambda\epsilon\iota\omega$ identifies the individual case types by the ordinal number of the relevant `case` command in the $\kappa\lambda\epsilon\iota\omega$ program. In the following example:

```
query name=source;part=family
case part=mother:age; ...
case part=father:age; ...
case part=child;repeat=yes;part=:age; ...
```

three case types are produced:

- Case type 1 for details about the father,
- Case type 2 for details about the mother and
- Case type 3 for details about the children.

If several types of case are generated, we would strongly advise you to include additional aids intended to help structure the resulting data and facilitate subsequent analysis, by instructing the system to output details of each document's ordinal number and/or details about the groups being processed.

8.3.6.1.2 Extracting information

The system derives one or more variables from each entry selected for processing by `part=`. The way this *group of variables* is constructed, and the format chosen for the output, depend on the data type.

8.3.6.1.2.1 The `number` data type

In the case of `number` data, the system only calls in entries in which a precise numerical value – i.e. not just a range between minimum and maximum values – is specified. This value is output to six places before and three places after the decimal point, and the latter is also included in the data. If there are not enough places for the number in front of the decimal point, the system strips off decimal places if possible. If the number is more than ten digits long, it is replaced by ten asterisks.

You can change the selection of information to be output using the `number=` parameter, and change the output format using the `signs=` parameter.

8.3.6.1.2.1.1 The `number=` parameter

This parameter expects as a parameter value one of the keywords `small`, `large`, `mean`, `deviation`, `number` or `total`, all of which can be abbreviated to their respective first character.

It specifies the variables which can be generated from elements of the `number` data type. You can specify this parameter as many times in succession as you need to in order to define the requisite group of variables.

By default the system generates one variable with a precisely known numerical value for every element of this data type. Again by default, the system treats elements for which only minimum and maximum values are known as missing.

More specifically, various keywords are used to instruct the system to include the following items of information in statistical variables:

- `small` instructs the system to output the minimum value if numerical data is inexact, and the exact number if it is exact.
- `large` instructs the system to output the maximum value if the numerical data is inexact, and the exact number if it is exact.
- `mean` instructs the system to output the mean value of minimum and maximum values if the numerical data is inexact and the exact number if it is exact.
- `deviation` instructs the system to output the difference between minimum and maximum values if the numerical data is inexact, or zero if the numerical data is exact.
- `number` instructs the system to output the number if the numerical data is exact, and to identify it as a missing value if it is inexact.
- `total` instructs the system to output the sum of minimum and maximum values if the numerical data is inexact, and the exact number if it is exact.

8.3.6.1.2.1.2 The `signs=` parameter

This parameter accepts any number as a parameter value. The system interprets the part of the number forming an integer as the number of digits appearing before the decimal point; the decimal fraction is interpreted as the number of decimal places required after the decimal point.

```
case part=:age;signs=2;
        part=:wealth
```

consequently instructs the system to output `age` as a two-digit number with no decimal places (and without a decimal point), `wealth` as a number with six places in front of and three places after the decimal point.

If the system encounters a number which cannot be displayed in this format, the format is modified using the procedure described above.

8.3.6.1.2.2 The `date` data type

By default, the system generates a group of three variables for every element of this data type, the first of which represents the day (two digits), the second the month (two digits) and the third the year (four digits) of a precisely known date. By default, the system regards elements for which only a time interval is known as missing.

You can modify the selection of data to be included using the `date=` parameter; you can change the output format using the `signs=` parameter.

8.3.6.1.2.2.1 The `date=` parameter

This parameter expects as a parameter value one of the keywords `day`, `month`, `year`, `date`, `weekday`, `difference`, `number`, `after`, `before` or `double`, all of which can be abbreviated to the *first four* letters of the keyword.

It defines which variables the system should generate from elements of the `date` data type. You can specify this parameter as many times in succession as necessary in order to define the required group of variables. If you specify this parameter once, the system starts by cancelling the default setting: the only data included in the statistical variables is what you request explicitly.

More specifically, various keywords are used to instruct the system to include the following items of information in statistical variables:

- Selecting separate items of information for the date:
  - `day` selects the day of the month (default output format: two-digit).
  - `month` selects the month (default output format: two-digit).
  - `year` selects the year (default output format: four-digit).
  - `weekday` selects the day of the week, in the following form: 1 = Monday, 2 = Tuesday, ..., 7 = Sunday (default output format: single digit).
  - `difference` selects the interval in days between the *terminus post* and *ante quem* (default output format: six-digit).
  - `number` selects the number of days which have passed since the first day of the first month of year one up to the day in question (default output format: six-digit).

- Choosing individual date formats:
  - `date` instructs the system only to use the selected information for dates which are known exactly.
  - `after` instructs the system to output the selected details for the *terminus post quem* where intervals of time are involved; or in the case of exact dates, to output details of the exact date.
  - `before` instructs the system to output the selected details for the *terminus ante quem* where intervals of time are involved; or in the case of exact dates, to output details of the exact date.
  - `double` represents a shortened form of `date=after;date=before`

  The following example:

```
case part=:date;date=year;date=double;
        part=:birthdate
```

hence generates an case which only contains the year, but does so for both date components for the element `:date`; for the element `:birthdate` on the other hand, it contains the default selection of day, month and year.

8.3.6.1.2.2.2 The `signs=` parameter

This parameter expects a number as a parameter value, which acts as a format specification.

The system interprets the part of this number which is an integer as the number of digits you wish to place in front of the decimal point; the part of the number following the (optional) decimal point as the number of decimal places required.

In the case of the `date` data type, *all* derived variables are always output in the given format if you specify this parameter. Because the default formats are generally speaking more memory-efficient, we advise you not to use this parameter for `date` data.

8.3.6.1.2.3 The `text` data type

If the system encounters `text` data in the element – which have not been transformed into `number` data by a codebook – the first twenty characters of the contents are incorporated into the statistics file as a character string by default, in the format of the platform in use.

You can use the `signs=` parameter to change this setting.

8.3.6.1.2.3.1 The `signs=` parameter

This parameter expects a number as a parameter value.

The part of the number which is an integer specifies the maximum number of characters in the preceding element which should be incorporated into the statistics file. Any data appearing after a decimal point is ignored:

```
case part=:name;signs=40;
        part=:firstname
```

hence means that forty characters from `:name` and twenty characters from `:firstname` should be included in the statistics file.

8.3.6.1.2.4 The `category` data type

The way this data is then processed for statistical purposes is regulated by the `category` definition you used to convert it. By appending a `number=` parameter to the `signs` directive, you can declare an explicit numerical value for each character, e.g. as follows:

```
item name=religion;usage=category
signs sign=c;write="Catholic";number=6
signs sign=p;write="Protestant";number=3
exit name=religion
```

If you do not use this parameter, the system generates the numerical values by counting up the individual `signs` directives beginning with one. If you had not specified the `number=` parameter in the example above, the values would have come out as `c = 1` and `p = 2`.

The system generates exactly the right number of variables for every `category` entry to ensure that

- every group of mutually exclusive characters defined by `part` is depicted in one variable and
- every character which can be combined with any other character, in any order, is depicted in its own variable.

We would like to point out that once you have established the linkage to a database, you can use a newly introduced `category` definition with the appropriate name to replace the temporary `category` definition held in the database. *But you when you do so, you must ensure that the individual characters are introduced in the same order as in the original definition.*

So in the following example, the system would ignore the `category` definition you used to set up the database:

```
query name=source;part=person
item name=religion;usage=category
part
signs sign=c;number=6
signs sign=p;number=3
exit name=religion
case part=:religion
```

The system uses a currently unmodifiable output format for `category` data: each derived variable is displayed with as many digits as necessary to represent the largest `number` value which appears in the codebook definition.

8.3.6.1.2.5 Other data types

At present, the system converts all other data types into the `text` data type before transforming them into statistical values.

8.3.6.1.3 The treatment of missing values

By default:

- the system replaces a variable representing a either a missing entry or element by zero.
- the “missing value” from a codebook is passed on directly to the statistics file.

8.3.6.1.3.1 The `null=` parameter

The `null=` parameter, which expects a number as its parameter value can be used to indicate either missing entries or elements with any other numerical value.

8.3.6.1.3.2 The `without=` parameter

The `without=` parameter, which expects a number as a parameter value, can be used to replace all missing values imported from a codebook with any other numerical value.

8.3.6.1.3.3 Example

```
case part=:codebook[:occupation,status,occupations];
          without=999;null=999;
       part=:codebook[:origin,region,placenames]
```

would consequently instruct the system to display any uncoded identifiers in the codebook as well as any missing entries as 999 in the case of `occupation`, but to restore the default settings in order to process `origin`.

8.3.6.1.4 Describing output in more detail

In the command file, the system identifies the different variables generated for the statistical software packages by assigning each variable a name formed from the prefix `VAR` and a four digit number in sequence.

The system also generates a command for every such variable, providing the variable with a label which explains its contents in more detail. By default, this label consists of the path which generated this variable, together with a suffix which specifies the kind of variable involved.

8.3.6.1.4.1 The `name=` parameter

This parameter expects as a parameter value a constant. When this parameter is specified, the constant is used to generate all variable names, which are derived from the element specified by `part`. If more than one such variable is created (as for example with `date` data, appropriate suffixes are added to the constant specified by the user.

So, the `case` command:

```
case part=marriage/child:dateofbirth;name="bdate"
```

will lead to the creation of the variable names `bdatepd`, `bdatepm` and `bdatepy`.

The suffixes are assigned as follows:

Data type `number`: A one character suffix, describing what kind of numerical information is put into the variable. The suffix is identical with the first character of the keyword with which the value is selected by the `number=` parameter.

`n`	*number* (the default selection)
`s`	*small* (or minimum)
`l`	*large* (or maximum)
`m`	*mean*
`d`	*difference*
`t`	*total*

Data type `date`: A two character suffix, describing which of the two calendar terms forming a $\kappa\lambda\epsilon\iota\omega$ date and which kind of date information is put into the variable. If no calendar term is selected, the suffix for the "earliest possible date" or *terminus post quem* is generated.

First part of suffix:

`p`	terminus *post* quem or "earliest possible date".
`a`	terminus *ante* quem or "last possible date".

Second part of suffix:

`d`	*day* (part of the default selection)
`m`	*month* (part of the default selection)
`y`	*year* (part of the default selection)
`w`	*weekday*
`i`	*interval* (requested by *difference*)
`n`	*number* of days

Data type `text`: No suffixes created.

Data type `category`: For each variable created from one element of this data type a one-character-suffix in alphabetical order, starting with the letter "A" is created.

8.3.6.1.4.2 The `write=` parameter

This parameter expects as a parameter value a constant. It replaces the path definition used to generate a label for the variable in question.

```
case part=:birthdate;name="dob";part=:dod
```

would thus mean that the system would assign the following tags to the variables derived from the element `birthdate` in the target statistical package (the example shows labels generated for SPSS-X):

```
VAR LABEL VAR0001 "case_no"
VAR LABEL VAR0002 "dob - day"
VAR LABEL VAR0003 "dob - month"
VAR LABEL VAR0004 "dob - year"
VAR LABEL VAR0005 ":dod - day"
VAR LABEL VAR0006 ":dod - month"
VAR LABEL VAR0007 ":dod - year"
```

8.3.6.2 The `translation` command

The general form of the `translation` command is:

FIRSt=<filename>	**(default: name of database + ”dat”)**
SECOnd=<filename>	**(default: name of database + ”dic”)**
OVERwrite=No | Yes	**(default: No)**
TARGet=SPSs | SAS | PCSpss | CENssys	**(default: SPSS)**
WRITe=Yes | No	**(default: Yes)**
CODEbook=<number>	**(default: 100)**
LIMIt=<number>	**(default: 100)**
DATE=DAY | MONTh | YEAR | DATE | WEEKday | DIFFerence | NUMBer | AFTEr | BEFOre | DOUBle	**(default: DAY, MONTh, YEAR, DATE. Can be specified more than once.)**
NUMBer=Small | Large | Mean | Deviation | Number | Total	**(default: Number. Can be specified more than once.)**
SIGNs=<number-as-format>	**(default: numbers: 6.3; character strings: 20. Can be specified more than once.)**
NULL=<number>	**(default: zero)**
WITHout=<number>	**(default: installation-dependent)**
MAXImum=<number>	**(default: one)**

This command describes the general framework of conditions in which a statistics file is generated. Each parameter has a default setting, so you can omit any or all of them if you wish.

