Vorbemerkung zur PDF-Ausgabe

Diese PDF-Datei wurde auf der Grundlage des Postscript-Dateien des Bandes 23 der *Halbgrauen Reihe zur historischen Fachinformatik* (HGR), Reihe A, erstellt.[1] Es handelt sich um die Ausgabe des $\kappa\lambda\epsilon\iota\omega$-Tutorials 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:

– eine Systembeschreibung aus dem Jahr 1993 (HGR, B 11), die 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 unter der URL: `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. Kapitel 8 (Catalogues), Kapitel 9 (Interactive text processing) und Kapitel 16 (Data entry with menus) des Tutorials sind damit – was die Menüs angeht – nicht mehr aktuell.

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. Da das Tutorial mit diversen Beispiels-Datenbasen arbeitet, können die zugehörigen Dateien nicht mehr einfach (wie 1993) in *einen* Ordner gespielt werden. Hierbei würde es zu Konflikten (und Fehlermeldungen) kommen.

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`

[1] Leider stimmt die Seitenzählung nicht mit der gedruckten Fassung überein, da in der PDF-Datei die Seitenzahl 155 fehlt. Der Paginierungsfehler hat keine Auswirkung auf die Vollständigkeit des Textes.

Halbgraue Reihe

zur Historischen Fachinformatik

Herausgegeben von

Manfred Thaller

Max-Planck-Institut für Geschichte

Serie A: Historische Quellenkunden

Band 23

Matthew Woollard & Peter Denley

Source-Oriented Data Processing for Historians: a Tutorial for $\kappa\lambda\epsilon\iota\omega$

Max-Planck-Institut für Geschichte

In Kommission bei

SCRIPTA MERCATURAE VERLAG

St. Katharinen, 1993

© Max-Planck-Institut für Geschichte, Göttingen 1993
Printed in Germany
Druck: Pachnicke, Göttingen
Umschlaggestaltung: Basta Werbeagentur, Göttingen
ISBN: 3-928134-92-2

Contents

Introduction

Recent years have witnessed a lively debate in the historical computing community as to whether the relational database management systems that dominate the worlds of commerce and administration are appropriate to historical research. Closely related to that debate, and no less vigorously disputed, is the pragmatic question of whether historians who use computers in their research should produce their own software or leave it in the hands of commercial manufacturers. Currently it is considered sensible for historians to describe their problems to the developers, leaving the former to do their research and the latter to deliver a product.

One database management system has been created by an historian for historians. Since 1978 Dr Manfred Thaller of the Max-Planck-Institut für Geschichte in Göttingen has been developing and enhancing κλειω. In its original form (known as CLIO), this system could only be used on the UNIVAC 1100 series using a Latin command language and documentation that was available only in German. These features did not help to make it widely available. By 1987 a PC version was released, but with the same constraints of command language and documentation. It is therefore unsurprising – though regrettable – that while κλειω has been widely used for over a decade in the German-speaking historical world, among non-German speakers it is known only by reputation if at all.[1] The interest in the system as described in various English-language publications has been matched only by the frustration of those who are unable to try it out.

The English version of κλειω that is presented here is the product of an initiative to make κλειω more widely available at last. It has been made possible by the generosity of the Royal Historical Society, the British Academy, the Max-Planck-Institut für Geschichte at Göttingen, the Committee for Advanced Studies of the University of Southampton and the Faculty of Arts of Queen Mary & Westfield College, University of London. It is hoped that it will at least now be possible for historians to assess κλειω for themselves, and to make more informed judgements on the controversial issues raised above.

[1] A French introduction was also produced in 1990; Josef Smets, *Créer une base de données historiques avec* κλειω. Halbgraue Reihe zur historischen Fachinformatik, A7 (St. Katharinen, 1990).

The principles of κλειω

κλειω – pronounced to rhyme with Ohio – is a complex and versatile system that has been designed specifically to cater for the computing needs of historians, particularly in those respects in which commercial software does not cater for them. As κλειω has been under continuous development since the late 1970s, the range of features offered and areas of historical computing covered have naturally expanded, and the implementation of new features has to an extent altered the direction of the project, which since 1989 has been known as 'The Historical Workstation Project'. The underlying principles that motivated κλειω's development and have guided its implementation have, however, remained largely unchanged. They can be listed as follows:

1. A source-oriented approach

κλειω allows the historian to enter historical sources into the computer in a form which is as close as possible to that of the original material, preserving features within the data where conflicting interpretations are possible. At the most obvious level this means that, for example, original spelling can be retained, as can original currency. There are two implications of this principle. First, that a source-oriented approach to data processing should be followed, i.e. a minimal amount of coding or mark-up should be performed on a source before analysis. Second, that the methods of analysis of a particular source need not, indeed should not, be chosen before analysis. For example, κλειω allows the user to make decisions *after* data input about possible semantic differences in the written description of an individual's occupation due to spatial or temporal factors. Similarly, κλειω provides the facility to input unbroken strings of text so that information can be "automatically" abstracted from that text, so that the method of analysis does not depend on the method of entering the data.

In other words, κλειω allows the user to enter data in a format that underlies the source rather than the user's intentions with that source. This is made possible by allowing the user to input the data in a flexible format within a structure based on that source, rather than one designed with the principles of formal database design in mind.

In effect this means that κλειω has the ability to accept all forms of historical source material in a format that relates to the source. Historical sources can present information in forms that are very hard to reconcile with the conventions of a traditional database. On the simplest level, a census return may contain an entry containing two distinct occupations. This can be described as a multi-value variable. In κλειω these *entries* can be combined in such a way that they remain in context but can be made logically equivalent for processing. *Elements* can contain any number of entries, which in turn can have different *aspects* (e.g. the original spelling, or the editor's comments, can be stored alongside the main version to be processed), *views* (e.g. Latin and vernacular equivalents of the same data could be stored as alternative views) and *visibility* (a quantitative estimate of the value or reliability of the information). These features allow fuzzy data (for example, a surname which might also represent an 'occupation' — such as Fletcher) to be defined as such, the user then having to choose at the information retrieval stage how to interpret such data. Elements are contained in *groups* of information (not unlike records in relational database design) which are related to each other in a hierarchical fashion, i.e. are logically subordinate or superordinate to each other. However, these groups can contain elements with the same name, so that κλειω can 'implicitly join' these related elements where 'the system suspects that the user might eventually be interested to view the two of them as being one and the same type of reality'.[2] The size of a database is limited only by the capacity of the machine being used; the complexity of a database is virtually unlimited (elements can contain up to 2 million characters; there can be up to 32,000 different element names, and 32,000 different groups).

All these concepts add up to κλειω's data model which has been described as a 'semantic network tempered by hierarchical considerations'.[3] This model seems most suited for the representation of complex historical data. Moving from the technical computing metaphor to the historical, it also displays close parallels with the concept of scholarly editing. It is no accident that recent literature on κλειω has stressed the concept of 'The Database as Edition'.[4]

2. A logical environment

κλειω provides a working environment where tools which implement solutions for history-specific problems can be developed and used. Examples of such problems might be the recognition of names with variant spellings or the solution of chronological problems encountered in historical data. κλειω contains a number of basic algorithms which allow it to handle unprocessed historical data. If we input data in a form as close to the source

[2] Manfred Thaller, 'What is "Source Oriented Data Processing"; What is a "Historical Information Science"?', paper given to conference on 'New Information Technologies in Historical Research and Teaching', June 1992 in Uzhgorod, Ukraine, published in Russian in *Istoriia i comp'iuter. Novye informatsionnye tekhnologii v istoricheskikh issledovaniiakh i obrazovanii*, eds. Leonid I. Borodkin & Wolfgang Levermann. Halbgraue Reihe zur historischen Fachinformatik, A15 (St. Katharinen, 1993), pp. 5–18. English typescript, p. 4.

[3] Manfred Thaller, 'The Historical Workstation Project', *Computers and the Humanities*, 25 (1991), pp. 149–62 (p. 155).

[4] Ibid., pp. 156–59; see also Susanne Botzem, Ingo H. Kropač, 'Integrated Computer Supported Editing, Approaches and Strategies', in *Historical Social Research/Historische Sozialforschung*, 16:4 (1991), pp. 106–15, and Susanne Botzem, Ingo H. Kropač, 'As You Like It or Archiving, Editing and Analysing Medieval Manuscripts', in *Histoire et Informatique*. V^{e} Congrès 'History & Computing', 4–7 Septembre 1990 à Montpellier, ed. J. Smets (Montpellier, 1992), pp. 267–78.

as possible, it will raise certain historical problems which, while they could be untangled or solved by the historical researcher, would more efficiently and effectively be solved by computer. κλειω provides these tools in the form of algorithms which can be altered by the researcher to cope with the problems inherent in their particular form of data. These tools include a variant on the well-known Soundex algorithm along with an algorithm for the pre-treatment (i.e. before Soundex) of phonologically or orthographically similar names, the Guth algorithm (which quantifies the degree of similarity between two character strings in numerical terms) (both for nominal record linkage), algorithms to assist in the conversion of different calendar systems to a pre-defined format, coping with Roman, Mosaic, Islamic and Byzantine calendars as well as dates in the format <Period> <Feast Day> <Year> (e.g. '4 days before Maundy Thursday 1853'), and algorithms to cope with complex numbers mainly to assist in the interpretation of different currency systems.

Some of these algorithms reside in the system; others have to be more fully defined so that they can become an integral part of the database, affecting data only when required. This means that assumptions about historical data can be administered within a database but entirely independently of the data itself. These algorithms, which together amount to κλειω's *logical environment*, can be seen as a form of expert system, which can be developed to make full use of context-sensitive historical data.[5]

3. Functionality

κλειω provides a full set of basic database operations, such as information retrieval and report generation, accessing the whole structure of a database which is based on an historical source. κλειω administers sources effectively and efficiently within databases, allowing the user to navigate easily through highly complex structures, and producing results to complex and unwieldy queries. Different databases can be linked (for example, two different databases can be accessed with a single query) or joined (in such a way as to allow data to be retrieved from either or both); queries can be framed in such a way as to produce output in a variety of formats.

4. Integration

κλειω aims to provide, where appropriate, simple integrated versions of applications which would usually have to be realised within different software modules. This goal has been achieved in two main areas, full-text analysis and mapping. κλειω can perform a limited number of operations relating to full-text material, including a system of embedded

[5] Manfred Thaller, 'Databases and Expert Systems as Complementary Tools for Historical Research', *Tijdschrift voor Geschiedenis*, 103 (1990), pp. 233–47 (pp. 240–42).

classifications within a text, as well as searching facilities which allow the user to integrate full-text retrieval along with more structured material. Choropleth and distribution maps can be produced, provided that digitised coordinates are prepared for κλειω to administer.

5. Compatibility

κλειω provides interfaces to other general-purpose packages. For example, it is possible to extract material from structures of source-material into statistical cases, which can be immediately understood by statistical processing software such as SPSS and SAS. κλειω also transforms data within complex structures into a format that statistical applications can understand, using flexible information retrieval and report generation facilities and its ability to combine information from different subsets of a database.

The Historical Workstation Project

As the κλειω project broadened in scope, fanned out and developed, it took on the aspect of a collaborative venture, inspired and led from Göttingen by its creator, Manfred Thaller, but augmented by research contributions from elsewhere. The result is that a number of refinements, additional algorithms or related packages have been developed. This is not the place to trace the history of all these developments, several of which are still in progress. (Those interested will be able to find further information in the bibliography of English-language publications relating to κλειω at the end of this book.) Below are described those initiatives which have led to software releases which are already available.

Lemmatisation

The full-text system of κλειω has been enhanced by the integration of a Latin lemmatization program, developed in Rome, which is available on request.[6]

[6] Andrea Bozzi & Giuseppe Cappelli, 'A Latin Morphological Analyser', in *Data Base Oriented Source Editions*. Papers from two sessions at the 23rd International Congress of Medieval Studies, Kalamazoo, 5–8 May 1988, ed. M. Thaller, pp. 47–54.

StanFEP

Alongside κλειω, Manfred Thaller developed a program in the 1980s called the Standard Format Exchange Program. The purpose of this is to enable historians (and, of course, others) to mark up electronic documents in a way which keeps integrated the various versions that might be deemed necessary for the various tasks that the scholar might wish to perform (e.g. diplomatic transcription, pre-edition, coding for use in a database system, final edition). The current version of StanFEP is included on disk with κλειω. The software uses English commands and conventions, though unfortunately the manual and tutorial still await translation from German.[7] A second stage of development is currently in progress (a collaborative project of the Max-Planck-Institut für Geschichte, Göttingen, and the Historical Informatics Laboratory, Lomonossov University, Moscow).

κλειω Image Analysis System (κλειω IAS)

The most radical addition to the original scope of the software has been a program which applies the principles of κλειω to the processing of images. In collaboration with the Max-Planck-Institut für Geschichte, Göttingen, the Institut für Realienkunde des Mittelalters und der Frühen Neuzeit, Krems, has developed κλειω IAS. In this system the textual description of an image can be bound to the image itself (rather like hypertext) and displayed simultaneously together. A variety of tools for digital image analysis is also provided for the enhancement of images, for immediate image retrieval and most recently pattern recognition.[8]

[7] The manual is Kathrin Homann, *StanFEP. Programm zur freien Konvertierung von Daten*. Halbgraue Reihe zur historischen Fachinformatik , B6 (St. Katharinen, 1990); the tutorial is Martin Gierl, Thomas Grotum & Thomas Werner, *Der Schritt von der Quelle zur historischen Datenbank. StanFEP: Ein Arbeitsbuch*. Halbgraue Reihe zur historischen Fachinformatik , A6 (St. Katharinen, 1990). An English introduction is Kathrin Homann, 'StanFEP — Standardization without Standards', in *Histoire et Informatique*. V^{e} Congrès "History & Computing", 4–7 Septembre 1990 à Montpellier, ed. J. Smets 1992), pp. 289–99.