A `translation` command controls subsequent `case` commands; if you need to, you can specify it as many times as you wish within a given task in order to impose different default settings on separate groups of `case` commands. In this case, every parameter value declared by this command remains in force until you explicitly cancel it. Hence both the following tasks would produce the same result:

```
query name=source; ...
translation first="statistics";second="commands";date=year
case ...
translation date=day;date=month;date=year;date=double
case ...
```

and

```
query name=source; ...
translation first="statistics";second="commands";date=year
case ...
translation first="statistics";second="commands";
        date=day;date=month;date=year;date=double
case ...
```

8.3.6.2.1 Defining the result files

By default, in the course of a task involving the creation of a file for statistical analysis, *κλειω* generates:

- A data file, composed of the database's name and the suffix ".dat". The name of the database is abbreviated as necessary so that it conforms to the maximum number of characters permitted on the platform in question; the suffix appears as a platform-specific filename extension: hence as a file extension under MS-DOS, as a file type under VM/CMS.
- A command file consisting of the name of the database and the suffix ".dic" (for data *dic*tionary). The database name is abbreviated so that it does not exceed the maximum number of characters permitted on the platform in question; the suffix appears as a platform-specific filename extension: so as a file extension under MS-DOS, as a file type under VM/CMS.
- If either the data file or command file already exist on the computer, the task is aborted.
- The command file contains commands for the statistical software package SPSS-X; the command file's suffix is `sps`.

You can change these settings using a preliminary group of parameters.

8.3.6.2.1.1 The `first=` parameter

This parameter expects a constant of any length as its definition. The system interprets this as the name of the data file you want to create.

The name may contain information about directories and suchlike in a form acceptable to your particular operating system.

Example for an MS-DOS system:

```
translation first="D:\stat\data.001"
```

8.3.6.2.1.2 The `second=` parameter

This parameter expects a constant of any length as its definition. The system interprets this as the name of the command file you want to create.

The name can contain information about directories and suchlike in a form acceptable to your particular operating system.

Example for an MS-DOS system:

```
translation second="D:\stat\format.001"
```

8.3.6.2.1.3 The `overwrite=` parameter

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

- `no` prevents the system from overwriting an existing file with the same name as the data or command file you specified, i.e. to behave as it would by default.
- `yes` allows the system to overwrite such files.

8.3.6.2.1.4 The `target=` parameter

This parameter expects as a parameter value one of the keywords `spss`, `pcspss`, `sas`, `censsys` or `null`. All of these can be shortened to the first *three* characters of each keyword.

- `spss` means that commands written to the command file are compatible with the SPSS-X statistical software package. A file handle statement for the input file is generated. The external filename produced for the input file may need editing on some mainframes, though. This is the system's default setting.
- `pcspss` means that commands written to the command file are compatible with the PC version of the SPSS statistical software package. A number of restrictions are taken care of: so only one missing value will be generated per variable; in codebook oriented processing this means that the separate missing value otherwise generated for terms in the database which are not in the codebook is *not* declared to be missing. The commands are separated by blank lines instead of being closed by full stops. Only one case type is allowed.
- `sas` means that commands written to the command file are compatible with the SAS statistical software package.
- `censsys` means that a subset of an SPSS data definition is produced, which is accepted by the CensSys tabulation program. Exactly two case types are required, which are assumed to be hierarchically related. Length restrictions on labels are relaxed.
- `null` suppresses the creation of the command file.

8.3.6.2.2 Defining other conditions

Regardless of the format you choose for the files to be generated, the following applies:

- $\kappa\lambda\epsilon\iota\omega$ represents the structure of the generated file in the form of a summary report.
- the system aborts the execution of a translation task if more than 100 entries of the `text` data type, which are supposed to be converted by a codebook, do not appear in the codebook in question and
- the system aborts the execution of a translation task if it finds more than 100 data items which it is unable to display in the specified format, hence must represent as asterisks in the data file.

These settings can be controlled by the following parameters

8.3.6.2.2.1 The `write=` parameter

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

- `yes` instructs the system to print out the file description generated by default.
- `no` instructs it not to do so.

8.3.6.2.2.2 The `codebook=` parameter

This parameter expects a number as a parameter value. It is interpreted as the maximum permitted number of codebook errors, i.e. of identifiers which cannot be found in the codebook in question.

8.3.6.2.2.3 The `limit=` parameter

This parameter expects a number as a parameter value. It is interpreted as the maximum permitted number of format errors, i.e. of values which cannot be displayed in the format in question.

8.3.6.2.3 Defining default settings for the `case` command

You can use the following parameters to specify a series of default settings for the `part=` parameter in all `case` commands which are *not* followed by this command's identically named parameters. If you used a `translation` command to specify one of these parameters earlier on, it remains as a default setting in the $\kappa\lambda\epsilon\iota\omega$ task in question until you explicitly cancel it by redefining this parameter in a new `translation` command.

8.3.6.2.3.1 The `number=` parameter

This parameter expects as a parameter value one of the keywords `small`, `large`, `mean`, `deviation`, `number` or `total`, all of which can be abbreviated to their initial character.

It tells the system which variables to generate in all the `part=` parameters of all `case` commands following this `translation` command from elements of the `number` data type, if the `case` command itself does not include a `number=` parameter. You can specify this parameter as many times in succession as you like in order to define the requisite group of variables. The meanings of the various keywords are explained in the detailed description in section 8.3.6.1.2.1.1.

8.3.6.2.3.2 The `date=` parameter

This parameter expects as a parameter value one of the keywords `day`, `month`, `year`, `date`, `weekday`, `difference`, `number`, `after`, `before` or `double`, all of which can be shortened to the *first four* letters of each keyword.

It instructs the system which variables to generate in all `case` commands' `part=` parameters, following this `translation` command, from elements of the `date` data type if the `case` command itself does not contain a `date=` parameter. You can specify this parameter as many times in succession as you like in order to define the requisite group of variables. The meanings of the various keywords are explained in the detailed description in section 8.3.6.1.2.2.1.

8.3.6.2.3.3 The `signs=` parameter

This parameter expects a number as a parameter value. The system interprets it as a format specification.

This parameter defines the standard output format. It may appear twice in a `translation` command.

If it appears *in front of* a `number=` parameter, it defines the default settings for the format of entries of the `text` data type. In this case, the system interprets the parameter value as described in section 8.3.6.1.2.3.1.

If it appears *after* a `number=` parameter, the system treats this format as the default setting for all variables derived from `number` elements. In this case, the parameter value is interpreted as described in section 8.3.6.1.2.1.2.

8.3.6.2.3.4 The `null=` parameter

This parameter expects a number as a parameter value. It specifies the numerical value which the system should use to display missing entries of the type described in section 8.3.6.1.4.1.

8.3.6.2.3.5 The `without=` parameter

This parameter expects a number as a parameter value. It specifies the numerical value which the system should use in a codebook to display uncoded identifiers of the type described in section 8.3.6.1.4.2.

8.3.6.2.3.6 The `maximum=` parameter

This parameter allows you to define a default value for complete paths and subpaths. The meaning of these definitions is discussed in sections 8.3.6.1.1.1.1 and 8.3.6.1.1.2.

Careful! If you use this parameter without some thought, the resultant files may become extremely large.

8.3.7 The `mapping` command

The general form of the `mapping` command is:

PART=<list of paths>	**(default: none)**
ALWAys=No | Yes	**(default: No)**
NAME=<name of group>	**(default: none)**
EACH=No | Yes	**(default: No)**
COLOur=Contrast | Red | Blue | Green	**(default: Contrast)**
LINEs=Simple | Double | Treble	**(default: Simple)**
USAGe=Solid | Halftone | Tenthtone | Empty	**(default: Solid)**
SYMBol=Circle | Square | Triangle	**(default: Circle)**
SIZE=<number>	**(default: device dependent)**
SIGN=Yes | No	**(default: Yes)**
NUMBer=No | Yes	**(default: No)**
WRITe=<character string>	**(default: none)**
FIRSt=<coordinate pair>	**(default: none)**
SECOnd=<coordinate pair>	**(default: none)**
TARGet=<file name>	**(default: see below)**
OVERwrite=No | Yes	**(default: No)**
WARNings=Yes | No	**(default: Yes)**

It is used to create maps.

A `mapping` command has two functions: First it selects a series of topographical objects to be displayed from a `location` definition, second, it defines the graphical attributes which these topographical objects should have on the map.

Topographical objects are selected for processing either by the fact that entries in their `location` format are included in the map from the database, or because a global instruction is issued to the effect that particular groups of objects contained in a `location` definition should be displayed.

All topographical objects selected for processing by a `mapping` command have the same graphical attributes: hence while it is sufficient to formulate a single `mapping` command in order to produce a map, it is normally necessary to issue several of them in order to display different classes of topographical object in various ways.

8.3.7.1 How several `mapping` commands interact

The way these different commands interact, best illustrated with the help of the following example, is determined by the following rules:

- You should only use `write=`, `first=` and `second=` parameters in the first `mapping` command.
- The `sign=`, `number=`, `colour=`, `usage=` and `lines=` parameters are all valid for each of the topographical objects selected by the `mapping` command in which they appear.
- If you use the `always=` and `each=` parameters in several `mapping` commands, only the parameter encountered in the last command is valid.

All of which brings us to the following example:

```
note All persons belonging to an abstract group are selected
note from the database -- e.g.  because they have a particular
note occupation.
query name=source;part=person:codebook[:occupation,status,occupations]=
        greater "299" and less "400"
note
note The map as a whole is labelled; all buildings which have not been
note selected by other commands are included on the map with their
note default attributes but without labels.
mapping write="Crafts and the Church";each=yes;sign=no
note
note Churches and monasteries are drawn on the map in green
mapping name=churches,monasteries;colour=green;colour=green;
        usage=solid;usage=tenthtone
note
note Craftsmen's homes are drawn in red
mapping part=root[0]:house_no;colour=red
```

8.3.7.2 Default settings for `mapping` commands

By default, every map you create using `mapping` contains

- all topographical objects from the `location` definition in which you specified a `always=yes` parameter and
- all topographical objects appearing in at least one of the entries addressed by `part=`.
- all topographical objects selected appear with the graphical characteristics defined for them in the `location` definition.
- the map expands to fill the drawing area as far as possible, and
- it is not labelled.
- if an output device has been defined by the `location` command, which leads to the creation of a file, that file is given a platform dependent name.
- if a file with that name exists already, the execution is aborted.
- for each topographical object in the entries addressed by `part=`, which cannot be found in the associated `location` definition, a warning message is generated.

8.3.7.2.1 Selecting topographical objects

8.3.7.2.1.1 The `part=` parameter

The `part=` parameter accepts a list of paths as a parameter value.