[8] The manual is Gerhard Jaritz, *Images. A Primer of Computer-Supported Analysis with* κλειω *IAS*. Halbgraue Reihe zur historischen Fachinformatik, A22 (St. Katharinen, 1993). See also Manfred Thaller, 'The Processing of Manuscripts', in *Images and Manuscripts in Historical Computing*, ed. M. Thaller. Halbgraue Reihe zur historischen Fachinformatik, A14 (St. Katharinen, 1992), pp. 41–72, and idem, 'The Archive on the Top of your Desk? On Self-Documenting Image Files', in *Image Processing in History: towards Open Systems*, eds. Jurij Fikfak & Gerhard Jaritz. Halbgraue Reihe zur historischen Fachinformatik, A16 (St. Katharinen, 1993), pp. 21–44.

Current versions of κλειω

The principles of image processing are not so dissimilar to those for other types of data as to make it necessary for κλειω IAS to be a separate program; on the other hand image processing obviously makes much heavier demands of computing resources. As a consequence, κλειω currently runs in two versions.

- Version 5.1.1 comprises the command language and the non-graphical menu systems and is machine independent. Because of this machine independence, if a database should become too large for a PC, a user can move to a more powerful one or to a mainframe to continue work.

- Version 6.1.1 (κλειω IAS) is identical to Version 5.1.1 but limited in use to a number of UNIX platforms, and it also provides a graphical user interface which is geared to the handling and processing of images. The manual for κλειω IAS is however included with Version 5.1.1, because it contains a high proportion of material relating to the present version of the system, and because we believe that all κλειω users may find it helpful to see how κλειω handles image data. We also include this volume as a taster of the Windows NT version of κλειω which should be released during 1994, which will not just handle images but also provide a graphical interface for the whole of the κλειω software.

It should also be pointed out that in both 5.1.1 and 6.1.1 the Latin/German and the English versions are integrated; that is, the user is offered a choice of language at installation. It is expected that the Latin/German version will be supported for a period of three years only.

How to use this book

It will already be apparent to anyone who has browsed through the pages of this book that κλειω is both rich in what it offers and demanding of the user. κλειω cannot be 'picked up' in an afternoon of dexterous exploration of pull-down menus and help screens. Some of the underlying concepts are complex, and there are no short cuts to learning the command language, or at least those parts of it which are relevant to the work of the researcher.

Some of the most successful κλειω users have been those who have taken one of the many κλειω courses that are run in various parts of Europe. The κλειω Support Team (see below, p. 345) will gladly provide details of such courses; increasingly they are being offered om

English, and are using the English data sets that have been prepared in connection with this volume and the current release of the software. This book is however intended as a practical guide for *all* those wishing to learn how to use κλειω, whether by taking a course or on their own. Each of the three parts of the book discusses separate aspects of κλειω.

- Part I, "Getting Started", introduces the basic concepts and terminology of κλειω and the simple accessing of a κλειω database, and shows how simple queries can be effected.

- Part II, "κλειω Basics", introduces the most frequently used features of κλειω. All the concepts described in this part are likely to be used on a regular basis. These include methods of creating a database, more complicated query facilities, the integration of knowledge within databases, the processing of textual material, the creation of 'look-up tables' for the coding and classification of data, and also some features to produce more sophisticated output.

- Part III, "Specialised Features", introduces more complicated features of κλειω, including nominal record linkage and automated cartography. It also introduces techniques for family reconstitution. Also included in this part are more advanced concepts of database design and an alternative method of constructing databases.

In each section the points made and the techniques introduced are illustrated by examples and exercises. The *practice databases and exercise files* are an integral part of the tutorial and should be installed with the software. The installation notes provided with the software will explain how to install the software and the tutorial files. Answers to the exercises can be found at the end of the book.

Finally, it is worth stressing that this book should be used in conjunction with the Reference Manual,[9] the full formal description of κλειω, which provides the fuller description of the many features of κλειω, including many which there was not space to describe here. The present volume provides frequent cross-references to the Reference Manual.

[9] Manfred Thaller, κλειω. *A Database System*. Halbgraue Reihe zur historischen Fachinformatik, B11 (St. Katharinen, 1993).

Acknowledgements

The debts of the ‘English Version of κλειω’ Project are many. The project has been made possible by the generosity of the Royal Historical Society, the British Academy, the Max-Planck-Institut für Geschichte at Göttingen, the Committee for Advanced Studies of the University of Southampton and the Faculty of Arts of Queen Mary & Westfield College, University of London. It was steered and monitored by an Advisory Committee, which allowed us to benefit from the expertise of Frank Colson (Chairman), John Childs, Jean Colson, Jay Hammond, Tom James, Derek Keene, Seamus Ross and Kevin Schürer. We benefited greatly from the hospitality of the Institute of Historical Research throughout the project. Bill Maslen of The Word Gym performed the original translation of the Reference Manual and the error messages of the software; he is not responsible for any errors and inelegancies which we have managed to introduce since then.

As far as concerns this volume specifically, we would like to thank the Public Record Office for permission to reproduce the 1881 enumerator’s return in Chapter 4, and the Hampshire Record Office for their assistance in locating material to be used in the probate database. The Winchester Data Project collaborated closely from the beginning with the preparation of data sets, and we would like to thank Anne Bailey, Jean Colson and Tom James in particular for their help in this. We must acknowledge our debt to the authors of the German κλειω tutorial, Peter Becker and Thomas Werner, and to Gabriele Gross, author of the introduction to the menu system. The students on the κλειω course at the Institute of Historical Research exhibited patience in their role of guinea-pigs and provided useful feedback. Humphrey Southall advised expertly on the cartographical side; Phillip Woollard created the glossary for the probate database, while Wendy Hefford, Deputy Curator, Textiles and Dress, Victoria and Albert Museum, explained some of the more technical terms found in that glossary. Jay Hammond painstakingly read the whole manuscript and offered constructive suggestions, while John Davies, as part of an industrial placement from Middlesex University and on secondment from London Guildhall University, worked through the entire draft and did all the exercises.

Finally two people should be thanked particularly warmly. Frank Colson’s drive and spirit of enterprise have been critical; without him the project would never have got off the

ground. Manfred Thaller taught a large part of the course and worked closely with the project at all stages. Those who know him will not be surprised to read that his encouragement, stimulus and expert advice were matched only by his modesty about it.

London, November 1993

PART I

GETTING STARTED

This section aims to introduce the basic concepts and terminology of κλειω, and to give the new user some practical experience of querying databases. Users are strongly encouraged to become completely familiar with these chapters before proceeding to the next section.

Chapter 1

Basic concepts

1.1 Data structures

For the newcomer, the most immediate difference between κλειω and conventional data processing software concerns data structures. In κλειω, there are three forms of information: documents, groups and elements.

Documents

A *document* is the highest level of information in a database. It is so called because it often corresponds to historical documents — for example, each census list, or each parish register might be a κλειω document. κλειω databases can contain different document types. We will meet this later on.

Groups

Documents consist of *groups*, which are abstract groupings of the information in the document. Groups are in some respects similar to records, rows or tuples in traditional data processing. In all κλειω databases each group has a specific relationship with every other group in this database. These relationships are defined by the user, and can be complex, with many groups dependent on a higher group, and with the same group name recurring in different parts of a database. We will meet such complexities, which illustrate the real power of κλειω, later on.

Elements

Groups are made up of any number of *elements*. Elements are similar to columns, variables, fields or attributes in traditional data processing. The length of an element is variable; they can range from 2 million characters in length to one character. κλειω supports up to 32,000 different element types within one database.

Elements are normally made up of *entries*, which are the smallest unit of information that κλειω can access. Entries have no equivalent in traditional data processing, as each element can contain more than one entry. Effectively this means that κλειω allows elements to contain many entries which have the same (or if specified a different) logical rank. For instance a person mentioned in an historical source may have two occupations. These may be represented in κλειω as **`schoolmaster;vicar`**. **`schoolmaster`** and **`vicar`** are both entries in an element called occupation.

A further sophistication is that entries can have different *aspects*. As well as the basic information, they can be accompanied by a comment or by the original text. κλειω can be told which aspect or aspects of the entry to process at any time. We will return to aspects in Chapter 4, Section 4.3.7.

1.2 The command language

κλειω is operated by a command language, which to an extent has to be learnt. Some features of κλειω are operated by a menu system, but this is of no help unless the user is fully aware of the command language. What follows is an introduction to the basic terminology of the command language.

The simplest possible κλειω *command file* looks like this:

```
query name=burial
write
stop
```

Line by line, this command file can be 'translated' into English as follows:

`query name=burial`
: I want to know about a database called burial,

`write`
: I would like the information selected to be displayed on the screen,

`stop`
: When all the information has been displayed, stop.

Every κλειω command file consists of one or more *tasks*. A task is a sequence of instructions required to provide the system with all the information it needs to select data from a database and present it as a result.

Each line in a κλειω command file is an *instruction*. Instructions must begin with a *command word* which must start in the first column of a new line. Immediately after the command word in an instruction there may be a *specification*. The specification for each instruction is composed of a series of *parameters*. Parameters are made up of *parameter names* and *parameter values*. The command word and the specification must be separated by at least one space.

In the above example, each line is an instruction.

```
query      name  =   burial
   |          |            |
command    parameter    parameter
word       name         value
              \          /
             specification
```

`write` and **`stop`** are also command words.

Another example of a parameter is **`target=`**. For example if we wanted κλειω to send the results of that task to a file we could specify this at the end of the sequence of instructions. In this task **`target=`** is the parameter name and **`"filename"`** is the parameter value:

```
query name=burial
write
stop target="filename"
```

Note that all of the commands described in what follows must be written in the form of a command file. Any text editor or word processing program can be used to create these command files before importing them to κλειω. These files must be in the 'extended ASCII character set' as defined by IBM, unless the configuration program has already been run. (This program allows you to use redefined character sets. Notes on the configuration program can be found in Appendix B of the Reference Manual.) To this end it is recommended that we use a screen editor to produce these command files. An example of such an editor is EDIT, which comes with version 5.0 or above of MS-DOS. To use EDIT, at the prompt type:

```
edit
```

This will take you into the editor. There should be a help screen, which you can read to get some assistance. Different language versions of EDIT use different commands, but the menu system should be easy enough to understand with some practice. To edit a file that already exists, type **`edit`** at the prompt followed by the name of the file you wish to edit. For example,

```
edit ex1.1
```

(This file will not be on your computer until κλειω has been installed.) You will get an empty file which you can start to work on.

1.3 Running κλειω

Installing κλειω

Before κλειω is run, it has to be installed. As the details of how to do this can vary with different releases, please refer to the notes accompanying the software disk, distributed separately. The installation proceedure will also copy all the databases and exercises described in this volume into a directory called tutorial.

Calling up κλειω

On a computer running MS-DOS, κλειω is always accessed from the prompt. At the prompt enter:

```
kleio
```

This will bring you into κλειω's menu-driven interface. This will not be used until later in the tutorial, so exit from this interface by pressing the ESCape key.

In order to import a command file to κλειω one must type:

```
kleio <input file>
```

If one wanted the result of a task sent to another file one would have to type:

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

Compiling a database

One further preparatory step is necessary. Before a database is used, it has to be *compiled*; that is, the data and the rules which explain it to κλειω need to be put into the system. The design and creation of databases will be discussed later; here it will suffice to carry out one example of compiling a database.

Files declaring the structure of a database are conventionally given the suffix **.mod** (short for 'model'), while files containing data are conventionally given the suffix **.dat**. To compile the database 'burial', our first sample database, the following two steps are necessary:

At the DOS prompt, first type:

```
kleio burial.mod
```

This 'loads' the declaration file, **burial.mod**, into κλειω. Note that the program checks the legality of the commands; if there were any 'syntax errors' it would fail to load and report the errors. As this file is legal, κλειω reports that it has executed the task. Next, type

```
kleio burial.dat
```

κλειω now reads the data from the data file, **burial.dat**. Again, it would report any 'illegal' or ambiguous data at this stage, and would ignore those entries.

Once these two files have been read by κλειω the database burial is compiled and ready for use.

1.4 A sample database

We will be working with a range of example databases during the course of this tutorial. In this introductory part, we will be using burial and baptism records taken from the parish register at Winchester Cathedral between 1619 and 1630. They are extremely simple data sets in their structure, and do not illustrate particularly well why one might wish to use κλειω as opposed to a conventional database management system. Rather they have been chosen for their suitability for simple exercises, and should help the novice acquire the basic skills of querying and extracting information for analysis.

The Winchester Cathedral parish registers are typical of such registers throughout England. For more information on this kind of source, see the suggestions for further reading at the end of this chapter.

In the burial registers, each item consists of the following information:

- Name of person buried
- Date of burial

These are the only two items that are consistent throughout the data.

Some or all of the following information is also included:

- Occupation of person to be buried
- Place of abode of person
- Name of a relation and the relationship
- Occupation of relation
- Place of burial within the Cathedral
- Titles of either the individual or their relation.

A typical excerpt looks like this:

1620	*Apr 2*	*Simon, s of Mr Harward, prebendary*
	May 18	*Sibill, w of Mr William Cole*
	Sept 24	*Anne, d of Mr William Trussell, clerke*
	Nov 16	*James Sutton, belringer*

If one were preparing this data for use in a conventional database system, one would normally include all the information about each individual entry within one record, something like this:

Date	*Fname*	*Sname*	*Occn*	*Relship*	*Relfname*	*Relsname*	*Reloccn*
02.04.1620	Simon			son		Harward	prebendary
18.05.1620	Sibill			wife	William	Cole	
24.09.1620	Anne			daughter	William	Trussell	clerk
16.11.1620	James	Sutton	Bellringer				

In κλειω it is possible to put the information about the two separate people into different records by making the relation dependent on the person who was buried. This is done by creating two groups, one containing information about the people who were buried, and another, dependent on the first, containing information about people who were related to those buried. The structure of this database is thus very simple:

```
doc
 |
 p
 |
relp
```

where doc stands for document, p for person and relp for related person. The highest level of this structure, 'doc', in this case relates to details about the source. p and relp are groups. (As you might already have guessed from the diagram, a document is in fact a special kind of group. This is discussed further in Chapter 4.)

The group p contains the following elements (the right-hand column gives the first of the four dead people by way of example):

status	Male, dead
title	
firstname	Simon
surname	
subtitle	
relation	Son
occupation	
abode	
place	
burialdate	2 Apr 1620

The group relp contains the following elements:

status	Male
title	Mr
firstname	
surname	Harward
subtitle	
occupation	prebendary

Relationships exist between groups. In this case Simon Harward is the son of Mr Harward the prebendary. This is a hierarchical structure.

```
p
|
|—relp
```

κλειω is not limited to hierarchical structures and it does not handle the data as a hierarchy. κλειω produces a network which links all of the data within a database.

To conclude this introduction, there follow some examples of how this data can be presented to κλειω and how κλειω lists it when it is asked for by the simple command file presented above.

The structure is expressed in the data file in the following way. Groups begin at the beginning of each line; group names terminate with a **$** sign, and that is followed by the elements. These are separated by the delimiter /. The excerpt given above might be entered like this:

```
p$status=dm/firstname=Simon/surname=Harward/relation=son
  /burialdate=2 Apr 1620
relp$status=m/title=Mr/surname=Harward/occupation=prebendary
p$status=fd/firstname=Sibill/surname=Cole/relation=wife
  /burialdate=18 May 1620
relp$status=m/title=Mr/firstname=William/surname=Cole
p$status=fd/firstname=Anne/surname=Trussell/relation=daughter
  /burialdate=24 Sept 1620
relp$status=m/title=Mr/firstname=William/surname=Trussell
  /occupation=clerke
p$status=md/firstname=James/surname=Sutton/occupation=belringer
  /burialdate=16 Nov 1620
```

Alternatively, since the information appears in the original in very structured form, if κλειω is told in what order to expect the elements to appear, it could be entered like this:

```
p$dm//Simon/Harward//son///2 Apr 1620
relp$m/Mr//Harward//prebendary
p$fd//Sibill/Cole//wife///18 May 1620
relp$m/Mr/William/Cole
p$fd//Anne/Trussell//daughter///24 Sept 1620
relp$m/Mr/William/Trussell//clerke
p$md//James/Sutton///belringer//16 Nov 1620
```

Design and entry will be discussed in Chapter 4.

We can now return to the simple command file encountered above:

Example 1.1

```
query name=burial
write
stop
```

To run this program, type **kleio** at the DOS prompt followed by the name of the command file, in this case ex1.1 (which stands for 'example 1'):

```
kleio ex1.1
```

The listing of the full database is now displayed on the screen. As this is quite long, it scrolls down quite rapidly. To freeze the screen temporarily, press the Pause key. To continue scrolling, press any other key.

Note the form in which the data is displayed. Typical output looks like this:

```
p (7 = "p-7")
                burialdate    2.4.1620
                status        male, dead
                firstname     Simon
                surname       Harward
                relation      son

        relp (1 = "rel-1")
                        status        male
                        title         Mr
                        surname       Harward
                        occupation    prebendary
```

Each block represents a group; within the group the element names are given on the left and the entries are given on the right. The relative indentation of the groups indicates the hierarchical relationship between them. Each group is headed by a line which indicates the position of the group in the database. We will examine these lines more closely later on.

Summary

This chapter has introduced

- Data structures; documents, groups and elements. Elements can have multiple entries, and different 'aspects'.
- The command language; command files consist of tasks, which in turn consist of instructions, which consist of command words, which may have specifications attached to them in the form of parameter names and values.
- How to invoke κλειω.
- How to compile a database.
- The sample database 'burial'.

The following chapter discusses ways of querying this database.

Further reading

The following titles may be of interest for students of parish registers:

M. Drake (ed.), *Population Studies from Parish Registers. A Selection of Readings from Local Population Studies* (Local Population Studies, Matlock, 1982).

K. Schürer, 'Historical Demography, Social Structure and the Computer', in *History and Computing*, eds. P. Denley & D. Hopkin (Manchester University Press, Manchester, 1987), pp. 33–45.

K. Schürer, J. Oeppen & R. Schofield, 'Theory and Methodology: an Example from Historical Demography', in *History and Computing II*, eds. P. Denley, S. Fogelvik & C. Harvey (Manchester University Press, Manchester, 1989), pp. 130–42.

E. A. Wrigley & R. S. Schofield, *The Population of England 1541–1871: a Reconstruction* (Edward Arnold, London, 1981, and revised version Cambridge University Press, Cambridge, 1989).

Chapter 2

Basic information retrieval and display

2.1 The **query** and **write** commands

In the simple command file encountered in the last chapter (**ex1.1**) the **query** command was introduced. This is the most basic way of extracting information from a database, and is the building-block for quite complex retrieval. This chapter explores it in detail.

The example given –

```
query name=burial
```

– is the most simple form of a **query** command that is acceptable to κλειω. It is necessary to specify the name of the database being queried. The result of Example 1.1 is that *all* the information in the database is retrieved and displayed. As a rule of thumb, the less that is specified in the task, the more information is obtained.

The **name=** parameter usually follows the **query** command. It takes as a parameter value the name of the database that you want to process.

2.1.2 The parameter `part=`

It is much more usual to wish to interrogate only parts of the database. To do this, a number of tools are available. The simplest of these is the parameter **`part=`**. In the following task we will be modifying the first task to look only at those people within the database who appear as relations to the deceased (who are mostly children in such cases). In this case we are asking κλειω to look at a database and then only interrogate those groups called relp. This is known as a *path*. It really means go to the highest level (by default) and then follow a path to the group relp.

Example 2.1

`query name=burial;part=relp`
: I want to know about a database called burial and I am only interested in those people who appear as relations of the deceased (i.e. who feature in the group relp).

`write`
: I want all the information that refers to these people displayed on the screen.

`stop`
: When this is done, stop.

Note a vital new rule introduced in this task:

> When a second parameter is used in an instruction it must be separated from the first parameter by a semi-colon.

Run this task by typing **`kleio ex2.1`**. Note that the full information is given for all those, and only those, who appear as relations in the data.

```
p (3 = "p-3" : relp (1 = "rel-1")
        status          male
        title           Dr
        surname         Hilton

p (4 = "p-4" : relp (1 = "rel-1")
        status          male
        title           Dr
        surname         Alexander
```

The following task displays all of the information about people who have an occupation:

Example 2.2

`query name=burial;part=:occupation`
: I want to know about a database called burial and I am only interested in those people whose occupation is given (i.e. who have an element occupation).

`write`
: I want all the information that refers to these people displayed on the screen.

`stop`
: When this is done, stop.

Run this task by typing **`kleio ex2.2`**. Full details are listed for all those people whose occupation is given.

This time we have asked κλειω to select records which have an element. This task introduces another essential concept:

> For κλειω to understand that 'occupation' is a name for an element, it must be preceded by a colon. The colon denotes that the parameter value is an element.

There is one problem with the last example. The search yielded all people with occupations, whether they have died or whether they are relatives. The following task shows how we might display only the occupations of those people who died:

Example 2.3

`query name=burial;part=p`
: I want to know about a database called burial and I am only interested in those people who died.

`write part=:occupation`
: I only want the information about those people's jobs to be displayed on the screen.

`stop`
: When this is done, stop.

Run this task by typing **`kleio ex2.3`**.

The **`part=`** parameter can thus be used to restrict queries or displays to a specific part of the database, and it can be used with the **`write`** command as well as with the **`query`** command (indeed it can be used with a number of commands).

If one wanted the occupations of those people who were related to the dead people the following task would find them:

Example 2.4

`query name=burial;part=relp`
: I want to know about a database called burial and I am only interested in those people who appear as relations of the deceased (i.e. who feature in the group relp).

`write part=:occupation`
: I only want the information about those people's jobs to be displayed on the screen.

`stop`
: When this is done, stop.

Run this task by typing **`kleio ex2.4`**.

The following task asks for other information about dead people to be displayed.

Example 2.5

`query name=burial;part=p`
: I want to know about a database called burial and I am only interested in those people who died.

`write part=:firstname,:surname,:occupation`
: I want those people's first names, surnames and occupations to be displayed on the screen.

`stop`
: When this is done, stop.

A new syntactical feature is introduced here:

When using the **`part=`** parameter with the **`write`** command, any combination of applicable elements may be displayed. In the command line they *must* be separated by a comma.

Run this task by typing **`kleio ex2.5`**.

There are a number of other parameters which work with the **`write`** command. These control the scope of the output on the screen or on paper. Some of these will be discussed in Chapter 5; details of others can be found in the Reference Manual in Section 8.3.1.

2.2 The element function `:each[]`

If you wanted to display all the elements directly related to a group you could set the following task:

Example 2.6

`query name=burial;part=:occupation`
I want to know about a database called burial and I am only interested in the parts which have the element occupation.
`write part=:each[]`
When all those items have been extracted from the database display all the information in all the groups where there is an occupation.
`stop`

`:each[]` is an example of an element function. It is defined as the set of all the elements in the last group to be found. It is defined like this because in the first line of this task κλειω doesn't look for all the elements called occupation, but rather selects all the groups which contain the element occupation. Run this task now.

Element functions are built-in functions which can appear at a position where an element identifier (e.g. occupation) might also appear.

Note carefully that **`:each[]`** is preceded by a colon. This is because it represents an element.

This is the simplest example of an element function. In fact, its effect is identical to that of the task below (**`ex1.3`**), as it selects exactly the same information, and displays the result in the same way.

```
query name=burial;part=:occupation
write
stop
```

On its own, the **`:each[]`** function used with the **`write`** command coincides with the 'default' setting of the **`write`** command. Nonetheless an important principle is being introduced here. The element function **`:each[]`** is a way of specifying something – in this context, display all the information – which in these examples is identical to the 'default' setting of the **`write`** command. But there are instances in more complex structures where it is necessary to call the function, and of course by 'filling' the square brackets, functions can be made more sophisticated. We will meet examples of these later.

From here on we shall not be giving instructions about running tasks. We suggest you read each section before running the appropriate task. Only those tasks headed with the title 'Example' are included on the tutorial disk. All others you will have to type in yourself. However, we recommend that you type in all the examples yourself, to get practice using the commands.

2.3 Querying with conditions

Rather than ask κλειω to produce all the information about all the people who are mentioned in the database as having a job, one can ask κλειω just to produce the information about a specific element. For instance, if one wanted to produce all the information about all the people who were prebendaries, one would use the following task.

Example 2.7

`query name=burial;part=:occupation="prebendary"`
  I am interested in a database called burial. I am only interested in that part of the database where there is an occupation and where that occupation is prebendary.

`write part=:each[]`
  When all those groups have been extracted from the database display all the information in each group where a prebendary occurs.

`stop`

The phrase **`="prebendary"`** is a *condition*.

Conditions can be added to the end of any *path definition*.

2.3.1 Conditions and logical operators

Logical operators (also known as Boolean connectors) are used to modify conditions. κλειω understands the three usual Boolean arguments: **`and`**, **`not`** and **`or`**.

Any combination of these three keywords can be used to modify a condition, and they can be used in conjunction with brackets to create complex conditions.

If the logical operator **and** connects two subconditions they must both be true for the condition to be satisfied.

If the logical operator **or** connects two subconditions one must be true for the condition to be satisfied.

If the logical operator **not** appears in front of a subcondition, it must be false in order for the condition to be satisfied.

Brackets modify conditions using multiple operators. In κλειω the logical operator **or** takes precedence over the others. For example, "x or y and z" is the same as "x or (y and z)". See the Reference Manual, Section 8.1.2.1.3.

Here are some simple examples of the use of logical operators:

Example 2.8

query name=burial;part=:occupation="prebendary" or "belringer"
I am interested in the database burial and I am only interested in those people whose occupation is either "prebendary" or "belringer".
write part=:each[]
When all those groups have been extracted from the database display all the information in each group where either a prebendary or a bellringer occurs.
stop

Example 2.9

```
query name=burial;part=:occupation="prebendary" and
   :surname="Darrell"
```
I am interested in the database burial and I am only interested in those people whose occupation is "prebendary" and whose surname is "Darrell".
write part=:each[]
When all those groups have been extracted from the database display all the information in each group.
stop

Note the new feature in this example: the second line of the task is indented. This is because κλειω only recognises commands as starting at the beginning of a new line. This second line could start any number of characters out from the left margin. Normally, to make it clearer to the reader it is indented four or five characters. In this case, this command line is in fact short enough to fit on one line on-screen in a text editor, but it is too long to fit on a single line in the format we are using in this volume.