It selects all entries of the `location` data type addressed by that list for display on the map.

If the list of paths contains elements which relate to more than one `location` definition the system attempts to apply all `mapping` commands to every `location` definition addressed. This results in a series of maps, each one representing a `location` definition.

8.3.7.2.1.2 The `always=` parameter

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

- `yes` instructing the system to include all topographical objects in the `location` definition with the parameter `always=yes` on the map, hence explicitly instructing the system to behave as it would by default.
- `no` prevents the system from including these objects on the map.

8.3.7.2.1.3 The `name=` parameter

This parameter expects a value consisting of a list of comma-delimited names, each of which appeared in the `name=` parameter of at least one `location` directive in the relevant `location` definition.

It instructs the system to include all topographical objects for which these names were specified on the map.

8.3.7.2.1.4 The `each=` parameter

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

- `no` instructs the system not to display any topographical objects which are not explicitly included in the map by any of the other parameters listed in section 8.3.7.2.1. This is the system's default setting.
- `yes` instructs the system to include all topographical objects of the above type on the map.

8.3.7.2.2 Displaying topographical objects

8.3.7.2.2.1 The `colour=` parameter

This parameter accepts the keywords `contrast`, `red`, `blue` and `green` as values. All of these can be abbreviated to their respective first character.

It defines a colour for the topographical objects selected by the `mapping` command in which it appears. This colour may differ from the colour specified in the corresponding `location` definition. You can specify this parameter as often as you like.

The first `colour=` parameter you specify determines the colour of the topographical objects' contours. The second, which only applies to closed topographical objects, determines the colour of the fill. The third `colour=` parameter determines the colour of any secondary contours, i.e. of the first inner contour(s); the fourth parameter determines the colour of the area(s) enclosed by these contours, and so on.

The parameter values are interpreted as described in section 7.3.1.13.1.2.1.

8.3.7.2.2.2 The `lines=` parameter

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

It determines the thickness of the contours of the topographical objects selected by the `mapping` command in question. This may differ from the value specified in the corresponding `location` definition.

The parameter values are interpreted in the manner described in section 7.3.1.13.1.2.2.

8.3.7.2.2.3 The `usage=` parameter

This parameter accepts the keywords `solid`, `halftone`, `tenthtone` and `empty` as parameter values. All of these can all be abbreviated to their respective first character.

It specifies a pattern for the topographical objects selected by the `mapping` command. This may differ from the pattern specified in the corresponding `location` definition. You can specify this parameter as often as you like.

The first `usage=` parameter you specify defines a pattern for the contours of the topographical objects. The second, which only applies to closed topographical objects, determines the area fill pattern. The third `usage=` parameter defines a pattern for secondary contours, i.e. the first inner contour(s); the fourth parameter defines a pattern for the area(s) enclosed by that contour, and so on.

The system interprets parameter values as described in section 7.3.1.13.1.2.3.

8.3.7.2.2.4 The `sign=` parameter

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

- `yes` instructs the system to label the selected topographical objects, regardless of the stipulations in the corresponding `location` definition.
- `no` suppresses these labels.

8.3.7.2.2.5 The `number=` parameter

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

- `yes` instructs the system to specify how many times the topographical objects selected by `part=` appeared in the database.
- `no` prevents this, i.e. explicitly restores the system's default setting.

8.3.7.2.2.6 The `symbol=` parameter

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

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

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

8.3.7.2.2.7 The `size=` parameter

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

8.3.7.2.2.8 The `cumulate=` parameter

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

- `no` instructs the system to display a chosen value of the `symbol=` parameter always at the size specified by the `size=` parameter. This is the default behaviour of the system.
- `yes` displays the symbols depending on the frequency with which the topographical object occurs in the database. A topographical object occurring exactly once in the database will be displayed by the size defined by the `size=` parameter. Other topographical objects will be displayed in sizes where the displayed area is the logarithm to base ten of the observed frequency. That means: a frequency of ten is reflected by a symbol which fills twice the area of the symbol being displayed for an object occuring once, a frequency of hundred is reflected by a symbol which fills three times the basic area and so on.

8.3.7.2.3 The appearance of the completed map

8.3.7.2.3.1 The `write=` parameter

This parameter expects as a parameter value a constant which appears as a centred caption beneath the map generated, e.g. as follows:

```
          write="Craftsmen's homes"
```

8.3.7.2.3.2 The `first=` parameter

This parameter expects as a parameter value two numbers separated by a comma.

The first number defines how far to the right of the left-hand edge of the drawing surface the map should start; the second specifies how far along the x-axis the drawing should extend.

8.3.7.2.3.3 The `second=` parameter

This parameter expects as a parameter value two numbers separated by a comma.

The first number defines how far below the top edge of the drawing surface the map should start; the second specifies how far along the y-axis the drawing should extend.

8.3.7.2.4 Files for preparing hardcopy

8.3.7.2.4.1 The `target=` parameter

If, with the help of the `location` command, a plotting device has been selected, which asks for the output of the `mapping` command to be stored in a file, the name of that file will depend on the system you are using. The most important types are:

- `kleio.plot` on all types of UNIX systems.
- `kleio.plt` on DOS systems.
- `kleio plot` on VM/CMS systems.

The `target=` parameter allows the user to override this filename. It expects as a parameter value a constant which is a file name according to the conventions of the specific operating system. Directory names, file types and the like can be specified up to any necessary length.

8.3.7.2.4.2 The `overwrite=` parameter

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

- `no` instructs the system to protect the file which shall contain the output of the `mapping` commands against accidental overwriting. This applies to the default output file as well as to a file the name of which has been specified with the help of the `target=` parameter. This is the default behaviour of the system.
- `yes` allows the system to overwrite an existing file.

8.3.7.2.5 General options

8.3.7.2.5.1 The `warnings=` parameter

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

- `yes` instructs the system to issue a warning message for each topographical object which becomes eligible for display, because it is included in the `part=` parameter of the `mapping` command, cannot be found in the associated `location` definition, however. This is the default behaviour of the system.
- `no` suppresses these messages.

9. Working with several databases

For purposes of record linkage in particular, i.e. so that you can integrate information held in several independent databases, *κλειω* also allows you to work with several databases simultaneously.

Three problems must be surmounted in order to do this effectively:

- The system must be able to identify the parts of two databases which you want to link together. This section describes the additional conventions available in the `query` and `write` commands which allow it to do so, and which go far beyond anything you can do with these commands in any one single database.
- You need to connect the items of information in separate databases together. You do so by setting up a link of the `relation` data type, defined in the following section (section 10). The current version of *κλειω* is *not* yet capable of transferring information physically from one database to another.
- The user must also be able to instruct the system that in addition to information items held in one database, it should use information items in another database which have been explicitly linked to the former. Such links are set up using the network transitions described in section 8.1.1.6.3.

9.1 Paths when you are using several databases simultaneously

Everything we have said so far about the use of paths in tasks which refer to *a single* database also applies to tasks which refer to several databases.

9.1.1 Based paths

You can explicitly relate *any* path to a particular database by prefixing it with a *basis*. Since this particular option is redundant in tasks which only refer to one database, it is normally omitted from them. The formal definition of a *based path* looks like this: $< data\ signal\ character\ 6 >< name\ of\ database >< data\ signal\ character\ 7 >< path >$

If you are using several databases in one task, the following rule applies: *any non-based path refers to the last database to be addressed.* This preserves the rules regarding the relativity of paths. In an analysis command, for example, the first path always refers to the database addressed by the element which terminated the first path after the `query` command's `part=` parameter.

In practice, it is certainly very much easier – in tasks which refer to more than one database – to base *all* paths rather than attempt to remember which rules are currently active.

Every database name used in a based path must first appear in the `query` command's `name=` or `also=` parameters (cf. section 9.2.1.1).

9.1.2 Managing logical objects

If you use logical objects in tasks which refer to more than one database, you can – and should – *base* their names as well; i.e. give their names as follows:

$< data\ signal\ character\ 6 >< name\ of\ data\ base >< data\ signal\ character\ 7 >$
$< name\ of\ definition >$

9.1.2.1 Restrictions in practice

At present, *κλειω* transforms logical objects addressed in both databases' permanent environments into *one single* local environment. The user is currently responsible for ensuring that logical objects with the same name have been defined in the same way in both databases. If logical objects with the same name are used in both databases, the system automatically uses the first object to be addressed in the course of the task.

9.2 Commands which refer to several databases at the same time

Currently, you can use the `query` and `write` commands to process several files simultaneously.

9.2.1 The `query` command

The `query` command is used to associate files with one another. For this purpose, one database is processed in the same way as in any other task. In addition, however, for every entry in the first database selected by the appropriate `part=` parameter, the system attempts to find all those entries in the second database which may refer to the information mentioned in the first database. These two entries in both databases are described as *linking entries*.

All the entries in *database 2* which could refer to one and the same entry in *database 1* are described as this entry's *domain* in *database 2.*

A `query` command relating to two databases starts by searching in order for the entry which is being addressed in *database 1*, and then searches in order for all entries in its domain in *database 2.* Every time it finds one of these entries, the system starts processing the evaluating commands (`write` etc.) in the given task.

In order to perform this operation:

- the `query` command must specify which databases are to be linked together *before* the `part=` parameter, and
- you must specify which path in *database 2* should form the domain for the path in *database 1.*

9.2.1.1 The `also=` parameter

`query` commands referring to two databases require an additional parameter, which has the general form:

ALSO=<name of database> (MUST be specified!)

This parameter specifies the database in which the system should search for the domain of the entries in the database specified in the `name=` parameter.

9.2.1.2 Setting up relationships between two databases

9.2.1.2.1 Setting up one database's domain in another

Paths in `query` tasks which are supposed to set up a linkage between two databases *must always* start with the built-in function `catalogue[]` (cf. section 8.1.1.2.2.3).

- The first argument in this built-in function represents a catalogue from the database specified in the `also=` parameter. If no `also=` parameter has been specified, the same construction can be used to create a "linkage of the database to itself".
- The second argument selects the access path in the traditional way.
- The third argument describes an access path in the database specified in the `name=` parameter. In this case, the catalogue entries for the individual entries in the database specified in `name=` form these entries' domain in the database specified in `also=`.

9.2.1.2.2 Limitations of the chosen domain

Once the `catalogue[]` function has been invoked as specified, you can use conditions (which can be as complex as you like) for the same construction as in `query` commands, i.e relating to individual databases.

The `catalogue[]` function corresponds to a number of entries in the database specified in `also=`.

After `negate` and `confirm`, you can set additional conditions which can be as complex as you like.

9.2.1.3 Example

For example, a formally correct `query` command for linking together two databases would look like this:

```
query name=linkage2;also=linkage1;
      part=catalogue[<linkage1>person2,algorithm,<linkage2>:name]
      :firstname=<linkage2>:firstname
```

9.2.2 The `write` command

In tasks relating to two databases, the `write` command has an additional, optional parameter which has the general form:

ONLY=No | Yes (default: No)

9.2.2.1 Alternating data output from several databases

Following a `write` command's `part` parameter in a task relating to more than one database, you can specify alternating paths relating to both databases in any order you choose.

By default, the system processes the *entire* path for every entry in the domain of all entries found for the file specified in `name=`.

9.2.2.1.1 The `only=` parameter

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

- `no` causes the system to process the entire path specified in `part` for every entry in the domain of all entries found for the database specified in `name=`, i.e. to behave as it would by default.
- `yes` means that the system only outputs the paths relating to the database specified in `name=` at the beginning of the information covering each domain of this kind, whereas for every entry in the domain it only processes the paths found in the database specified in `also=`.