Example 2.10

```
query name=burial;part=:occupation=not "alderman"
```
I am interested in the database burial and I am only interested in those people whose occupation is not given as "alderman".
```
write part=:each[]
```
When all those groups have been extracted from the database display all the information in each group.
```
stop
```

The following example also demonstrates the use of the logical operator **`and`** as well as the keyword **`not`**. This task will display all those people who are prebendaries but whose surname is not "Darrell".

Example 2.11

```
query name=burial;part=:occupation="prebendary" and :surname=
   not "Darrell"
```
I am interested in the database burial and I am only interested in those people whose occupation is given as "prebendary" but whose surname is *not* "Darrell".
```
write part=:firstname,:surname,:occupation
```
When those groups have been extracted display the first names, surnames and occupations of those people.
```
stop
```

The `null` keyword

The following three examples all demonstrate the use of the keyword **`null`**. This keyword can be used to specify 'empty' elements. In this first example, κλειω is asked to display all the information from a group where people do not have an occupation:

Example 2.12

```
query name=burial;part=p:occupation=null
```
I am interested in the database burial and I am interested in all those dead people who, according to the database, do not have an occupation.
```
write part=:each[]
```
When all those groups have been extracted from the database display all the information in each group.
```
stop
```

(We have described **`null`** as a *keyword*. The term keyword is a non-technical word used to describe a subset of κλειω's command language.)

The following task produces a strange answer:

Example 2.13

`query name=burial;part=:occupation=null`
I am interested in the database burial and I am interested in all groups where the element occupation is empty.
`write part=:each[]`
When all those groups have been extracted from the database display all the information in each group.
`stop`

The reason why κλειω has produced no result in this case is that the system demands that a group is specified when the keyword **`null`** is used.

The example below demonstrates how the keyword **`null`** can be negated using the keyword **`not`**. This condition is thus treated as true if there are people with occupations that are 'not null', i.e. they exist.

Example 2.14

`query name=burial;part=p:occupation=not null`
I am interested in the database burial and I am only interested in those dead people who have an occupation. (i.e. the same as **`p:occupation`**)
`write part=:each[]`
When all those groups have been extracted from the database display all the information in each group.
`stop`

2.3.2 The element function `:total[]`

Run the following task:

Example 2.15

`query name=burial;part=:status="f" and :surname="Love"`
I am interested in the database burial and I am interested in all those people who are female and have the surname "Love".
`write part=:each[]`
When all those groups have been extracted from the database display all the information in each group.
`stop`

Now run the task below, which demonstrates the use of the element function **`:total[]`**.

Example 2.16

`query name=burial;part=:status="f" and :surname="Love"`
I am interested in the database burial and I am interested in all those people who are female and have the surname "Love".
`write part=:total[]`
When those groups have been extracted from the database display all the information in each group *and for all groups subordinate to the first group accessed.*
`stop`

The function **`:total[]`** can be defined as the set of all elements contained in the last group to be activated and in all the groups logically subordinated to the latter.

2.3.3 Conditions and comparison modifiers

As we have seen in **`ex2.7`**, if we ask for all the information about the people who have the occupation prebendary with the specification **`part=:occupation="prebendary"`**, all those whose occupation is "prebendary" is displayed. There is a slight problem here for within the source the given occupation of one prebendary is "prebend". This entry within the database is not retrieved as it does not correspond exactly to the character string "prebendary". If one were to run the following task we would be able to bring up the occupations of all those people who, in this list, are prebendaries, including the one listed as prebend.

Please note that from here onwards the 'translations' of the κλειω *queries will become scarcer (to save trees) and only relate to new items of information.*

Example 2.17

`query name=burial;part=:occupation="prebend"`
I am interested in the database burial and I am only interested in those people whose occupation contains the character string "prebend".
`write part=:occupation`
`stop`

If on the other hand one only wanted the information about those people in the database who were actually described in the source as "prebend", and not those whose occupation in the source contained the character string "prebend" (for example, people described as "prebendary"), we could phrase the task as follows:

Example 2.18

```
query name=burial;part=:occupation="prebend" equal
```
I am interested in the database burial and I am only interested in those people whose occupation contains the character string "prebend" and nothing else.
```
write part=:occupation
stop
```

By default κλειω when confronted by a condition assumes that the user wants the string to appear anywhere within an entry. As we have seen, one can ask for the string to be the entire entry, using the comparison modifier **`equal`**. If one wanted all those entries which *began* with the string "prebend", rather than those in which the letters "prebend" appeared anywhere within the entry, one could replace **`equal`** with the comparison modifier **`start`**; conversely, if one wanted all those occupations that *ended* with the string "prebend", one could replace **`equal`** with the comparison modifier **`limit`**.

The following two examples illustrate the usage of these terms.

Example 2.19

```
query name=burial;part=:occupation="prebend" start
```
I am interested in a database called burial and I am only interested in those people whose occupation begins with the character string "prebend".
```
write part=:occupation
stop
```

Example 2.20

```
query name=burial;part=:occupation="ary" limit
```
I am interested in a database called burial and I am only interested in those people whose occupation ends with the character string "ary".
```
write part=:occupation
stop
```

The last example did not produce a very helpful result. To broaden the search one might look for occupations which ended in the character string "er".

Exercise 2.21

```
query name=burial;part=:occupation="er" limit
```
I am interested in the database burial and I am only interested in those people whose occupation ends with the character string "er".
```
write part=:occupation
stop
```

The following task shows another instance where one might use an abbreviation. If we wanted information about people in the choir we might ask κλειω to run this task:

Exercise 2.22

```
query name=burial;part=:occupation="cho"
write part=:occupation
stop
```

Note that this too is of limited usefulness. This task has displayed "choir", "chorister" and "of the choir"; but it has also picked up "master of the choristers" (which we might have wanted) and even "schoolmaster" (which we most certainly did not want), while failing to pick up "chanter". If on the other hand we had specified that the string "cho" was to appear at the start of the string:

```
query name=burial;part=:occupation="cho" start
```

we would have eliminated "schoolmaster" and "master of the choristers", but we would also have eliminated "of the choir", and we would of course still not pick up "chanter". All this demonstrates that, while pattern-matching is useful where data is of considerable regularity, or where one is searching for elements which have been carefully specified, for something as complex as occupational data the historian needs more sophisticated tools. These κλειω provides. The first step towards these is introduced through a new command, **`index`**.

2.4 The `index` command

The following task produces a list of all the surnames of all the people who were aldermen.

Example 2.23

```
query name=burial;part=:occupation="alderman"
write part=:surname
stop
```

If we wanted to produce the same information but have it sorted into alphabetical order we can use the **index** command in place of the **write** command. This command sorts previously selected information into alphabetical order.

Example 2.24

```
query name=burial;part=:occupation="alderman"
index part=:surname
```

When the first command has been performed produce an alphabetical list of all of the contents of the surname element.

```
stop
```

The next example hints at the potential value of this command. It demonstrates how one can produce a list of all the surnames and first names of those people who were aldermen.

Example 2.25

```
query name=burial;part=:occupation="alderman"
index part=:surname;
   part=:firstname
```

When the first command has been performed produce an alphabetical list of all of the contents of the surname element. Also display the contents of the element firstname, after the surname and sort them into alphabetical order too.

```
stop
```

There are three things to note from this example:

- As there are so few people retrieved by this task it is impossible to see if the first names are also in alphabetical order or not. If there were more first names they would be sorted in alphabetical order.

- The **index** command must have a **part=** parameter for each logical column of retrieved material. Each **part=** parameter should be separated by a semi-colon.

- We have (again) used a continuation line in the second line of the task, even though the line is short. This practice is strongly recommended; it makes for clarity and ease of diagnosis when the task does not work as intended.

There is one other thing to note about the **index** command. Running a task like this:

Example 2.26

```
query name=burial;part=relp
index part=:occupation;
  part=:surname
stop
```

would *not* produce an index of *all* the people who are defined as related people. The task asks κλειω to produce an index of all people described as related people who have an occupation. Because κλειω has specifically been asked to sort by occupation and to put the occupation in the first column, it will ignore all people who do not have an occupation.

In order to circumvent this problem, we can use a built-in function **`:form[]`**, which will, in effect, add an 'empty' column to the index. For this function to work in the manner we want it should contain an 'empty' character string.

Example 2.27

```
query name=burial;part=relp
index part=:form[""];
   part=:occupation;
   part=:surname
stop
```

Adding an empty string to the **`:form[]`** function, produces an extra column of no spaces, in front of the index. However, one space will be added, as κλειω always adds a space after each logical column it displays in an index.

Summary

This chapter introduced

- The specification command **`query`** and the analysis commands **`write`** and **`index`**
- Some of the ways in which a query may be modified using *conditions* by means of *logical operators* like **`and`**, **`or`** and **`not`**, by *comparison modifiers* like **`equal`**, **`start`** and **`limit`** and by the keyword **`null`**
- The functions **`:each[]`** and **`:total[]`** which both specify which particular elements related to a group should be displayed after an analysis command.
- The function **`:form[]`**

Exercises

The answers to these exercises and all subsequent ones will be found at the back of this volume.

Produce tasks which display the following information:

Exercise 2.1

Retrieve all information about the people who died having previously held the post of alderman.

Exercise 2.2

Retrieve all the information about the related people who held the post of alderman.

Exercise 2.3

Produce an alphabetical list which includes, in this order, the surname, first name and occupation of all the people who died.

Exercise 2.4

Modify this task to include all the people in the database who had a surname.

Exercise 2.5

Produce a similar list of all the women in the database who died (use the specification **`part=p:status="f"`**).

Further exercises

Rather than using the database burial for these further exercises, we shall use a database called baptism. This has already been constructed and will be found in the directory called tutorial. The two files baptism.mod and baptism.dat will have to be compiled before the database can be used. The database contains a list of baptisms from the Winchester Cathedral register of baptisms and contains all the entries from 1599 to 1630.

The database contains the groups: doc, person, and relp, with the following hierarchy:

```
doc
 |
 p
 |
relp
```

You should note that the hierarchy is identical to the database of burials.

The group p contains the following elements about those people who were baptised. The element bapdat contains the date of baptism.

id
status
firstname
surname
relation
bapdat
abode

The group relp contains the following elements about the people who were related to those people who were baptised:

status
title
firstname
surname
subtitle
occupation

Each of these elements has the same meaning as for the database burial (see Chapter 1).

Using this database, perform the following tasks:

Exercise 2.6

Produce an alphabetically sorted list of all the related people where each line of the list starts with the element surname and is followed by the first name and then the occupation.

Exercise 2.7

Produce an alphabetically sorted list of all the related people who had an occupation. Include the surname and the first name of all such people.

Exercise 2.8

Using the *burial* database produce an alphabetical list of all the occupations containing the character string "clerke". It is possible that you will get more information than you asked for. Try to work out why you have more information and guess how the correct information might be displayed.

2.5 Troubleshooting

κλειω's user-friendliness, as you may have spotted, leaves a little to be desired, but built into the system are a large number of particularly useful help messages. The method of accessing these messages is rather inconvenient as you have to issue a command which contains an error before you can receive a message.

For example, if you run this task (which does contain an error):

```
query name=burial;part=:occupation
index part=:surname;
  part=occupation
stop
```

you would get the following result:

```
query name=burial;part=:occupation
index part=:surname;
          part=occupation

***** Error: The following group identifier is unknown
***** Error: An analysis command requires an element specification

stop
```

This should be enough for you to solve the problem, but if you were still stuck you could add a command line to the task which would tell κλειω to issue a more comprehensive message. This command line is **`options explain=yes`** and the new task would look like this:

```
options explain=yes
query name=burial;part=:occupation
index part=:surname;
  part=:occupation
stop
```

If you run this task you will see that a further message for each of the error messages previously displayed will be shown on the screen which should make it much easier for you to correct the error in the task.

The **`note`** command

A further hint: it is possible to include comment in your tasks, by prefacing them with the **`note`** command. This tells κλειω to ignore everything that follows until the next command is encountered. For example:

```
note practice.1
note my first attempt to retrieve dentists
query name=burial;part=:occupation="dentist"
write part=:surname
stop
```

Although you may see little point in using this feature at the moment, once you start to work with complex databases it will save you a lot of rummaging and backtracking.

Chapter 3

Data Types

3.1 Introduction

The aim of this chapter is to describe the different types of data that κλειω is able to process. However, the chapter will not describe how to implement these data types. Basic implementation of two of them will be discussed in the following chapter while other implementations will be discussed later in this volume.

Every entry that is contained within a κλειω database must belong to one of the seven different data types that κλειω supports. If an entry is not defined as belonging to a particular data type, κλειω may not be able to process that entry in the manner in which the user would like it to be processed. This is not as cumbersome a task as it might seem, since by default κλειω processes all data as type **`text`**; so all data is understood to be of type **`text`** unless it is defined otherwise.

In conventional data processing, whenever one designs a database one defines the types of data that one is going to use. Conventionally these are alphanumeric, numeric, date and decimal (for currencies). Historical data is more complicated than the data normally used with modern commercial software, so the preparation of data for processing requires more care than usual. However, as κλειω has been designed specifically to allow the user to keep the source intact, the user must carefully define what sort of data is being used, thus creating rules applicable to processing.

κλειω supports seven different data types. These are:

```
text
date
number
category
relation
location
image
```

In this chapter we shall describe some of the salient points about each of these data types. Included with the description of each data type is a pointer to the relevant section of the Reference Manual. We do not expect you to follow these cross-references up when using the tutorial for the first time. They are included so that when you start using your own data you should be able to refer easily to the reference manual.

3.2 The `text` data type

`text` is the default data type. Examples of `text` are:

```
Richard

Smythe

Prebendary

I Thomas Stubbington, loader being sicke of bodie but of
perfect memorie I prayse the Lord my god for it, do constitute
make and ordayne this my last will and testament in manner and
forme folloinge. First I give and bequeath my soule into the
hands of Allmightie god my maker & Redemer hopinge onlye to be
saved by the precious Death and blodsheading of my Lord and
Saviour of my Lord & Saviour Jesus Christ onlye, and my bodie
to the earth from whence first toke ..... essence or being & to
be buried in the Churche yerde of the holie Trinitie.
```

(Hampshire Record Office A88/1 — will of Thomas Stubbington)

Data of the data type `text` must be made up of non-reserved characters. That limitation aside, data of this type can include any character that you are able to produce using EDIT. In addition, the configuration program can be used to get κλειω to accept other non-standard character sets like Cyrillic or Greek. The reserved characters can be redefined by this means as well. (Refer to the notes on the configuration program in Appendix B of the Reference Manual.) In some circumstances there is a limit to the number of characters one can have in any particular entry, but usually one can say that there is no limit to the number of characters that can exist in an element of data type **`text`**. For further information, see Section 7.3.1.2 of the Reference Manual.

3.3 The `date` data type

κλειω is exceptionally well equipped to deal with what can be one of the trickiest problems of historical material. All κλειω demands is that anything of the data type **`date`** should be able to be converted into a calendar date.

By default κλειω assumes four things about dates, but each of these things can be changed depending on the choice of the user. These four default assumptions are:

- date information must be in the format: "Day.Month.Year"
- each of the three parts of a date must be numerical
- the year starts on 1 January
- every date used must be between 1.1.1500 and 1.1.1994.

The last of these assumptions should be clarified first. This does not mean that κλειω can not cope with dates before 1 January 1500, but means that κλειω will not accept dates before that date unless it is specifically told to do to. This helps the program to spot problems when entering data.

Though κλειω assumes that all dates must be in the form "Day.Month.Year", it is perfectly possible to change this order for all the different types of calendar understood by κλειω (except the Roman calendar).

It is also perfectly possible for κλειω to understand months as a textual string. With some calendars abbreviations like "Nov" would be acceptable, and the month "Maie" could also be accepted if it had been defined as an alternative spelling for "May".

κλειω also allows one to change the date of the beginning of the year. Certain commands allow the user to define 1 March, 25 March, Easter Sunday, 25 September or 25 December as the beginning of a particular Christian calendar.

κλειω is also not restricted to Christian calendars. The full range of calendars supported by κλειω is listed below.

```
numbers
western
islam
revolution
byzantine
moses
latin
saints
```

The default calendar, described above, is **`numbers`**. If you do not specify an alternative calendar, dates are accepted in the following format, which is interpreted as a date in the Christian calendar after the birth of Christ.

```
16.5.1630
```

If one defines the calendar as **`western`**, one is allowed to enter the date in the following format:

```
16 September 1630
```

(where months are understood to be correctly spelled months in English).

It is also permissible to abbreviate months in the **`western`** style of dates as long as they represent at least the first three letters of the month:

```
16 Sep 1630
```

κλειω also supports dates from the Hijra or Hegira calendar. Using the definition **`islam`** dates can be entered in the following format.

```
12 Ramadan 956

6 Safar 1156
```

These months can not be abbreviated.

If one defines the calendar as **`revolution`**, dates from the French Revolutionary calendar (also known as the Republican calendar) can be processed. The following examples are valid.

```
5 Frimaire 3

10 Nivose 14
```

These months can be abbreviated to the first four characters.

If one defines the calendar as **`byzantine`**, dates can be expressed in the same format as the **`western`** calendar but they are interpreted as a date in the Byzantine era, with the year beginning on 1 September until 1 January 1700 when the year begins on 1 January.

If one defines the calendar as **`moses`**, dates from the Jewish calendar can be processed, as long as the date is in the following format:

```
5 Nisan 5301

13 Tam 5127
```

Months in the Jewish calendar can also be abbreviated to their first three characters.

If one defines the calendar as **`latin`**, dates from the Roman calendar can be processed by κλειω. These must be in the following format:

```
prid non Dec 1227

4 KAL JUN 1125
```

For how κλειω deals with special chronological problems concerning Latin dates refer to the Reference Manual, Sections 7.3.1.3.1.1.1. Note that this calendar interprets a date as a date in the Christian calendar after the birth of Christ. It is not designed to deal with dates which are B.C.

κλειω also allows the user to define dates using saints' days or church feast days.

As there are generally two types of church feast day, specific saints' days which fall on a particular day of a year and religious festivals which fall a number of days, weeks or months after Easter, κλειω allows the user to define these days in a separate file, which interacts with the database and converts all such dates to a date after the birth of Christ. As κλειω does not 'understand' these dates, one would not be restricted to using saints' days, but could use secular dates like Elizabeth I's birthday.

Users are able to define a feast day as either a particular day in a year, a date related to Easter Sunday (e.g. Pentecost is the seventh Sunday after Easter) or an expression related to another date (e.g. Advent is the fourth Sunday before Christmas).

More details about producing a file containing information about saints' days which κλειω interprets in association with a database can be found in Chapter 6.

Whichever calendar the user considers to be most appropriate to his/her use, there are a number of ancillary rules which apply to most dates.

First, the number zero can be used to describe an unknown part of a date. If one knows that an event happened on an unspecified day in May 1630 one can represent this as:

```
0.5.1630
```

Similarly if one knows that an event occurred on the first of an unspecified month in 1630 one can represent it like this:

```
1.0.1630
```

This refers only to the **`numbers`** form of date, but if any other form of dating system is being used (except **`latin`**) an unknown month can be defined as "unknown". For example:

```
12 unknown 1423
```

Two dates can also be joined together by a hyphen. If one knew that a person died between two dates, for instance 8 May 1630 and 12 May 1630, one could represent it as follows:

```
8.5.1630-12.5.1630
```

(The first of these dates is known as a *terminus post quem* and the latter a *terminus ante quem*).

Similarly if one knew that an event happened before a certain date, it could be represented as follows:

```
-12.5.1630
```

This feature can also be applied the other way round;

```
8.5.1630-
```

Within κλειω Christian calendars conforms to the usual rules (e.g. "Thirty days hath September..."). The abbreviations OS and NS can also be added to the end of a date to denote Old or New Style, i.e. Julian or Gregorian calendars.

```
8.5.1630 OS
15.10.1752 NS
```

κλειω is thus able to cope with a remarkable variety of styles of dates and is able to cope with a variety of them within the same database.

For further information, see Section 7.3.1.3 of the Reference Manual.

3.4 The **`number`** data type

By default κλειω only accepts numbers between zero and one million. However users can specify that they want to use numbers greater than one million and less than zero. There are some cases in historical data where it may be necessary to use negative numbers. For example, in an inventory of an individual's goods and chattels, debts owed by the individual may also be included which might be described as a negative sum owed to the deceased.

κλειω will only accept information of data type **`number`** if it is possible to convert it into a number.

Numbers can be represented not only by individual numbers but as expressions using the standard arithmetical operators '+', '-', '*' and ':' to represent the operations of addition, subtraction, multiplication and division. These expressions can also be nested within brackets. κλειω is untypical as it uses arithmetical operators in the order in which they occur rather than in the usual mathematical order; i.e. 2 + 3 * 4 = 20, not 14.

Within historical data there are many occasions on which numerical information is imprecise. κλειω allows individual numbers to be prefixed by one of the following uncertainty operators: **`equal`**, **`circa`**, **`greater`** or **`less`**. For example:

```
query name=dummy;part=:age="17" greater
```

These can be combined to make compounds such as **`equal circa`**, **`equal greater`**, **`equal less`**. Other combinations are even more complex, and will not be treated here; for an explanation, see Section 8.1.2.3.3 of the Reference Manual.

Numbers can also be defined as doubles or triples. It is very rare for information relating to currencies in historical data to be in decimal format, and the same is true of weights and measures. The examples below are from Carlo M. Cipolla, *Cristofano and the Plague* (Collins, London, 1973), Appendix 2, pp. 140–143.

The currency in early seventeenth-century Tuscany was based on both an accounting system and monetary units. For accounts the following system was used:

1 *ducato* (or *scudo*) = 7 *lire*
1 *lira* = 20 *soldi*
1 *soldo* = 12 *denari*

Thus the value 13 *soldi* 4 *denari* would be equal to a total of 160 *denari*.

Coinage at the time was more complicated. The coins that were most commonly used were:

		l	*s*	*d*	total *d*	total *l*
doppia	=	20	0	0	4800	20
mezza doppia	=	10	0	0	2400	10
piastra	=	7	0	0	1680	7
testone	=	2	0	0	480	2
lira	=	1	0	0	240	1
giulio	=	0	13	4	160	0.666
mezzo giulio	=	0	6	8	80	0.333
crazia	=	0	1	8	20	0.1
quattrino	=	0	0	4	4	0.02

With κλειω, the user is able to define any form of currency as long it refers back to a series of singles, doubles or triplets and as long as there is a base. In the above table it can be seen how either the *denaro* or the *lira* could be used as a base.

Other measures which are often more complicated than currency — such as weights — can be adapted in the same way.

Numbers can be written in the following ways.

`1`

`12li 14s 2d` (denoting 12 pounds 14 shillings and 2 pence or 12 *lire* 14 *soldi* and 2 *denari*).

`12.14` (denoting 12 pounds 14 shillings or 12 pounds and 14 ounces)

`12.75` (denoting 12¾ of something)

For further information, see Section 7.3.1.4 of the Reference Manual.

3.5 The `category` data type

Data of the **`category`** data type are often used in κλειω as 'value-added' data. In the Winchester Cathedral burial and baptism databases the element status is of data type **`category`**. Inherent within the source is that some people are dead, some are male and some are female. This information can be added to the database in order to make it more comprehensible. For instance the following example may represent a dead man:

```
dm
```

If a female witness to a will was to leave a mark rather than sign her name one might be led to the conclusion that that woman was illiterate. The following example may represent that woman:

```
fiw
```

where **`f`** stands for female, **`i`** for illiterate and **`w`** for witness.

Information of this data type is always made up of individual characters, each of which is analysed separately. The order in which these characters occur is unimportant. One could have specified the previous example as **`wif`** or **`ifw`** etc. The only qualification that must be made is that each letter or number can only refer to one category or value. The following example would be impossible:

```
wiw
```

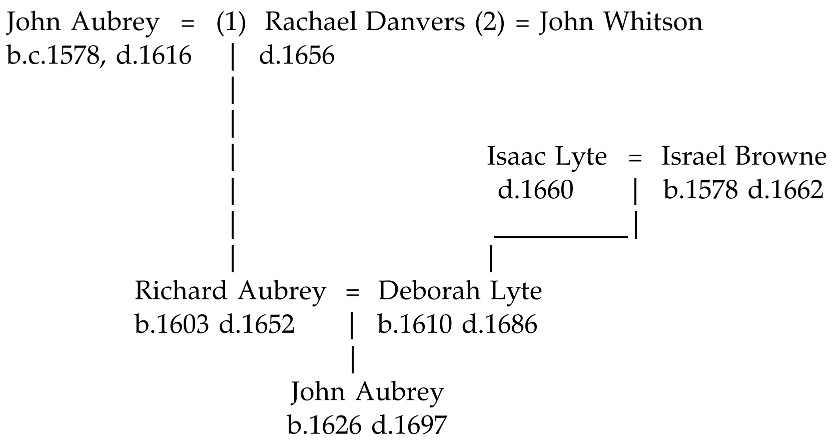
(where **`w`** stands for woman, **`i`** for illiterate and **`w`** for witness).

For further information, see Section 7.3.1.5 of the Reference Manual.

3.6 The `relation` data type

Data of **`relation`** data type will be fully discussed in Chapter 13. This data type is always user-defined. They are similar to unique identifiers found in relational data processing. In κλειω they can be used to link entries relating to one person.

Consider the following family tree:


John Aubrey = (1) Rachael Danvers (2) = John Whitson
b.c.1578, d.1616
d.1656
Isaac Lyte = Israel Browne
d.1660
b.1578 d.1662
Richard Aubrey = Deborah Lyte
b.1603 d.1652
b.1610 d.1686
John Aubrey
b.1626 d.1697

(Adapted from Anthony Powell, *John Aubrey and his Friends*, 2nd ed. (Mercury Books, London, 1963))

Theoretically all the information in this family tree could have been taken from parish registers. If we were to assume that they were all taken from the same parish register the information could look something like this:

Births	*Deaths*
1578 John Aubrey	1616 John Aubrey
1578 Israel Browne	1652 Richard Aubrey
1603 Richard Aubrey	1656 Rachael Whitson
1610 Deborah Lyte	1660 Isaac Lyte
1626 John Aubrey	1662 Israel Lyte
	1687 Deborah Aubrey
	1697 John Aubrey

In this case we know that Israel Brown married Isaac Lyte and took his name. Likewise we know that Rachael Whitson was born Rachael Danvers, married John Aubrey and gave birth to Richard Aubrey. Later, after the death of John Aubrey, she – presumably using the name Rachael Aubrey – married John Whitson, and took his name in the process. Thus unless one compared all three parish registers (if one was only using that source) it would be impossible to connect Rachael Whitson who died in 1656 with Rachael Aubrey the mother of Richard Aubrey.