9.2.2.2 Example

For example, a formally correct `write` command relating to two databases would look like this:

```
write part=<linkage2>:each[],<linkage1>:each[]
```

10. Linking any number of databases together

The `relation` data type can represent references between any number of databases. In the first instance, you can set up references in one of two ways:

- On the one hand, you may already have *associated* two databases; i.e. they may contain identification numbers which refer to one another. In this case you enter the identification numbers as `text` information items while you are setting up the separate databases, and then use the `bridge` command to convert them into data of the `relation` data type, each referring to the other.
- On the other hand you can also link data together on the basis of other considerations, for example, based on the analysis of the relationships between two databases described in the previous section. You use individual `bridge` directives to define these point-to-point connections. Each directive creates a link between two points in the two databases.

10.1 Linking databases together: the `bridge` definition

Let us assume we have two sources, both of which have been prepared as *κλειω* databases. The first of them contains details of a series of people, including their taxpayers' reference numbers:

```
database name=source;first=person;overwrite=yes
part name=person;position=id,name,first_name,occupation,tax_ref_no
exit name=source
read name=source
person$document-1/Maier/Friedrich/tailor/a-123.c
person$document-2/Schrusser/Hans/doctor/a-146.d
person$document-3/Wilkog/Peter/farmer/a-234.a
person$document-4/Russen/Anna/widow/a-242.c
person$document-5/Bohlen/Paul/blacksmith/a-278.b
person$document-6/Ahrens/Johann/farmer/a-321.c
```

The second source, a tax list, shows the amount of tax paid for every taxpayer's reference number on the list, thus:

```
database name=tax_list;first=b;overwrite=yes
part name=taxpayer;position=id,tax_payment
exit name=tax_list
read name=tax_list
b$a-123.c/1fl 4x
b$a-146.d/5fl
b$a-234.a/2fl 12x
b$a-242.c/8x
b$a-278.b/3fl 20x
b$a-321.c/2fl 14x
```

In this example, one source contains items of information which are included in the second source as document identifications. From a formal point of view:

- the groups to be linked in one source are "absolutely addressable". These groups either open each document in question, or else they were introduced by a `part` directive in which you specified `identification=yes`.
- the other source contains elements of the `text` data type, which contain the identifications of the groups in the first source.

10.1.1 The `bridge` command

In this situation, you can use a `bridge` command to link the two databases.

This has the general form:

FIRSt=<filename>	**(MUST be specified!)**
SECOnd=<filename>	**(MUST be specified!)**
PART=<list of paths>	**(default: none)**
TYPE=Definitions | Confirmations	**(default: none)**
RELAtion=<name of definition>	**(default: reciprocal filename)**
BRIDge=<name of element>	**(default: reciprocal filename)**

Note: you should only define one of the two parameters part= or type=!

10.1.1.1 Links between associated databases

The following `bridge` command is required in order to link the two sources in our example:

```
bridge first=source;part=:tax_ref_no;second=tax_list
```

The three parameters used expect the following values:

- The `first=` parameter expects as a value the name of the database containing details of the link between the two databases in the form of the contents of one or more elements.
- The `part=` parameter expects as a value a list of paths with the names of all the elements which this `bridge` command should use in order to set up a link. The data element in question should refer to the database specified in `first=`.
- The `second=` parameter expects as a value the name of the database containing the character strings held in the entries defined by `part=` as identifications.

10.1.1.1.1 Properties of a standard link

This command is used to set up a linkage between the two databases in the following way:

An element named `tax_list` is introduced in the database named `source`. Ideally, this element's data type is `relation`, which refers to the `relation` definition `tax_list`. If you use this element in an evaluation command, the system assumes that it should be processed as a `relation` data item.

The above-mentioned `relation` definition is placed in the `source` database: an annotation is made to the effect that it is setting up a linkage to the `tax_list` database.

In every group in the `source` database containing an entry which can be linked to an identifier in the `tax_list` database, an entry is made for the `tax_list` element referring to the group in the `tax_list` database to whose identifier the group in the `source` database can be linked.

Exactly the same data is entered in the `tax_list` database in each case, but in such a way that it sets up a linkage to the `source` database. So once the `bridge` command has been executed, an element with the name `source` and the preferred data type `relation` – relating to the `source relation` definition – is also known in the `tax_list` database. In every group whose identifier can be linked to an entry in `tax_list`, an entry is made in the `tax_list` element which refers to the corresponding group in this database.

If you assign the name of one and the same database to both parameters, `first=` and `second=`, a linkage is constructed within this database which does not differ in any way from the linkages of the `relation` data type defined while the data was being input.

10.1.1.1.2 Modifying the standard linkage

The names of `relation` definitions and elements generated in this way can be changed by defining the appropriate `relation=` and `bridge=` parameters. Both parameters can only be defined after the first appearance of the `first=` or `second=` parameter. In this case, they both relate to the one of these two parameters which was last encountered. These may not be specifed more than twice.

- `relation=` is used to specify the name of the `relation` definition in the last database to be addressed.
- `bridge=` is used to define the name of the element in the last database to be addressed. Hence the `bridge` command

  ```
  bridge first=source;part=:tax_ref_num;bridge=linkage;relation=tax;
          second=tax_list;bridge=person;relation=connections
  ```

  would have the following result:
- A new `linkage` element is incorporated into the `source` database which has the `relation` data type as its preferred data type and uses the `relation` definition `connections`.

10.1.1.1.3 Exceptions

You can use the two parameters we have just explained in order to set up the linkage between two databases using names which are easier to remember or interpret. Otherwise they are only important if you want to link databases with names identical to those of the elements or `relation` definitions contained in them at the present moment.

In this case (if you are not using `bridge=` or `relation=`), the following applies:

If a `bridge` command is used to introduce an existing element, a note is added to this element, drawing attention to the fact that entries of the `relation` data type are related to the `relation` definitions with the corresponding name. *However, these elements still contain the data type which has been preferred up to now. If you want to use them as* `relation` *data, you should therefore use the* `relation[]` *function if appropriate.*

If an existing `relation` linkage is addressed by the linkage between the two databases, it is supplemented in such a way that elements referring to it can also refer to the newly linked database.

10.1.1.2 Linking databases which have not been associated

Linkages between databases can also be made regardless of the information items contained in them. Let us assume that the first source we used as an example is to be linked to another source, e.g. an address list:

```
database name=addresses;first=person;overwrite=yes
part name=person;position=ser.no.,name,firstname,address
exit name=addresses
read name=addresses
person$1/Ahrens/Johann/Kirchengasse 27
person$2/Bohlen/Paul/Hauptplatz 12
person$3/Maier/Friedrich/Hauersteig 14
person$4/Russen/Anna/Kirchengasse 18
person$5/Schrusser/Hans/Am Freithof 2
person$6/Wilkog/Peter/Vor dem Berge 39
```

In this case, the `part=` parameter in the `bridge` command must first be replaced by the `type=` parameter with a value of `definitions`. This indicates that the linkages between the two databases should be made by means of explicit definitions, for example, thus:

```
bridge first=source;second=addresses;type=definitions
```

Written in this way, the `bridge` command opens a definition with the same name; the command is followed by `bridge` directives which each define one linkage between two groups in the databases concerned and are closed by an `exit` directive containing the keyword `definitions` in its `name=` parameter. So we can set up six linkages, written something like this:

```
bridge first=source;second=addresses;type=definitions
bridge
bridge
bridge
bridge
bridge
bridge
exit name=definitions
```

10.1.1.2.1 The `bridge` directive

These directives have the general form:

FIRSt=<symbolic address>	**(MUST be specified!)**
SECOnd=<symbolic address>	**(MUST be specified!)**
ONLY=<number>	**(default: none)**
VISIbility=<number>	**(default: none)**

10.1.1.2.1.1 Basic format of a `bridge` directive

In most cases, however, only two of these four parameters are ever used. `first=` and `second=` both include a constant – i.e. between quotation marks or inverted commas – consisting of a definition stating where in the particular database the linkage should be set up. In its simplest form, a "symbolic address" of this kind consists of a document identifier without any additional information. So, continuing with our example, in order to create a linkage between `document-1` in the `source` database (containing basic information about Mr. Friedrich Maier) and his address in document `3` in the `addresses` database, we require the following `bridge` directive:

```
bridge first="document-1";second="3"
```

From this, we obtain the following `bridge` definition to connect our two small databases:

```
bridge first=source;second=addresses;type=definitions
bridge first="document-1";second="3"
bridge first="document-2";second="5"
bridge first="document-3";second="6"
bridge first="document-4";second="4"
bridge first="document-5";second="2"
bridge first="document-6";second="1"
exit name=definitions
```

If you write the `bridge` command, opening up the `bridge` definition, in this way, it makes no practical difference whether you refer to a database in `first=` or `second=`. It is important, however, that all the documents quoted in the `bridge` *directives* following the `first=` parameter really do relate to the database quoted in `first=` in the `bridge` *command*; the same applies to the `second=` parameter of the command or of the directives.

10.1.1.2.1.2 Symbolic addresses

Symbolic addresses are a platform-independent means of defining *where* a particular group is in a database. Since you can use a document's identification to give access to the group initialising this document, the document identifier is at the same time the document's symbolic address and the simplest kind of symbolic address there is. We have already given some examples of how it can be used.

However, because you can only use this particular method to access a document as a whole, the element representing such a symbolic address is also always present in the group initialising the document. For this reason, as a general rule you should always specify addresses more precisely in structured databases.

In the following examples, we will work with the database fragment given below:

```
database name=census;
        first=house
part name=house;
        position=housenumber;
        part=flat
part name=flat;
        position=flatnumber;
        part=hdofhousehold,spouse,son,daughter
part name=hdofhousehold,spouse,son,daughter;
        position=familyname,firstname
element name=flatnumber;identification=yes
exit name=census
read name=census
house$123a
flat$a
hdofhousehold$Müller/Friedrich
spouse$Müller/Johanna
son$Müller/Fritz
son$Müller/Josef
daughter$Müller/Maria
daughter$Müller/Anna
flat$b
hdofhousehold$Meier/Andreas
spouse$Meier/Katherina
son$Meier/Wilhelm
son$Meier/Hans
daughter$Meier/Ulrike
```

In this database, the following `bridge` directive

```
bridge first="123a"
```

obviously refers to the house as a whole. Every resulting element becomes a constituent of the group `house`.

Because all groups invoked in the rest of the sequence belong to this house, all their symbolic addresses also begin with the character string we have just quoted. If you want to distinguish between the flats in the house, you must append the identifier of the group "flat", which depends directly on the document, to the document identifier after an slash.

```
bridge first="123a/flat"
```

If you do not add any additional specifications, this always refers to the group's first appearance in the document. In our example then, it refers to flat `a`, the flat owned by the Müller family.

There are two ways we can refer to flat `b`, owned by the Meier family.

On the one hand, we can specify the ordinal number of the requisite flat, after a semi-colon:

```
bridge first="123a/flat;2"
```

On the other hand, we can specify the group identifier of the requisite flat after an equals sign. Because the element `flatnumber` was defined as the group identifier for this group, we can write the following:

```
bridge first="123a/flat=b"
```

At this point, note that the two following statements agree with what we have just been saying:

```
note The flat owned by the Müller family can also be described
note in detail by means of the expression
bridge first="123a/flat=a"
note or
bridge first="123a/flat;1"
```

You can continue to apply the principle we have just described to as many levels in the hierarchy as you like. Please satisfy yourself that the following symbolic addresses match the principles described above:

```
note The symbolic address for Friedrich Müller goes as follows:
bridge first="123a/flat/hdofhousehold"
note The symbolic address for Hans Meier is as follows:
bridge first="123a/flat=b/son;2"
note The symbolic address for Ulrike Meier goes as follows:
bridge first="123a/flat;2/daughter"
note The symbolic address for Josef Müller is as follows:
bridge first="123a/flat;1/son=son-2"
```

10.1.1.2.1.3 Weighted `bridge` directives

The two `bridge` directive parameters we have not yet discussed can be used to assign a specific visibility to a linkage you have defined, or assign a particular view to it.