κλειω is able to understand relationships like these by letting the user define relationships between individuals. The user must give an identical item of **`relation`** type data to the entry for each entry which related to the same person, in this case Rachael Danvers/ Aubrey/Whitson.

κλειω is able to cope with this by using a relational network. When a κλειω database is created, all of the parts of the **`relation`** data type which are known as *network identifiers* are found, and at the point where each one is found κλειω makes a note of where in the database another occurrence of the identical network identifier can be found. The result is a *subnetwork*.

For further information, see Section 7.3.1.6 of the Reference Manual.

3.7 The `location` data type

Data of **`location`** data type will be covered in detail in Chapter 14. It is only necessary to mention here that κλειω is able to produce maps. These maps are made using topographical data provided by the user.

κλειω is able to produce maps based on topographical data provided by the user. These maps can be produced based on other data within a database as well. In Chapter 14 we will be demonstrating how to construct a map with κλειω using material from the English 1881 Census.

This profile of Göttingen's tax declarations in 1749 was produced by κλειω.

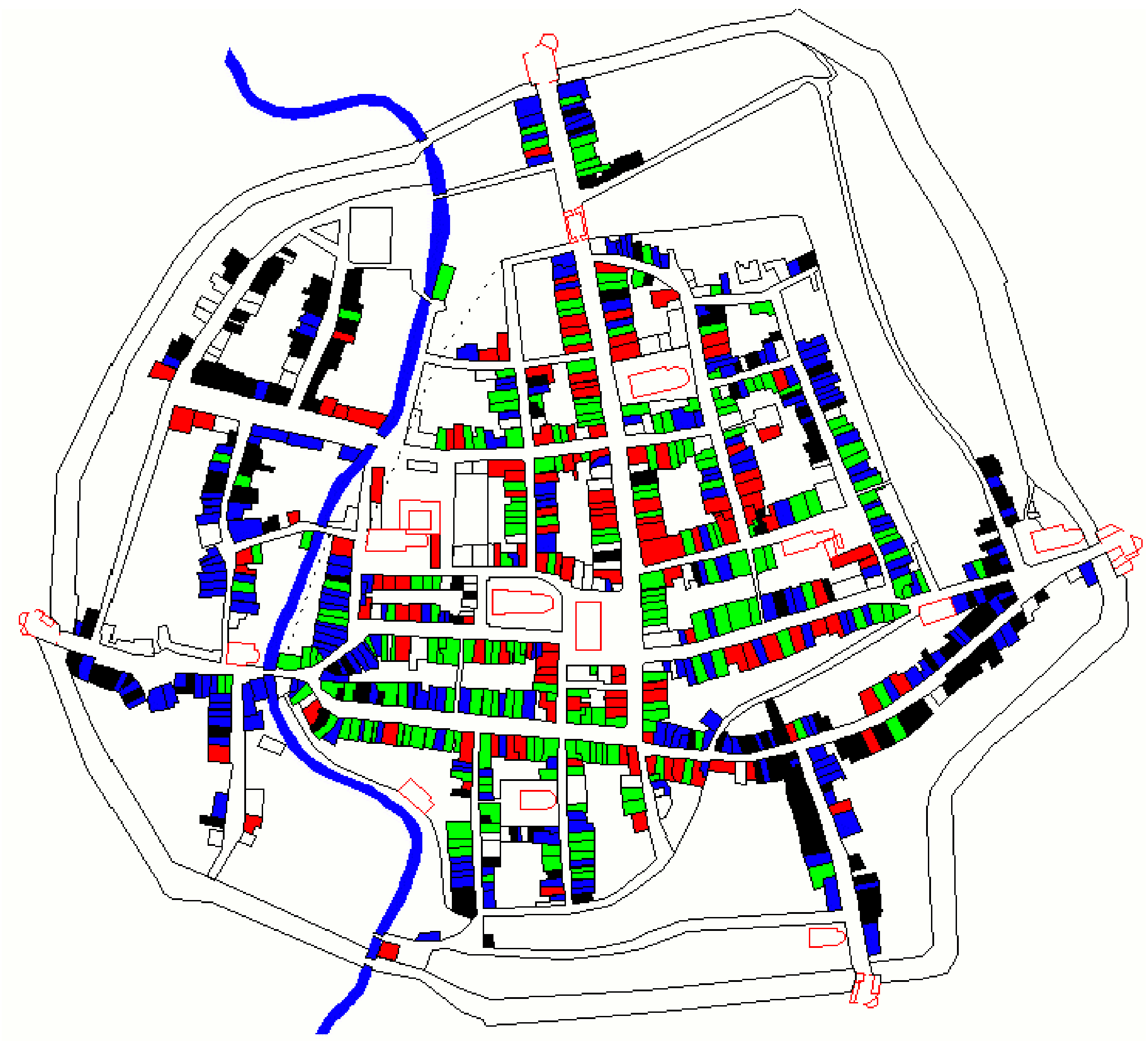


Figure 3.1

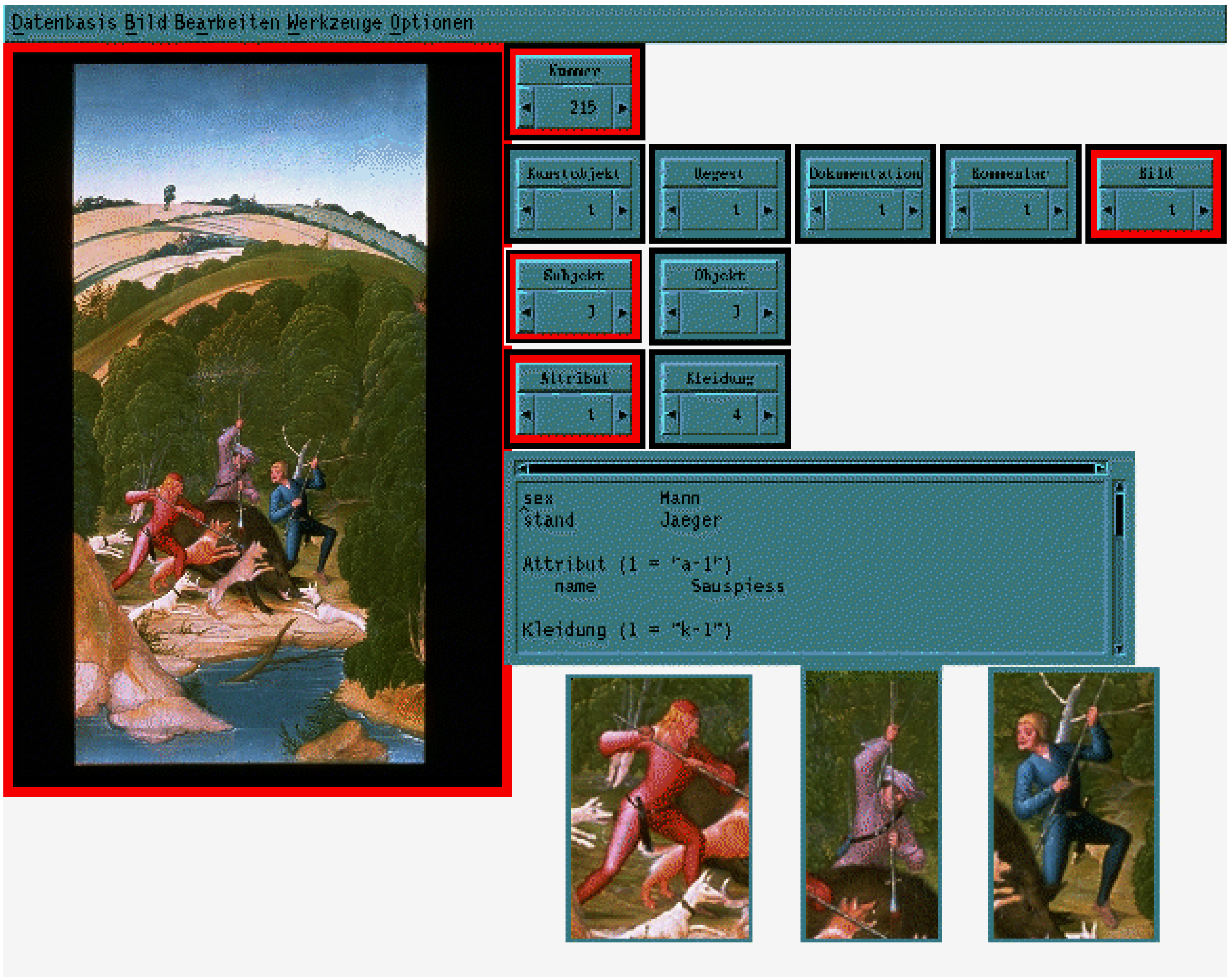
3.8 The `image` data type

When run on a suitably powerful machine, κλειω is able to process images. This feature will not be considered further in the book as it is a separate part of the software. For further details on image processing see:

J. Fikfak & G. Jaritz (eds.), *Image Processing in History: towards Open Systems*. Halbgraue Reihe zur historischen Fachinformatik, A16 (St. Katharinen, 1993).

G. Jaritz, *Images. A Primer of Computer-Supported Analysis with* κλειω *IAS*. Halbgraue Reihe zur historischen Fachinformatik, A22 (St. Katharinen, 1993).

M. Thaller (ed.), *Images and Manuscripts in Historical Computing*. Halbgraue Reihe zur historischen Fachinformatik, A14 (St. Katharinen, 1992).



```
sex            Mann
stand          Jaeger

Attribut (1 = "a-1")
   name            Sauspiess

Kleidung (1 = "k-1")
```

Figure 3.2. A screen from the image analysis system

PART II

κλειω BASICS

This section introduces the most frequently used features of κλειω. All the concepts described in this part are likely to be used on a regular basis. These include methods of creating a database, more complicated query facilities, the integration of knowledge within databases, the processing of textual material, the creation of 'look-up tables' for the coding and classification of data, and also some features to produce more sophisticated output.

Chapter 4

Creating a database

4.1 Introduction

κλειω is a database management system in the broadest sense of the term. It stores an historical source which has been structured according to certain rules, while the source is kept as close to the original as possible. This makes it accessible for both hermeneutical interpretation and formal investigation. We should reiterate here that historical computing is not just about formal quantitative analysis but also about interpretative and qualitative analysis. κλειω allows one to do both. One can query a source in a state that resembles the original in order to give quantitative results, or one can use κλειω to sift through data in the same way that historians do when consulting archives in a record office. Obviously the problem here is getting the source material into a format that κλειω can understand so that it can be processed.

We have stated that κλειω manages databases, but we have not yet explained exactly how it does so. This not the place to describe the process formally; this chapter will be concerned with constructing a database and getting source material into a κλειω database. You will already be aware, perhaps without knowing it, of a couple of vitally important principles of κλειω database design. These have been touched on in the previous chapters, but have not yet explicitly been described.

To complicate matters further, κλειω is only able to process databases that have been specifically designed for use with κλειω. (At least one history specific software package, CensSys, does convert data into a format that κλειω can understand, but at the moment it can only convert it into a format that the German version of κλειω can understand. This should change shortly.) However, as κλειω understands slash-delimited data in ASCII files, it is possible to export data from most proprietary database management systems, and of course κλειω is able, with a minimal amount of effort, to export data for use with other packages.

The usual method of creating a database for use with κλειω is to produce an ASCII file of the data, composed according to rules formulated by the user, and a file containing a definition of these rules so that κλειω can process the data. As we have seen in Chapter 1, these two parts of the database are usually (on DOS machines) given the file extensions .mod and .dat. (These are not rules but a fairly long-standing convention.) Both of these files need to be created and executed (in the correct order) in order for κλειω to produce a working database.

The first stage in constructing a database containing historical material, whether for a conventional system or for κλειω, is to have some understanding of the source. For potentially very large databases it may be impractical to have an intimate knowledge of all the material, but it is necessary to have some notion of the source. When constructing almost any historical database with an application other than κλειω, one needs to have a clear idea of some of the questions that the database will be asked. This is necessary because the format in which data is entered into a database may affect the questions that can possibly be asked of it. With κλειω this is not the case. In fact it may positively hinder the proper design of a κλειω database, as one would be making the method fit the question rather than the other way round. κλειω has been specifically designed to allow the user to put the data into the database in such a way that it should not have to be changed significantly from the original material. Who knows what one may discover if one spends some time with the source?

There are two different approaches to constructing a database with κλειω. The first is to begin with the source and to elaborate the input structure, proceeding to the definition of that structure. The second is to define the structure first and then to enter the data. The two approaches achieve the same result. The decision as to which approach to adopt will depend at least partly on one's knowledge of the sources that are are going to be used to construct the database. In practice most people use a combination of the two.

Within a κλειω database the data is 'transcribed' from the original source into a format that κλειω will be able to understand, i.e. in accordance with a collection of κλειω conventions. This process is very different from a traditional 'fixed-field' database. The rules used in the manipulation of the data into a format that κλειω can understand must also be described for the benefit of the system in a fashion similar to a traditional data dictionary.

To recapitulate, κλειω needs two things to make a database:

- data, collected according to rules composed by the user for that source, based on existing κλειω conventions

- a description of those rules for κλειω to understand the data.

The only difference in the two methods of creating a database for κλειω is the order in which these two files are constructed. Either is acceptable, and databases created for use with κλειω are usually made by a compromise of these two techniques.

The first of these processes ensures that the structure of a database is considered with the source in mind. Practically, this means that if one has a basic knowledge of any source, one can sit down and start entering the data into a data file (as long as one has a basic knowledge of the rules of κλειω); once a number of records have been entered, the rules pertaining to that data can be written into another file. These rules normally consist of two parts, first the structure of the database (known as the *structure declaration*), and then any additional rules pertaining to the data itself (known as *logical objects*). Once both the data and the rules for the database have been completed, the file containing the rules must be put into the system to create the κλειω equivalent of an empty table. The data can then also be put into the system. If any errors are discovered here either the data file or the rules file can be corrected and re-compiled to create a working database. This process is usually hit-and-miss and lacks the intellectual rigour of the other method, but it will always eventually end up with a database ready for querying.

The second process can be compared with the more advanced techniques of Entity Relationship Modelling or Relational Data Analysis, though the theory involved in creating a structure for a κλειω database is not so involved. However, like ERM or RDA it involves a complete understanding of the source(s) being processed. This route of creating a database is therefore only recommended if one does have such complete understanding of the source, or if the source is simple and heavily structured, like a census record or a parish register.

As we said earlier, the process normally used to make a database in κλειω is essentially a compromise between these two techniques. The typical κλειω user starts by typing a sample of the data into a database and making decisions about the structure on the hoof, and then creates the structure file, including information contained in the source but not yet encountered. We recommend that you follow the example below, which gives a detailed description of the creation of a database using the first method.

We should mention here that this is a long chapter, which needs to be fully understood. We strongly suggest that you work all the way through to the end of Section 4.6, following all of the instructions.

Before moving directly on to the creation of database, there are a few essentials to discuss.

4.2 Preliminaries

Many of these points reiterate and enlarge on what has already been said in Chapter 1. Nonetheless they should be repeated here.

4.2.1 Files

As stated above, a κλειω database is made up of at least two parts, the data and the rules. These two parts are commonly held in two different files. The data file usually has the extension .dat and the structure and rules file the extension .mod. It is not necessary for these two parts to be kept in separate files but this is usually done for convenience.

Both of these files must be in an ASCII format, which must be composed of characters in the extended ASCII set as defined by IBM. However, modified national character sets can also be used if the configuration program is used. Refer to Appendix B of the Reference Manual.

Rendering a database usable is similar in process to querying a database. To get κλειω to interpret either a .mod or a .dat file, one must type:

```
kleio <input file>
```

and if one wanted to put the result of that into another file one could type:

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

When creating a database, one must first compile the .mod file and then the .dat file. It is essential to compile them in that order; otherwise, data would be fired either into a non-existent database, causing an error, or into an existing database (if one already exists) which would subsequently be destroyed by the compilation of the .mod file on top of it.

This last process is performed by typing (at the DOS prompt):

```
kleio filename.mod
```

and when that has been done

```
kleio filename.dat
```

If there are no mistakes the database will be able to answer queries.

It is wise to send the output from these two commands into an output file in order to check that they have been performed correctly. This can be achieved by adding the name of a result file to the end of the command, as the following example shows.

```
kleio filename.dat result
```

4.2.2 Commands, directives and parameters

As shown in Chapter 2, certain rules apply to all the 'commands' used in κλειω. These can be summed up as follows:

- Commands must start in the first column of a line. If a command line is longer than the width of a screen it is perfectly acceptable to continue the command over as many lines as is necessary as long as the second and subsequent lines do not start in the first column. (If they do, κλειω will attempt to understand that line as another command, and will probably fail and produce an error message.)

- Directives must be followed by at least one space.

- Parameters must be followed by a parameter value.

- Where two parameters appear in sequence the first must be followed by a semi-colon.

At a guess, around 70% of all error messages encountered while using κλειω relate to one of those few rules.

4.3 The sample database

4.3.1 Description

The sample database uses material from the 1881 Census of England and Wales. This source has been chosen primarily because the source is heavily structured, but also because of the similarity of census material from different countries. The material used is not even 'real' historical data; it was made up by the General Registry Office to aid and assist the enumerators in completing their enumeration books. Thus there should be relatively few problems inherent within the data. There will also be some mention of 'real' English Census data and the problems that might crop up while converting the source into a form that κλειω can understand.

Figure 1 is a copy of the source from which a database is to be constructed. Figure 2 is a transcription of that original.

4.3.2 The **`read`** command

Open a new file called censsamp.dat, and go into it, using the editor. The first line to be typed involves a new command, **`read`**. The **`read`** command is placed above all of the data in the data file, and must be followed by a **`name=`** parameter. The **`name=`** parameter is given a user-defined name as a parameter value to define the name of the database. This is an essential part of the data file; without it κλειω would not know into which database to read the data that follows it. Let us call this database censsamp. Names of databases can be of any length, and must be made up of alphanumerical characters only. Internally, however, only the first twelve characters are used and under DOS only the first eight.

```
read name=censsamp
```

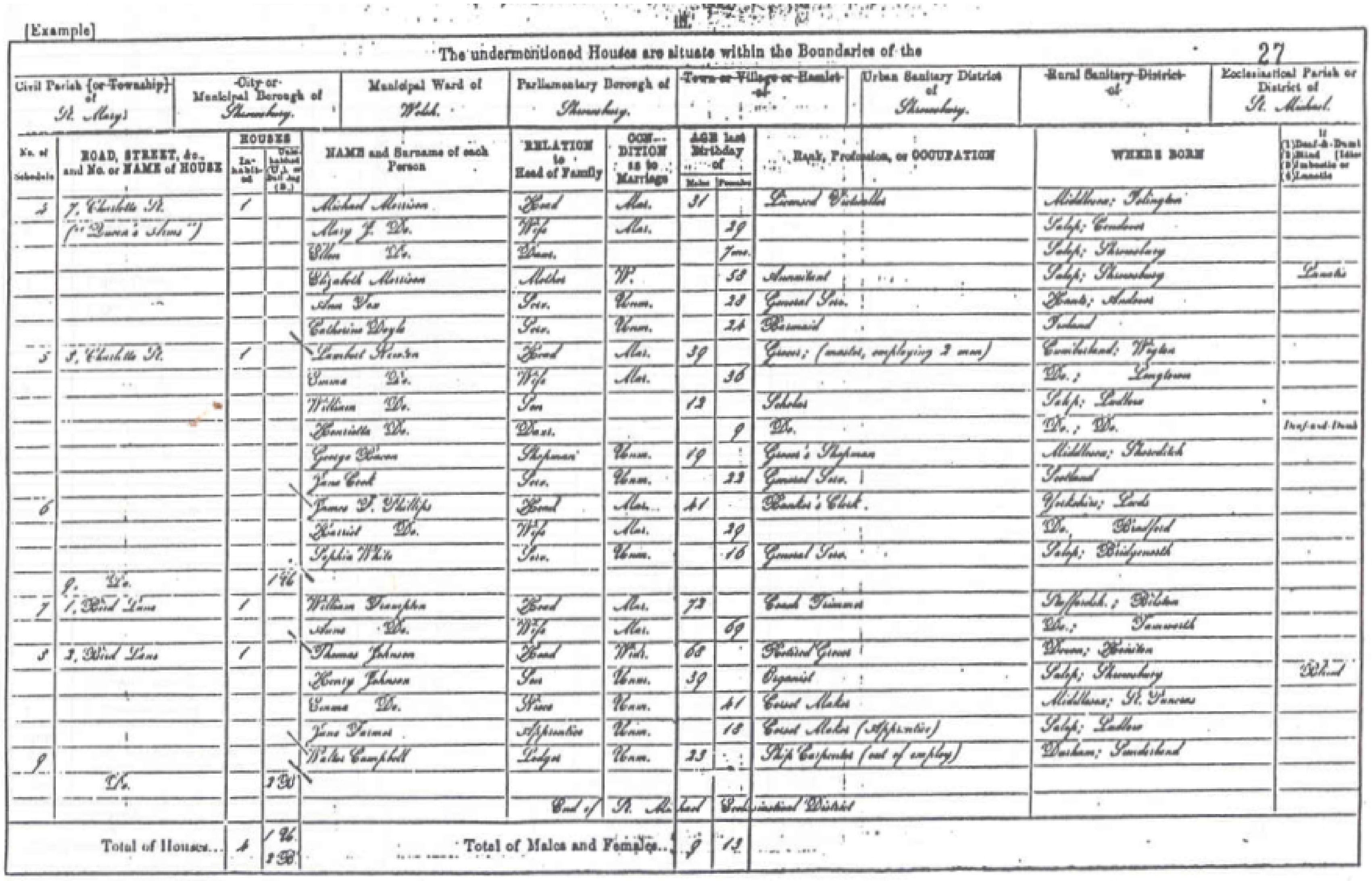
We shall now start to create the database.

4.3.3 The hierarchy of a database

As we said above, this is a description of how to construct a database using the first method, i.e. taking the source as the starting-point. However, it is still necessary to keep in mind certain rules that will pertain to the structure of the database.

The *hierarchy* of the database is probably the most important issue to keep in mind when beginning the construction of a database. The term hierarchy as used with database design implies that something, usually a record or series of records, must be associated with the highest level of a hierarchy. In κλειω the first step in creating a database is to decide, for any source or collection of sources, what is going to be the highest level of the hierarchy. This level of a κλειω database is known as the *document* level.

It is perfectly possible for a κλειω database to have only a document and no further levels of hierarchy.



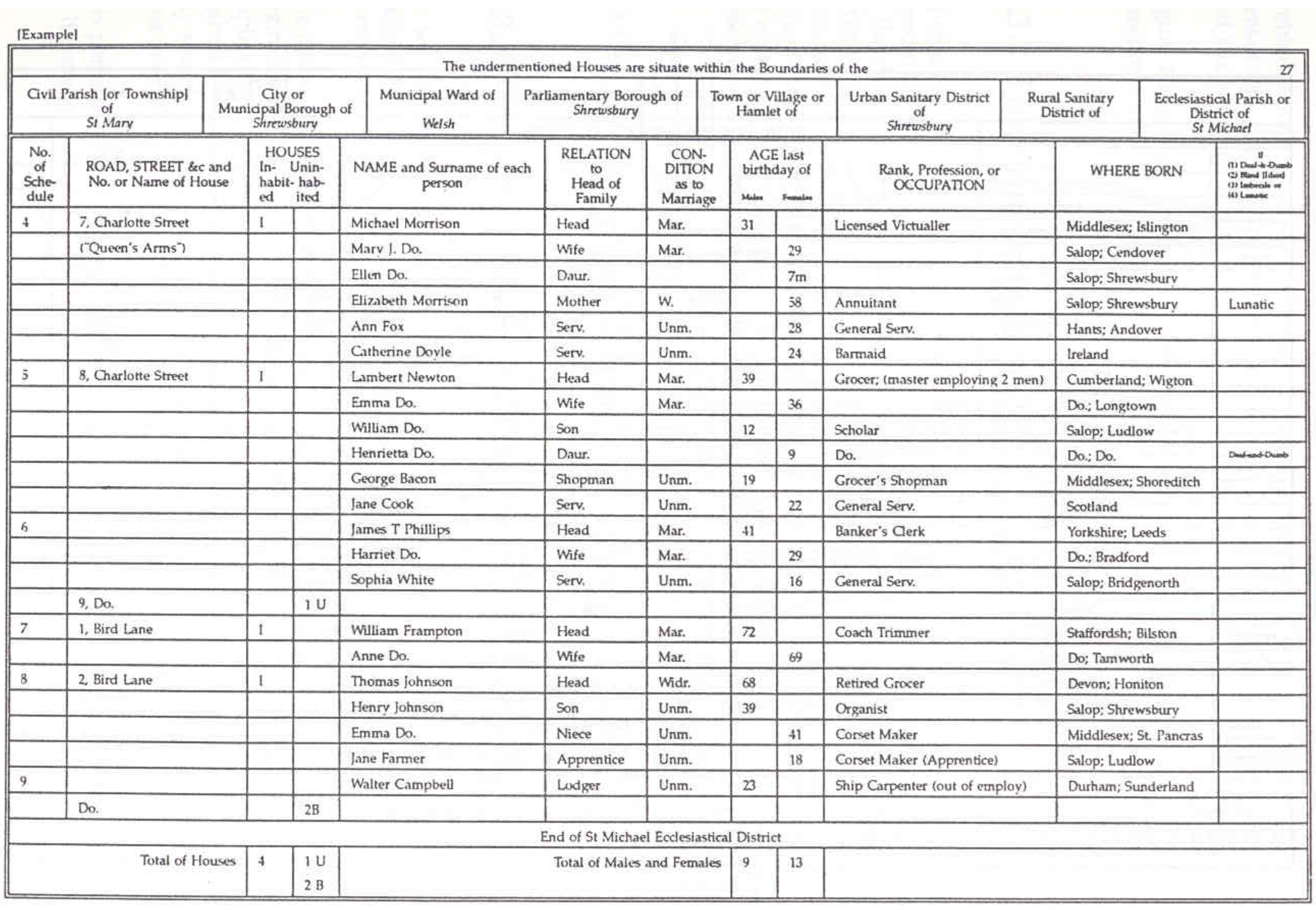
[Example]

The undermentioned Houses are situate within the Boundaries of the 27

Civil Parish [~~or Township~~] of	~~City or~~ Municipal Borough of	Municipal Ward of	Parliamentary Borough of	~~Town or Village or Hamlet of~~	Urban Sanitary District of	~~Rural Sanitary District of~~	Ecclesiastical Parish or District of
St. Mary	Shrewsbury	Welsh	Shrewsbury		Shrewsbury		St. Michael