- The `visibility=` parameter expects a value consisting of
  - a question mark, which is assigned a value of 0.5, or
  - an exclamation mark, which is assigned a value of 0.95, accompanied by
  - a number between zero and one, which is accepted as it is, or
  - a number greater than one, which is repeatedly divided by one hundred until the result lies between zero and one.

This parameter specifies that the visibility of the linkage between the two databases being processed corresponds to the given numerical value.

- The `only=` parameter expects a value consisting of a whole number between one and the largest alternative level code valid on the platform in question. (Typical values of this kind include 8, 16, 32 etc.)

It specifies that the linkage between the two databases being processed only exists in the view with the given level code.

10.1.1.2.2 Once you have set up linkages ...

Linkages set up by a `bridge` command with the `type=definitions` exactly correspond to those set up by a `bridge` command with the `part=` parameter. All additional parameters and exceptions explained there also apply in this instance.

10.1.1.3 Confirming proposed linkages

The parameter value `conformations` in the `bridge` command's `type=` parameter will give you the option of linking groups from different databases – groups which the computer has suggested are potentially identical – together. This feature has not yet been released.

10.2 `relation` definitions referring to other databases

In particular cases (above all when linking a large number of databases together), it may be desirable for purposes of organisation to define which databases a given `relation` definition can link to the database for which this definition is being made – before you actually link the databases together.

For this purpose, the user can use the `bridge` directive in `relation` definitions.

It accepts the `name=` parameter as its only parameter. The latter expects the name of a database as a value, to which the database to which this `relation` definition belongs is supposed to be linked in the course of time. The database referred to does not have to exist at this moment in time.

You can use this directive as often as you like in a `relation` definition. However each directive can only be used to prepare a linkage with a single database.

So in order to prepare the structural organisation whereby three sources are linked together you can use, for example, the following `relation` definition:

```
item name=linkages;usage=relation;source=example;type=permanent
bridge name=tax_list
bridge name=addresses
exit name=linkages
```

10.3 Managing linked databases

If you have linked several databases together, it is generally advisable to ensure that they are simultaneously available on magnetic disk. $\kappa\lambda\epsilon\iota\omega$ sometimes addresses a database which is linked to other databases even when the user has not explicitly requested it to. This happens:

1) If when processing one of the two elementary functions `each[]` or `total[]` in the context of a `write` command, the system encounters an entry which refers to another database. If the relevant database is not available, an appropriate message is displayed and the system continues to process the original database.
2) If you want to delete a database which is linked to other databases either explicitly – by issuing a `delete` command or implicitly by including `overwrite=yes` in a `database` command - $\kappa\lambda\epsilon\iota\omega$ attempts to remove all references to this databases from the databases which were linked to it. If the system is unable to address one of these databases at this point in time it does not delete the original database, but all references to it in all databases *accessible* to the system are removed. The missing database should then be made accessible and the system should then attempt to delete the one from which it started. *If in this situation you delete the "awkward" database using the operating system's own resources, and then recreate it at a later date, you may find that serious inconsistencies in the processing of linked databases result.*

Appendix A: Concordance of the command languages

A.1. Introduction

The following concordance of the command language is addressed to users who are familiar with one of the two command languages and want to use a system where the other is implemented. Such a situation may, for example, arise when:

- An experienced user wants to move to the English command language.
- A German speaking user wants to use the Image Analysis System, which so far is only documented in English, with the system he or she has currently access to.
- An English speaking user wants to apply some of the methods discussed in older volumes of the *Halbgraue Reihe*, where the Latin command language is used throughout.

Between the two command languages, there is not always a one to one relationship. In some cases, a Latin word required a translation in one context, which was semantically pointless in another one. This situation is taken care of in two ways:

- If one of the two translations has to be used (primarily with command words and directives) the appropriate context for the application of the two translations is given.
- If any translation will be accepted — as, e.g., in the case of the parameter names — the different translations are simply listed. The Latin parameter name `certe=` is translated in some contexts as `guaranteed=`, in others as `precise=`. While mnemonically only one of these two translations makes sense, $\kappa\lambda\epsilon\iota\omega$ will accept `precise=` wherever `guaranteed=` is legal.

In all sections the English commands are given in the `typewriter font` used in this series for reproduced elements of the command language; the Latin equivalents are shown in *Italics*.

A careful reader will notice that about five percent of the command vocabulary covered by this concordance is not explained in the currently published documentation. The following sections cover the command language used at the moment by *all* versions of the system, including those which make available some features only at specific installations. This course was chosen, to make this appendix useful for the remaining time during which both command languages are supported. It is *not* a good idea, though, to enter the commands quoted below, if they are not documented. In most cases a message that the feature is unreleased will be the result.

A.2. English – Latin

A.2.1 Commands and Directives

`bridge`	⇒	*pons*
`case`	⇒	*casus*
`catalogue`	⇒	*repertorium*
`category`	⇒	*condicio*
`concordance`	⇒	*concordantia*
`confirm`	⇒	*confirmatio*
`connection`	⇒	*connectio*
`continue`	⇒	*continuatio*
`conversion`	⇒	*conversio*
`create`	⇒	*creatio*
`cumulate`	⇒	*cumulatio*
`cycle`	⇒	*cyclus*
`database`	⇒	*nomino*
`date`	⇒	*tempora*
`declension`	⇒	*declinatio*
`delete`	⇒	*delende*
`describe`	⇒	*descriptio*
`device`	⇒	*device*
`double`	⇒	*duplex*
`element`	⇒	*terminus*
`evaluation`	⇒	*evaluatio*
`exit`	⇒	*exitus*
`form`	⇒	*forma*
`image`	⇒	*imago*
`index`	⇒	*index*
`item`	⇒	*item*
`keyword`	⇒	*index*
`limit`	⇒	*limes*
`location`	⇒	*situs*
`mapping`	⇒	*tabula*
`name`	⇒	*nomen*
`negate`	⇒	*negatio*
`note`	⇒	*nota*
`number`	⇒	*numerus*
`options`	⇒	*optiones*
`part`	⇒	*pars*
`path`	⇒	*iter*
`query`	⇒	*quaero*
`read`	⇒	*lege*
`redundancy`	⇒	*redundantia*
`relation`	⇒	*relatio*
`series`	⇒	*seria*
`sign`	⇒	*signum*
`signs`	⇒	*signa*
`simple`	⇒	*simplex*
`source`	⇒	*fons*
`stop`	⇒	*finis*
`substitution`	⇒	*substitutio*
`suffix`	⇒	*suffix*
`text`	⇒	*lingua*
`training`	⇒	*exercitium*
`translate`	⇒	*translatio*
`triplet`	⇒	*triplex*
`type`	⇒	*modus*
`usage`	⇒	*usus*
`visibility`	⇒	*visibilitas*
`volume`	⇒	*volumen*
`write`	⇒	*scribe*

A.2.2 Parameter Names

`after=`	⇒	*post=*
`alias=`	⇒	*alias=*
`also=`	⇒	*ceteri=*
`always=`	⇒	*semper=*
`arbitrary=`	⇒	*repetitio=*
`before=`	⇒	*prae=*
`bridge=`	⇒	*pons=*
`catalogue=`	⇒	*repertorium=*
`category=`	⇒	*condicio=*
`classification=`	⇒	*classificatio=*
`codebook=`	⇒	*thesaurus=*
`colour=`	⇒	*color=*
`connection=`	⇒	*connectio=*
`continuation=`	⇒	*continuatio=*
`conversion=`	⇒	*conversio=*
`cumulate=`	⇒	*cumulatio=*
`current=`	⇒	*antiquum=*
`date=`	⇒	*tempora=*
`declension=`	⇒	*declinatio=*
`device=`	⇒	*device=*
`easter=`	⇒	*pascha=*
`environment=`	⇒	*kontext=*
`explanation=`	⇒	*explanatio=*
`first=`	⇒	*primum=*

`format=`	⇒	*forma=*
`gender=`	⇒	*genus=*
`german=`	⇒	*germanus=*
`guarantueed=`	⇒	*certe=*
`guth=`	⇒	*guth=*
`identification=`	⇒	*identificatio=*
`image=`	⇒	*imago=*
`lemma=`	⇒	*lemma=*
`limit=`	⇒	*limes=*
`lines=`	⇒	*linea=*
`location=`	⇒	*situs=*
`maximum=`	⇒	*maximum=*
`minimum=`	⇒	*minimum=*
`more=`	⇒	*plures=*
`name=`	⇒	*nomen=*
`null=`	⇒	*nullus=*
`number=`	⇒	*numerus=*
`only=`	⇒	*solum=*
`order=`	⇒	*ordo=*
`other=`	⇒	*alter=*
`overwrite=`	⇒	*antiquum=*
`part=`	⇒	*pars=*
`path=`	⇒	*iter=*
`position=`	⇒	*locus=*
`precise=`	⇒	*certe=*
`prefix=`	⇒	*prae=*
`preparation=`	⇒	*prae=*
`read=`	⇒	*lege=*
`relation=`	⇒	*relatio=*
`repeat=`	⇒	*repetitio=*
`result=`	⇒	*novum=*
`same=`	⇒	*idem=*
`second=`	⇒	*secundum=*
`self=`	⇒	*ipse=*
`sequence=`	⇒	*sequentia=*
`signs=`	⇒	*signa=*
`skeleton=`	⇒	*skeleton=*
`soundex=`	⇒	*soundex=*
`source=`	⇒	*fons=*
`start=`	⇒	*principium=*
`substitution=`	⇒	*substitutio=*
`suffix=`	⇒	*post=*
`target=`	⇒	*destinatio=*
`text=`	⇒	*lingua=*
`total=`	⇒	*totum=*
`type=`	⇒	*modus=*
`usage=`	⇒	*usus=*
`visibility=`	⇒	*visibilitas=*
`volume=`	⇒	*volumen=*
`warning=`	⇒	*mone=*
`weekday=`	⇒	*feria=*
`without=`	⇒	*sine=*
`write=`	⇒	*scribe=*

A.2.3 Builtin Functions

`after[]`	⇒	*post[]*
`also[]`	⇒	*ceteri[]*
`back[]`	⇒	*avus[]*
`before[]`	⇒	*ante[]*
`brother[]`	⇒	*frater[]*
`catalogue[]`	⇒	*repertorium[]*
`category[]`	⇒	*condicio[]*
`codebook[]`	⇒	*thesaurus[]*
`collect[]`	⇒	*collectio[]*
`comment[]`	⇒	*commentarium[]*
`continue[]`	⇒	*continuatio[]*
`conversion[]`	⇒	*conversio[]*
`core[]`	⇒	*nucleus[]*
`cousin[]`	⇒	*consobrinus[]*
`cumulation[]`	⇒	*cumulatio[]*
`date[]`	⇒	*tempora[]*
`days[]`	⇒	*dies[]*
`difference[]`	⇒	*differentia[]*
`each[]`	⇒	*totum[]*
`expectation[]`	⇒	*expectatio[]*
`father[]`	⇒	*pater[]*
`first[]`	⇒	*primum[]*
`form[]`	⇒	*forma[]*
`forward[]`	⇒	*nepos[]*
`guth[]`	⇒	*guth[]*
`image[]`	⇒	*imago[]*
`keyword[]`	⇒	*index[]*
`last[]`	⇒	*ultimum[]*
`lemma[]`	⇒	*lemma[]*
`lines[]`	⇒	*linea[]*
`location[]`	⇒	*situs[]*
`month[]`	⇒	*mensis[]*

`name[]` ⇒ *nomen[]*
`none[]` ⇒ *nullus[]*
`number[]` ⇒ *numerus[]*
`offspring[]` ⇒ *filius[]*
`order[]` ⇒ *ordo[]*
`original[]` ⇒ *verum[]*
`page[]` ⇒ *pagina[]*
`part[]` ⇒ *pars[]*
`path[]` ⇒ *iter[]*
`position[]` ⇒ *locus[]*
`query[]` ⇒ *ibidem[]*
`relation[]` ⇒ *relatio[]*
`root[]` ⇒ *radix[]*
`same[]` ⇒ *idem[]*
`selection[]` ⇒ *selectio[]*
`sign[]` ⇒ *signum[]*
`skeleton[]` ⇒ *skeleton[]*
`soundex[]` ⇒ *soundex[]*
`status[]` ⇒ *status[]*
`target[]` ⇒ *destinatio[]*
`text[]` ⇒ *lingua[]*
`total[]` ⇒ *omnes[]*
`usage[]` ⇒ *usus[]*
`visibility[]` ⇒ *visibilitas[]*
`weekday[]` ⇒ *feria[]*
`word[]` ⇒ *verbum[]*
`year[]` ⇒ *annus[]*

A.2.3 Classes of Logical Objects

`bridge` ⇒ *pons*
`catalogue` ⇒ *repertorium*
`category` ⇒ *condicio*
`chronology` ⇒ *chronologia*
`classification` ⇒ *classificatio*
`codebook` ⇒ *thesaurus*
`connection` ⇒ *connectio*
`conversion` ⇒ *conversio*
`date` ⇒ *tempora*
`guth` ⇒ *guth*
`image` ⇒ *imago*
`keyword` ⇒ *index*
`location` ⇒ *situs*
`name` ⇒ *nomen*
`number` ⇒ *numerus*
`order` ⇒ *ordo*
`path` ⇒ *iter*
`relation` ⇒ *relatio*
`skeleton` ⇒ *skeleton*
`soundex` ⇒ *soundex*
`source` ⇒ *fons*
`substitution` ⇒ *substitutio*
`text` ⇒ *lingua*
`usage` ⇒ *usus*

A.2.4 Keywords in Conditions

`after` ⇒ *post*
`and` ⇒ *et*
`before` ⇒ *ante*
`circa` ⇒ *circa*
`equal` ⇒ *aequus*
`greater` ⇒ *maior*
`less` ⇒ *minor*
`limit` ⇒ *limes*
`not` ⇒ *non*
`null` ⇒ *nullus*
`or` ⇒ *vel*
`others` ⇒ *alii*
`start` ⇒ *principium*

A.2.5 Miscellaneous Keywords

`a declension` ⇒ *a-deklination*
`absolute` ⇒ *absoluta*
`after` ⇒ *post*
`aix-screen` ⇒ *aix-screen*
`algorithm` ⇒ *algorithmus*
`always` ⇒ *semper*
`annunciation` ⇒ *annunciatio*
`arcinfo` ⇒ *arcinfo*
`back` ⇒ *avus*
`batch` ⇒ *munus*
`before` ⇒ *ante*
`black-white` ⇒ *niger*
`blue` ⇒ *caeruleus*
`byzantine` ⇒ *byzantium, byzanz*
`category` ⇒ *condicio*
`censsys` ⇒ *censsys*

`character` ⇒ *signa*
`circumcision` ⇒ *circumcisio*
`clio` ⇒ *clio*
`colour` ⇒ *color*
`colourpostscript` ⇒ *colourpostscript*
`comment` ⇒ *commentarium*
`complete` ⇒ *totum*
`confirmations` ⇒ *corroborationes*
`continue` ⇒ *continuatio*
`contrast` ⇒ *niger*
`count` ⇒ *numerus*
`creation` ⇒ *creatio*
`date` ⇒ *tempora*
`day` ⇒ *dies*
`days` ⇒ *dies*
`definitions` ⇒ *definitiones*
`destruction` ⇒ *destructio*
`detailed` ⇒ *differentialiter*
`deviation` ⇒ *variatio*
`devices` ⇒ *devices*
`dialogue` ⇒ *conversatio*
`difference` ⇒ *differentia*
`digipad` ⇒ *digipad*
`disspla` ⇒ *disspla*
`dos-screen` ⇒ *dos-screen*
`double` ⇒ *duplex*
`dynamic` ⇒ *variatio*
`easter` ⇒ *pascha*
`element` ⇒ *terminus* except: ⇒ *nomen* as value of the `type=` / *modus=* parameter of the `name` / *nomen* directive of a `substitutio` / *substitution* definition.
`elements` ⇒ *terminus*
`empty` ⇒ *vacuum*
`equality` ⇒ *aequalitas*
`exclusive` ⇒ *excludens*
`female` ⇒ *frau*
`first` ⇒ *primum* except: ⇒ *erstwort* within the definition of germanic name lemmatizations.
`foreign` ⇒ *fremd*
`friday` ⇒ *veneris*
`full` ⇒ *totum*
`generation` ⇒ *generatio*
`generic` ⇒ *genera, generanda*
`germanic` ⇒ *germanisch*
`green` ⇒ *viridis*
`group` ⇒ *pars*
`halftone` ⇒ *dimidius*
`human` ⇒ *homo*
`i declension` ⇒ *i-deklination*
`image` ⇒ *imago*
`images` ⇒ *imago*
`inclusive` ⇒ *includens*
`index` ⇒ *index*
`insertion` ⇒ *insertio*
`islam` ⇒ *islam*
`large` ⇒ *magnum*
`last` ⇒ *ultimum*
`latin` ⇒ *latina, latinum*
`left` ⇒ *sinister*
`letters` ⇒ *litterae*
`limit` ⇒ *limes*
`location` ⇒ *situs*
`long` ⇒ *longae*
`machine` ⇒ *machina*
`male` ⇒ *mann*
`map` ⇒ *tabula*
`mean` ⇒ *centrum*
`monday` ⇒ *lune*
`month` ⇒ *mensis*
`moses` ⇒ *moses*
`multiple` ⇒ *multiplex*
`multiple entries` ⇒ *pars*
`multiple entry` ⇒ *pars*
`names` ⇒ *nomen, nomina*
`nativity` ⇒ *nativitas*
`negative` ⇒ *albus*
`no` ⇒ *non*
`none` ⇒ *nulla, nullum*
`null` ⇒ *nullus*
`number` ⇒ *numerus*
`numbers` ⇒ *numerus* except: ⇒ *deutsch* as value of the `name=` / *nomen=* parameter of the `type` / *modus* directive of a `date` / *tempora* definition.
`o declension` ⇒ *o-deklination*

`octave` ⇒ *octava*

`original` ⇒ *verum*

`others` ⇒ *alii*

`partial` ⇒ *partialis*

`parts` ⇒ *pars, partes*

`pcspss` ⇒ *pcspss*

`permanent` ⇒ *permanens* except: ⇒ *semper* as argument of `query[]` / *ibidem[].*

`phonetic` ⇒ *pars*

`postscript` ⇒ *postscript*

`previous` ⇒ *pridie*

`production` ⇒ *laborandi*

`red` ⇒ *rubrum*

`relation` ⇒ *relatio*

`revolution` ⇒ *revolution*

`right` ⇒ *dexter*

`roman` ⇒ *romana*

`root` ⇒ *radix*

`s declension` ⇒ *s-deklination*

`saints` ⇒ *sancti*

`sas` ⇒ *sas*

`saturday` ⇒ *saturni*

`second` ⇒ *zweitwort*

`self` ⇒ *ipse*

`sentence` ⇒ *sententiae*

`sentences` ⇒ *sententiae*

`short` ⇒ *brevis* except: ⇒ *kurzwort* in the definition of germanic name lemmatizations.

`shsecond` ⇒ *kuzweit*

`simple` ⇒ *simplex*

`small` ⇒ *parvum*

`solid` ⇒ *plenus*

`spss` ⇒ *spss*

`standard` ⇒ *standard*

`start` ⇒ *principium*

`stop` ⇒ *finis*

`structure` ⇒ *structura*

`substitution` ⇒ *substitutio*

`sunday` ⇒ *solis*

`survey` ⇒ *generaliter*

`table` ⇒ *tabula*

`temporary` ⇒ *ad hoc* except: ⇒ *nunc* as argument of `query[]` / *ibidem[].*

`tenthtone` ⇒ *tenuis*

`terms` ⇒ *terminus* except: ⇒ *nomina* as value of the `write=` / *scribe=* parameter of the `database` / *nomino* or the `read` / *lege* command.

`text` ⇒ *lingua*

`thursday` ⇒ *jovis*

`total` ⇒ *summa*

`training` ⇒ *exercandi*

`triple` ⇒ *triplex*

`tuesday` ⇒ *martis*

`u declension` ⇒ *u-deklination*

`unknown` ⇒ *unbekannt*

`variation` ⇒ *variatio*

`volumes` ⇒ *volumen*

`wednesday` ⇒ *mercurii*

`weekday` ⇒ *feria*

`western` ⇒ *english*

`without` ⇒ *sine*

`words` ⇒ *verba*

`year` ⇒ *annus*

`yes` ⇒ *sic*

A.2.6 Calendar Keywords

The great majority of keywords is – at least in their significant parts – identical in both command languages. Only keywords which deviate are listed.

`av` ⇒ *ab*

`cheshvan` ⇒ *cheschwan*

`december` ⇒ *dezember*

`dhul-hijja` ⇒ *dsulhidscha*

`dhul-qa'da` ⇒ *dsulkada*

`iyyar` ⇒ *ijar*

`jumada1` ⇒ *dschumada1*

`jumada2` ⇒ *dschumada2*

`march` ⇒ *maerz*

`may` ⇒ *mai*

`nisan` ⇒ *nissan*

`october` ⇒ *oktober*

`os` ⇒ *as*

`rajab` ⇒ *radschab*

`sha'ban` ⇒ *schaban*

shawwal ⇒ *schawwal*
shevat ⇒ *schebat*
sivan ⇒ *siwan*
tevet ⇒ *tebeth*
tishri ⇒ *tischri*
unknown ⇒ *unbekannt*
ve-adar ⇒ *weadar*

A.3. Latin – English

A.3.1 Command Words and Directives

casus ⇒ case
concordantia ⇒ concordance
condicio ⇒ category
confirmatio ⇒ confirm
connectio ⇒ connection
continuatio ⇒ continue
conversio ⇒ conversion
creatio ⇒ create
cumulatio ⇒ cumulate
cyclus ⇒ cycle
declinatio ⇒ declension
delende ⇒ delete
descriptio ⇒ describe
device ⇒ device
duplex ⇒ double
evaluatio ⇒ evaluation
exercitium ⇒ training
exitus ⇒ exit
finis ⇒ stop
fons ⇒ source
forma ⇒ form
imago ⇒ image
index ⇒ index except: ⇒ keyword as directive in a catalogue / *repertorium* definition.
item ⇒ item
iter ⇒ path
lege ⇒ read
limes ⇒ limit
lingua ⇒ text
modus ⇒ type
negatio ⇒ negate
nomen ⇒ name
nomino ⇒ database
nota ⇒ note
numerus ⇒ number
optiones ⇒ options
pars ⇒ part
pons ⇒ bridge
quaero ⇒ query

redundantia ⇒ `redundancy`
relatio ⇒ `relation`
repertorium ⇒ `catalogue`
scribe ⇒ `write`
seria ⇒ `series`
signa ⇒ `signs`
signum ⇒ `sign`
simplex ⇒ `simple`
situs ⇒ `location`
substitutio ⇒ `substitution`
suffix ⇒ `suffix`
tabula ⇒ `mapping`
tempora ⇒ `date`
terminus ⇒ `element`
translatio ⇒ `translate`
triplex ⇒ `triplet`
usus ⇒ `usage`
visibilitas ⇒ `visibility`
volumen ⇒ `volume`

A.3.2 Parameter Names

alias= ⇒ `alias=`
alter= ⇒ `other=`
antiquum= ⇒ `current=`, `overwrite=`
certe= ⇒ `guarantueed=`, `precise=`
ceteri= ⇒ `also=`
classificatio= ⇒ `classification=`
color= ⇒ `colour=`
condicio= ⇒ `category=`
connectio= ⇒ `connection=`
continuatio= ⇒ `continuation=`
conversio= ⇒ `conversion=`
cumulatio= ⇒ `cumulate=`
declinatio= ⇒ `declension=`
destinatio= ⇒ `target=`
device= ⇒ `device=`
explanatio= ⇒ `explanation=`
feria= ⇒ `weekday=`
fons= ⇒ `source=`
forma= ⇒ `format=`
genus= ⇒ `gender=`
germanus= ⇒ `german=`
guth= ⇒ `guth=`
idem= ⇒ `same=`
identificatio= ⇒ `identification=`
imago= ⇒ `image=`
ipse= ⇒ `self=`
iter= ⇒ `path=`
kontext= ⇒ `environment=`
lege= ⇒ `read=`
lemma= ⇒ `lemma=`
limes= ⇒ `limit=`
linea= ⇒ `lines=`
lingua= ⇒ `text=`
locus= ⇒ `position=`
maximum= ⇒ `maximum=`
minimum= ⇒ `minimum=`
modus= ⇒ `type=`
mone= ⇒ `warning=`
nomen= ⇒ `name=`
novum= ⇒ `result=`
nullus= ⇒ `null=`
numerus= ⇒ `number=`
ordo= ⇒ `order=`
pars= ⇒ `part=`
pascha= ⇒ `easter=`
plures= ⇒ `more=`
pons= ⇒ `bridge=`
post= ⇒ `after=`, `suffix=`
prae= ⇒ `before=`, `prefix=`, `preparation=`
primum= ⇒ `first=`
principium= ⇒ `start=`
relatio= ⇒ `relation=`
repertorium= ⇒ `catalogue=`
repetitio= ⇒ `arbitrary=`, `repeat=`
scribe= ⇒ `write=`
secundum= ⇒ `second=`
semper= ⇒ `always=`
sequentia= ⇒ `sequence=`
signa= ⇒ `signs=`
sine= ⇒ `without=`
situs= ⇒ `location=`
skeleton= ⇒ `skeleton=`
solum= ⇒ `only=`
soundex= ⇒ `soundex=`
substitutio= ⇒ `substitution=`

tempora= ⇒ `date=`
thesaurus= ⇒ `codebook=`
totum= ⇒ `total=`
usus= ⇒ `usage=`
visibilitas= ⇒ `visibility=`
volumen= ⇒ `volume=`

A.3.3 Builtin Functions

annus[] ⇒ `year[]`
ante[] ⇒ `before[]`
avus[] ⇒ `back[]`
ceteri[] ⇒ `also[]`
collectio[] ⇒ `collect[]`
commentarium[] ⇒ `comment[]`
condicio[] ⇒ `category[]`
consobrinus[] ⇒ `cousin[]`
continuatio[] ⇒ `continue[]`
conversio[] ⇒ `conversion[]`
cumulatio[] ⇒ `cumulation[]`
destinatio[] ⇒ `target[]`
dies[] ⇒ `days[]`
differentia[] ⇒ `difference[]`
expectatio[] ⇒ `expectation[]`
feria[] ⇒ `weekday[]`
filius[] ⇒ `offspring[]`
forma[] ⇒ `form[]`
frater[] ⇒ `brother[]`
guth[] ⇒ `guth[]`
ibidem[] ⇒ `query[]`
idem[] ⇒ `same[]`
imago[] ⇒ `image[]`
index[] ⇒ `keyword[]`
iter[] ⇒ `path[]`
lemma[] ⇒ `lemma[]`
linea[] ⇒ `lines[]`
lingua[] ⇒ `text[]`
locus[] ⇒ `position[]`
mensis[] ⇒ `month[]`
nepos[] ⇒ `forward[]`
nomen[] ⇒ `name[]`
nucleus[] ⇒ `core[]`
nullus[] ⇒ `none[]`
numerus[] ⇒ `number[]`
omnes[] ⇒ `total[]`
ordo[] ⇒ `order[]`
pagina[] ⇒ `page[]`
pars[] ⇒ `part[]`
pater[] ⇒ `father[]`
post[] ⇒ `after[]`
primum[] ⇒ `first[]`
radix[] ⇒ `root[]`
relatio[] ⇒ `relation[]`
repertorium[] ⇒ `catalogue[]`
selectio[] ⇒ `selection[]`
signum[] ⇒ `sign[]`
situs[] ⇒ `location[]`
skeleton[] ⇒ `skeleton[]`
soundex[] ⇒ `soundex[]`
status[] ⇒ `status[]`
tempora[] ⇒ `date[]`
thesaurus[] ⇒ `codebook[]`
totum[] ⇒ `each[]`
ultimum[] ⇒ `last[]`
usus[] ⇒ `usage[]`
verbum[] ⇒ `word[]`
verum[] ⇒ `original[]`
visibilitas[] ⇒ `visibility[]`

A.3.3 Classes of Logical Objects

chronologia ⇒ `chronology`
classificatio ⇒ `classification`
condicio ⇒ `category`
connectio ⇒ `connection`
conversio ⇒ `conversion`
fons ⇒ `Source`
guth ⇒ `Guth`
imago ⇒ `image`
index ⇒ `keyword`
iter ⇒ `path`
lingua ⇒ `Text`
nomen ⇒ `Name`
numerus ⇒ `Number`
ordo ⇒ `Order`
pons ⇒ `bridge`
relatio ⇒ `relation`
repertorium ⇒ `catalogue`

situs ⇒ `location`
skeleton ⇒ `skeleton`
soundex ⇒ `soundex`
substitutio ⇒ `substitution`
tempora ⇒ `date`
thesaurus ⇒ `codebook`
usus ⇒ `usage`

A.3.4 Keywords in Conditions

aequus ⇒ `equal`
alii ⇒ `others`
ante ⇒ `before`
circa ⇒ `circa`
et ⇒ `and`
limes ⇒ `limit`
maior ⇒ `greater`
minor ⇒ `less`
non ⇒ `not`
nullus ⇒ `null`
post ⇒ `after`
principium ⇒ `start`
vel ⇒ `or`

A.3.5 Miscellaneous Keywords

a-deklination ⇒ `a declension`
absoluta ⇒ `absolute`
ad hoc ⇒ `temporary`
aequalitas ⇒ `equality`
aix-screen ⇒ `aix-screen`
albus ⇒ `negative`
algorithmus ⇒ `algorithm`
alii ⇒ `others`
annunciatio ⇒ `annunciation`
annus ⇒ `year`
ante ⇒ `before`
arcinfo ⇒ `arcinfo`
avus ⇒ `back`
brevis ⇒ `short`
byzantium ⇒ `byzantine`
byzanz ⇒ `byzantine`
caeruleus ⇒ `blue`
censsys ⇒ `censsys`
centrum ⇒ `mean`
circumcisio ⇒ `circumcision`
clio ⇒ `clio`
color ⇒ `colour`
colourpostscript ⇒ `colourpostscript`
commentarium ⇒ `comment`
condicio ⇒ `category`
continuatio ⇒ `continue`
conversatio ⇒ `dialogue`
corroborationes ⇒ `confirmations`
creatio ⇒ `creation`
definitiones ⇒ `definitions`
destructio ⇒ `destruction`
deutsch ⇒ `numbers`
devices ⇒ `devices`
dexter ⇒ `right`
dies ⇒ `day, days`
differentia ⇒ `difference`
differentialiter ⇒ `detailed`
digipad ⇒ `digipad`
dimidius ⇒ `halftone`
disspla ⇒ `disspla`
dos-screen ⇒ `dos-screen`
duplex ⇒ `double`
english ⇒ `western`
erstwort ⇒ `first`
excludens ⇒ `exclusive`
exercandi ⇒ `training`
feria ⇒ `weekday`
finis ⇒ `stop`
frau ⇒ `female`
fremd ⇒ `foreign`
genera ⇒ `generic`
generaliter ⇒ `survey`
generanda ⇒ `generic`
generatio ⇒ `generation`
germanisch ⇒ `germanic`
homo ⇒ `human`
i-deklination ⇒ `i declension`
imago ⇒ `image, images`
includens ⇒ `inclusive`
index ⇒ `index`
insertio ⇒ `insertion`
ipse ⇒ `self`
islam ⇒ `islam`

jovis ⇒ `thursday`
kurzwort ⇒ `short`
kuzweit ⇒ `shsecond`
laborandi ⇒ `production`
latina ⇒ `latin`
latinum ⇒ `latin`
limes ⇒ `limit`
lingua ⇒ `text`
litterae ⇒ `letters`
longae ⇒ `long`
lune ⇒ `monday`
machina ⇒ `machine`
magnum ⇒ `large`
mann ⇒ `male`
martis ⇒ `tuesday`
mensis ⇒ `month`
mercurii ⇒ `wednesday`
moses ⇒ `moses`
multiplex ⇒ `multiple`
munus ⇒ `batch`
nativitas ⇒ `nativity`
niger ⇒ `contrast` except: *niger* ⇒ `black-white` as image qualifier in input data.
nomen ⇒ `names` except: ⇒ `element` as value of the `type=` / *modus=* parameter of the `name` / *nomen* directive of a `substitution` / *substitutio* definition.
nomina ⇒ `terms` except: ⇒ `names` as argument of `lemma[]` / *lemma[]*.
non ⇒ `no`
nulla ⇒ `none`
nullum ⇒ `none`
nullus ⇒ `null`
number ⇒ `number`
numerus ⇒ `number, numbers` except: ⇒ `count` as value of the `type=` / *modus=* parameter of the `index` / *index* command.
nunc ⇒ `temporary`
o-deklination ⇒ `o declension`
octava ⇒ `octave`
pars ⇒ `parts` except: `group` as value of the `type=` / *modus=* parameter of the `name` / *nomen* directive of a `substitution` / *substitutio* definition; ⇒ `multiple entry, multiple entries` as value of the `position=` / *locus=* parameter of the `catalogue` / *repertorium* command; ⇒ `phonetic` as value of the `more=` / *plures=* parameter of the `part` / *pars* directive of a `soundex` / *soundex* definition.
partes ⇒ `parts`
partialis ⇒ `partial`
parvum ⇒ `small`
pascha ⇒ `easter`
pcspss ⇒ `pcspss`
permanens ⇒ `permanent`
plenus ⇒ `solid`
post ⇒ `after`
postscript ⇒ `postscript`
pridie ⇒ `previous`
primum ⇒ `first`
principium ⇒ `start`
radix ⇒ `root`
relatio ⇒ `relation`
revolution ⇒ `revolution`
romana ⇒ `roman`
rubrum ⇒ `red`
s-deklination ⇒ `s declension`
sancti ⇒ `saints`
sas ⇒ `sas`
saturni ⇒ `saturday`
semper ⇒ `always` except: ⇒ `permanent` as argument of `query[]` / *ibidem[]*.
sententiae ⇒ `sentence, sentences`
sic ⇒ `yes`
signa ⇒ `character`
simplex ⇒ `simple`
sine ⇒ `without`
sinister ⇒ `left`
situs ⇒ `location`
solis ⇒ `sunday`
spss ⇒ `spss`
standard ⇒ `standard`
structura ⇒ `structure`
substitutio ⇒ `substitution`

summa ⇒ `total`
tabula ⇒ `map` except: ⇒ `table` as value of the `type=` / *modus=* parameter of the `index` / *index* command.
tempora ⇒ `date`
tenuis ⇒ `tenthtone`
terminus ⇒ `element, elements` except: ⇒ `terms` as value of the `type=` / *modus=* parameter of the `catalogue` / *repertorium* command.
totum ⇒ `full` except: ⇒ `complete` as argument of `catalogue` / *repertorium[]*.
triplex ⇒ `triple`
u-deklination ⇒ `u declension`
ultimum ⇒ `last`
unbekannt ⇒ `unknown`
vacuum ⇒ `empty`
variatio ⇒ `deviation` except: ⇒ `dynamic` as image qualifier in input data; ⇒ `variation` as argument of `cumulate[]` / *cumulatio[]*.
veneris ⇒ `friday`
verba ⇒ `words`
verum ⇒ `original`
viridis ⇒ `green`
volumen ⇒ `volumes`
zweitwort ⇒ `second`

A.3.6 Calendar Keywords

The great majority of keywords is – at least in their significant parts – identical in both command languages. Only keywords which deviate are listed.

ab ⇒ `av`
as ⇒ `os`
cheschwan ⇒ `cheshvan`
dezember ⇒ `december`
dschumada1 ⇒ `jumada1`
dschumada2 ⇒ `jumada2`
dsulhidscha ⇒ `dhul-hijja`
dsulkada ⇒ `dhul-qa'da`
ijar ⇒ `iyyar`
maerz ⇒ `march`
mai ⇒ `may`
nissan ⇒ `nisan`
oktober ⇒ `october`
radschab ⇒ `rajab`
schaban ⇒ `sha'ban`
schawwal ⇒ `shawwal`
schebat ⇒ `shevat`
siwan ⇒ `sivan`
tebeth ⇒ `tevet`
tischri ⇒ `tishri`
unbekannt ⇒ `unknown`
weadar ⇒ `ve-adar`

Appendix B: The κλειω configuration program

B.1. Introduction

κλειω reads many of the assumptions the program uses from a *configuration file.* This file can be modified by a utility program distributed with the software, which is known as *κλειω configurator.* The configurator in turn expects directives defining what default assumptions κλειω should use, which are prepared as an ASCII file and compiles them into the configuration file κλειω actually uses.

This configuration can be changed by the user; in some cases a behaviour of the program will arise, which for the uniniate may look like a completely different piece of software. Indeed, one of the intentions in providing this facility was the possibility to create end user configurations, which require significantly less understanding of the underlying logic than κλειω as it is distributed.

The person using the configuration facilities, however, is expected to understand the system quite well and to have a good understanding of the concept of environment variables.

The configuration system is currently rapidly expanding: that is, many additional aspects of κλειωs behaviour will in the near future become controlled by the logic described here. This appendix does therefore only describe the basic logic of the system: the current state of the implementation can be gotten directly out of the distributed files.

B.2 Inspecting the current configuration

To get a report on the configuration κλειω uses currently on your installation, type the command `showconfig` at the command line. A report on the configuration in the command format discussed in section B.4 will be displayed. `showconfig` is copied by the installation program into the same directory which also holds the executable κλειω program.

B.3 Where does κλειω expect the configuration to be?

When κλειω is called, it looks for a file with the name `kl@@cfg`. The double at-signs shall prevent the user accidentally to create another file with that name. If that file is missing, the program cannot execute. If its contents have been changed in any other way but the one described here, the results are unpredictable.

When installed, κλειω establishes a copy of this file in the directory where the executable program is kept.

When κλειω is called, it tries to find a file with this name according to the following rules:

1) If the directory in which the program is activated contains a file `kl@@cfg` that is used.
2) Otherwise the system checks whether an environment variable `KLEIOCONFIG` has been defined and whether it names a directory, which contains `kl@@cfg`. If both conditions are met that file is used.
3) On UNIX systems only: next the HOME-directory of the user is checked. If that directory contains `kl@@cfg` that file is used.
4) Finally the system checks whether an environment variable `KLEIO` has been defined and whether it names a directory, which contains `kl@@cfg`. If both conditions are met that file is used.
5) If no file `kl@@cfg` has been found the system stops.

This mechanism has been provided to enable installations, where different users and/or different databases can use one version of the software with significantly different types of behaviour.

B.4 Changing the configuration

Before you change the configuration, please make a copy of the current version of the `kl@@cfg` *file. If it gets damaged, κλειω cannot longer be activated.*

To change a configuration file, you use a program called `config` which by the installation process has been placed in the same directory as the executable κλειω program. This program expects to be called with the name of a *configuration script*. A configuration script is an ASCII file containing a series of configuration parameters.

If you specify the name of a configuration script without an extension, `.cfg` will be added to the name. Calling

```
config local
```

directs the κλειω configurator therefore to look for a configuration script in a file named `local.cfg`.

B.4.1 Understanding configuration scripts

When the system is installed on your machine a configuration script `default.cfg` is copied into the directory which contains the executable κλειω program. This file can be inspected with any editor or wordprocessor.

To understand the syntax of the configuration language, we reprint the beginning of the default script, as distributed with versions 5.1.1/6.1.1.

```
# Section Default:
# This section controls various default sizes, characters
# and numerical quantities used by the system
################################################################################
# Subsection Default:Display:
# This subsection controls the sizes used for composing output pages.
################################################################################
# Subsection Default:Display:Printer:
# This subsection controls the sizes used to compose output pages to be sent
# into a file which is afterwards to be printed.  Such files are composed
# when by any means the output is directed to a file instead of the screen.
# ==============================================================================
# Parameter Rows=
# This parameter defines the number of rows a page on the printer is expected
# to have.
# It expects a number as value.
Default:Display:Printer:Rows=62.
# ==============================================================================
# Parameter Columns=
# This parameter defines the number of columns a page on the printer is
# expected to have.
# It expects a number as value.
Default:Display:Printer:Columns=132.
###############################################################################
# Subsection Default:Display:Screen:
# This subsection controls the sizes used to compose output pages to be sent
# to the screen.
# =============================================================================
# Parameter Rows=
# This parameter defines the number of rows the screen is expected to have.
# It expects a number as value.
Default:Display:Screen:Rows=22.
# =============================================================================
# Parameter Columns=
# This parameter defines the number of columns the screen is expected to have.
# It expects a number as value.
Default:Display:Screen:Columns=80.
###############################################################################
# Subsection Default:Charset:
# This subsection controls the signal characters used by kleio.  Please be
# aware that redifining these characters does not only affect data during
# the reading of raw data, but also the characters used to delimit various
# elements of the command language, like parameters.
# =============================================================================
# Parameter Dollar=
# This parameter defines data signal character one, defaulting to a dollar sign.
# It expects a single character as value.
Default:Charset:Dollar=$.
# =============================================================================
# Parameter Slash=
# This parameter defines data signal character two, defaulting to a slash.
# It expects a single character as value.
Default:Charset:Slash=/.
# =============================================================================
# Parameter Equal=
# This parameter defines data signal character three defaulting to an
# equal sign.
# It expects a single character as value.
Default:Charset:Equal==.
# =============================================================================
# Parameter Hash=
# This parameter defines data signal character four, defaulting to a hash sign.
# It expects a single character as value.
Default:Charset:Hash=#.
# =============================================================================
# Parameter Percent=
# This parameter defines data signal character five, defaulting to a percent
# sign.
# It expects a single character as value.
Default:Charset:Percent=%.
# ============================================================================
# Parameter Left=
# This parameter defines data signal character six, defaulting to a left
# angular bracket.
# It expects a single character as value.
Default:Charset:Left=<.
# ============================================================================
# Parameter Right=
# This parameter defines data signal character seven, defaulting to a right
# angular bracket.
# It expects a single character as value.
Default:Charset:Right=>.
# ============================================================================
# Parameter Semicolon=
# This parameter defines data signal character eight, defaulting to a semicolon.
# It expects a single character as value.
Default:Charset:Semicolon=;.
# ============================================================================
# Parameter Colon=
# This parameter defines data signal character nine, defaulting to a colon.
# It expects a single character as value.
Default:Charset:Colon=:.
# ============================================================================
# Parameter Backslash=
# This parameter defines data signal character ten, defaulting to a backslash.
# It expects a single character as value.
Default:Charset:Backslash=\.
# ============================================================================
# Parameter Question=
# This parameter defines flag character one, defaulting to a question mark.
# It expects a single character as value.
Default:Charset:Question=?.
# ============================================================================
# Parameter Exclamation=
# This parameter defines flag character two, defaulting to an exclamation mark.
# It expects a single character as value.
Default:Charset:Exclamation=!.
# ============================================================================
##############################################################################
```

From this file the following rules are probably immediately apparent:

1) A line starting with a hash sign # is a comment line.
2) A line starting with anything else is assumed to be a valid parameter definition.
3) Parameter definitions start with a keyword describing the section of the configuration to be modified, followed by a variable length chain of keywords describing subsections, being separated by colons and finally a parameter name ending with an equal sign.
4) After the equal sign a parameter value follows, which has to be specified according to the type of parameter used.
5) Independent of the type of parameter the parameter definition *must* end with a full stop.

B.4.2 Writing configuration scripts

To write your own alternate configuration script, you should copy such parameters as you require into an ASCII file, delete everything that does not need to be changed and modify the parameter values of those parameters, which you want to change. So, if you want to configure your system in such a way, that it assumes your printer to have lines of 80 characters, you would produce a configuration file containing the single line:

```
Default:Display:Printer:Columns=80.
```

B.4.3 Compiling configuration scripts

When you call the `config` program, it will look for a `kl@@cfg` file to modify according to your configuration script. It will use the same logic in searching for that file which has been described in section B.3 above.

You will normally proceed in the following way therefore:

1) Copy the `kl@@cfg` file from the kleio directory into the directory where you want the new one to reside. On a DOS system for example by something like:

   ```
   copy \kleio\bin\kl@@cfg \mydatabase
   ```

2) Now you go into the new directory and write the shell script, as described in section B.4.2.
3) And compile it by a call like

   ```
   config local
   ```

4) Now the new configuration will take effect whensoever you call κλειω from the directory `\mydatabase`. *The old configuration will still remain valid, when the system is called from anywhere else.*
5) Check now, whether the new configuration behaves as expected. If this is so,
6) you might copy the new configuration file into the default directory to become activated by all future calls to κλειω, for example by:

   ```
   copy \mydatabase\kl@@cfg \kleio\bin
   ```

When you get any error messages during compilation of your configuration script it is very strongly recommended, that you recopy the original configuration file, as in step 1 above, before you run the corrected shell script again.

Index of builtin functions

If two pages are given, the first one refers to the group function with the respective name and the second one to the element function.

Max-Planck-Institut für Geschichte, Göttingen