No. of Schedule	ROAD, STREET, &c., and No. or NAME of HOUSE	HOUSES Inhabited	HOUSES Uninhabited (U.), or Building (B.)	NAME and Surname of each Person	RELATION to Head of Family	CONDITION as to Marriage	AGE last Birthday of Males	AGE last Birthday of Females	Rank, Profession, or OCCUPATION	WHERE BORN	If (1) Deaf-and-Dumb (2) Blind (3) Imbecile or Idiot (4) Lunatic
4	7, Charlotte St.	1		Michael Morrison	Head	Mar.	31		Licensed Victualler	Middlesex; Islington	
	("Queen's Arms")			Mary J. Do.	Wife	Mar.		29		Salop; Condover	
				Ellen Do.	Daur.			June.		Salop; Shrewsbury	
				Elizabeth Morrison	Mother	W.		58	Annuitant	Salop; Shrewsbury	Lunatic
				Ann Fox	Serv.	Unm.		28	General Serv.	Hants; Andover	
				Catherine Doyle	Serv.	Unm.		24	Barmaid	Ireland	
5	8, Charlotte St.	1		Lambert Newton	Head	Mar.	39		Grocer; (master, employing 2 men)	Cumberland; Wigton	
				Emma Do.	Wife	Mar.		36		Do.; Longtown	
				William Do.	Son		12		Scholar	Salop; Ludlow	
				Henrietta Do.	Daur.			9	Do.	Do.; Do.	Deaf-and-Dumb
				George Bacon	Shopman	Unm.	19		Grocer's Shopman	Middlesex; Shoreditch	
				Jane Cook	Serv.	Unm.		22	General Serv.	Scotland	
6				James T. Phillips	Head	Mar.	41		Banker's Clerk	Yorkshire; Leeds	
				Harriet Do.	Wife	Mar.		39		Do. Bradford	
				Sophia White	Serv.	Unm.		16	General Serv.	Salop; Bridgnorth	
	9, Do.		1 U								
7	1, Bird Lane	1		William Troughton	Head	Mar.	73		Coach Trimmer	Staffordsh.; Bilston	
				Anne Do.	Wife	Mar.		69		Do.; Tamworth	
8	2, Bird Lane	1		Thomas Johnson	Head	Widr.	68		Retired Grocer	Devon; Honiton	
				Henry Johnson	Son	Unm.	39		Organist	Salop; Shrewsbury	Blind
				Emma Do.	Niece	Unm.		41	Corset Maker	Middlesex; St. Pancras	
				Jane Farmer	Apprentice	Unm.		18	Corset Maker (Apprentice)	Salop; Ludlow	
9				Walter Campbell	Lodger	Unm.	23		Ship Carpenter (out of employ)	Durham; Sunderland	
	Do.		1 B								
					End of	St. Michael			Ecclesiastical District		
	Total of Houses...	4	1 U, 1 B	Total of Males and Females...			9	12			

Figure 4.1



[Example]

The undermentioned Houses are situate within the Boundaries of the 27

Civil Parish [or Township] of *St Mary*	City or Municipal Borough of *Shrewsbury*	Municipal Ward of *Welsh*	Parliamentary Borough of *Shrewsbury*	Town or Village or Hamlet of	Urban Sanitary District of *Shrewsbury*	Rural Sanitary District of	Ecclesiastical Parish or District of *St Michael*

No. of Schedule	ROAD, STREET &c and No. or Name of House	HOUSES In-habited	HOUSES Uninhabited	NAME and Surname of each person	RELATION to Head of Family	CONDITION as to Marriage	AGE last birthday of Males	AGE last birthday of Females	Rank, Profession, or OCCUPATION	WHERE BORN	If (1) Deaf-&-Dumb (2) Blind [Idiot] (3) Imbecile or (4) Lunatic
4	7, Charlotte Street	1		Michael Morrison	Head	Mar.	31		Licensed Victualler	Middlesex; Islington	
	("Queen's Arms")			Mary J. Do.	Wife	Mar.		29		Salop; Cendover	
				Ellen Do.	Daur.			7m		Salop; Shrewsbury	
				Elizabeth Morrison	Mother	W.		58	Annuitant	Salop; Shrewsbury	Lunatic
				Ann Fox	Serv.	Unm.		28	General Serv.	Hants; Andover	
				Catherine Doyle	Serv.	Unm.		24	Barmaid	Ireland	
5	8, Charlotte Street	1		Lambert Newton	Head	Mar.	39		Grocer; (master employing 2 men)	Cumberland; Wigton	
				Emma Do.	Wife	Mar.		36		Do.; Longtown	
				William Do.	Son		12		Scholar	Salop; Ludlow	
				Henrietta Do.	Daur.			9	Do.	Do.; Do.	Deaf-and-Dumb
				George Bacon	Shopman	Unm.	19		Grocer's Shopman	Middlesex; Shoreditch	
				Jane Cook	Serv.	Unm.		22	General Serv.	Scotland	
6				James T Phillips	Head	Mar.	41		Banker's Clerk	Yorkshire; Leeds	
				Harriet Do.	Wife	Mar.		29		Do.; Bradford	
				Sophia White	Serv.	Unm.		16	General Serv.	Salop; Bridgenorth	
	9, Do.		1 U								
7	1, Bird Lane	1		William Frampton	Head	Mar.	72		Coach Trimmer	Staffordsh; Bilston	
				Anne Do.	Wife	Mar.		69		Do; Tamworth	
8	2, Bird Lane	1		Thomas Johnson	Head	Widr.	68		Retired Grocer	Devon; Honiton	
				Henry Johnson	Son	Unm.	39		Organist	Salop; Shrewsbury	
				Emma Do.	Niece	Unm.		41	Corset Maker	Middlesex; St. Pancras	
				Jane Farmer	Apprentice	Unm.		18	Corset Maker (Apprentice)	Salop; Ludlow	
9				Walter Campbell	Lodger	Unm.	23		Ship Carpenter (out of employ)	Durham; Sunderland	
	Do.		2B								
				End of St Michael Ecclesiastical District							
	Total of Houses	4	1 U 2 B		Total of Males and Females		9	13			

Figure 4. 2

4.3.4 The document

The document in a κλειω database is considered to be the highest level in a hierarchy. Every database must have a document and it must be defined as such in the structure declaration (see Section 4.4). This is because κλειω assumes that it is going to process data with a similar structure (even if the structure consists of just a reference and its related text). Thus the document must be a group which holds a piece of data upon which all relevant data in the source depends. In this example the highest level could be the Civil Parish, the Municipal Borough, the page number, the folio number (upper right), or even the Public Record Office reference found at the bottom of the page. In this case we have chosen to make the PRO reference number the highest level. This has a number of advantages, not least being that individual documents are easy to access.

The document name

There are a number of rules which pertain to a document name. Firstly it can be as long as one likes, though κλειω only uses the first twelve characters for processing. The name must not contain any of the reserved data signals (see the Appendix to this chapter). There can, however, be any number of spaces between the name of the document (or group) and the **$** sign. It must begin with a character and only contain letters and numbers. Like all groups used in a data file it must be followed immediately by data signal 1 ("$"). If the document for this database were to be called **reference** the first line of the database could be written:

```
reference$RG11-5490
```

Notice that the name of the group is separated from the entry in the (as yet unnamed) element by a dollar sign, and that there are no spaces on either side of the dollar sign. (Notice a slight change from the original source in this line. As κλειω treats the forward slash ("/") as an element delimiter, it would not be appropriate to keep it here. There are rules which would allow us to keep the forward slash from the original but it would be unhelpful to describe them here.) Refer to the Reference Manual Section 4.1.1. for further information.

You should now type this in below the **read** command line. Please note that the **read** command *must* be followed on the very next line by a line containing the name of a document; otherwise κλειω will go through all the data until it finds one, and only then start adding data from the data file to the database.

4.3.5 Other groups

Once we have decided on the first group, the next step is to consider what other groups are to be used and to determine how they should fit into the hierarchy. Groups are usually entities or concepts that are immediately recognisable in a source. In this census data each individual could be a separate group, containing all the geographical information in the heading of this source. This would be poor style and necessitate a great deal of redundant data. The next type of information within this source which could be considered for definition as a group is the full address as given on the top of the page. None of the items that relate to the address can be considered to be dependent on one another (they look as though they might but careful examination of a considerable number of census records will suggest otherwise).

Thus the address could be added in a line like this:

```
address$St Mary/Shrewsbury/Welsh/Shrewsbury/Shrewsbury
  /St Michael
```

where St Mary refers to the parish, Shrewsbury is the town, Welsh is the ward, Shrewsbury is also the parliamentary borough and the sanitary district, and St Michael is the ecclesiastical parish. Notice that each element is separated by a forward slash ("/").

This line could be added to your file now. Note that for the purposes of display here we have used a second, 'continuation' line. There is no need for you to do this in the data file you are creating, but now at least you know how to! For longer items it is recommended that this is done for readability.

As each schedule number refers to a separate household, the information contained in the schedule field of the example could be one of the elements included in a household group. In this case, however, as different schedule numbers sometimes refer to the same address (see schedule numbers 5 & 6) it would seem wise for identification purposes to include it in a separate subordinate group. Even so, this will cause a problem later.

After the group **house** comes the group **schedule** which refers to the column headed 'road, street &c'. The following two lines could now be added below the last.

```
house$7 Charlotte Street/Queen's Arms
schedule$4
```

The remaining information relates to the head of households and their 'family'. It would be useful to make the heads of household a separate group, leaving the remainder to be in groups subordinate to them. Knowledge of this source tells us that lodgers, who live in the same house as another family, are not described as heads of household; their relation to the head of household is given as 'lodger'. Lodgers are given a separate schedule number, but not defined as a separate household. For this database a decision has been made to make lodgers heads of their own household so that they can more conveniently be analysed separately, while retaining the information that they are lodgers so that analysis can be performed on how many lodgers lived in each house.

At this stage, we recommend that you open a new file called censsamp.dat and type in the material already discussed. This is not a pointless task in data entry. It will give you a feel for entering data into a κλειω data file and should allow us to demonstrate some of the problems you may encounter when creating a κλειω database.

4.3.6 Elements

When we have decided on all the groups, the next stage is to decide on the different elements that each group should contain. The simplest way to do this is to start entering some of the data with the groups already decided upon. The next few lines of the database could look something like this *(do not type this)*:

```
head$Michael/Morrison/Head/Mar/31/Licensed Victualler
   /Islington/Middlesex
relp$Mary/J/Morrison/Wife/married/29/Cendover/Salop
relp$Ellen/Morrison/Daur/7months/Shrewsbury/Salop
relp$Elizabeth/Morrison/Mother/W/58/Annuitant/Shrewsbury/Salop
   /Lunatic
relp$Ann/Fox/Serv/Unm/28/General Serv/Andover/Hants
relp$Catherine/Doyle/Serv/Unm/24/Barmaid/Ireland
```

Before considering the problems within this example, it would be worth mentioning that during this process almost all of the elements that will be used have been decided upon. These elements also need to have names defined for them. The rules that apply to element names are similar to those that apply to names of groups. It makes sense to keep them short and memorable, as element names are used frequently when querying a database. This list shows the names of the groups along with the name of the elements they contain.

```
reference
   refnum

address
    parish/town/ward/parlb/sandis/ecclp

house
   address/name

schedule
   schednum

head
   firstname/surname/relation/cond/age/occupation/birthto/
      birthco
```

Now there are problems. If the data referring to related people were to be columnized it would look something like this:

Mary	J	Morrison	Wife	Married	29	Cendover	Salop	
Ellen	Morrison	Daur	7months	Shrewsbury	Salop			
Elizabeth	Morrison	Mother	W	58	Annuitant	Shrewsbury	Salop	Lunatic
Ann	Fox	Serv	Unm	24	General Serv	Andover	Hants	
Catherine	Doyle	Serv	Unm	24	Barmaid	Ireland		

Only the first, eighth and ninth columns actually give the right data. So in order for this material to be understood it must be written in the following format. Notice that empty 'columns' must be inserted.

```
relp$Mary/J/Morrison/Wife/married/29//Cendover/Salop
relp$Ellen//Morrison/Daur//7months//Shrewsbury/Salop
relp$Elizabeth//Morrison/Mother/W/58/Annuitant/Shrewsbury
   /Salop/Lunatic
relp$Ann//Fox/Serv/Unm/28/General Serv/Andover/Hants
relp$Catherine//Doyle/Serv/Unm/24/Barmaid////Ireland
```

where the names of the elements are:

```
firstname/fname2/surname/relation/age/birthco/birthto/
   affliction/country
```

Even though the place of birth, Ireland, in the original data is in the same column as the county and town of birth, it has been given a separate logical column.

There are further refinements that one can make to this data before considering whether this data file is in a format that κλειω will interpret correctly. This can be done without changing the nature of the data, but there would be some change in the order in which it appears.

In the enumerator's book there are two columns entitled 'houses', which the enumerator has filled in with an "I" or a "U" to denote whether the house is inhabited or not. This piece of information could be added to the house group as a item of data of `category` data type (see Section 3.5). One could abbreviate the items inhabited and uninhabited to **i** and **u** respectively. So the revised data might look like this:

```
house$i/7 Charlotte St/Queen's Arms
```

We will call this extra element status, and will use it for a number of abbreviated items of information.

In order to save time during data entry, and without taking anything away from the source, it would be possible to do something similar for the 'condition' column in the original source. People are either married, unmarried or widowed. These could be represented by the letters **m**, **u** and **w**). However, the abbreviation **u** for uninhabited has already been used, so arbitrarily the letter **s** has been chosen to represent unmarried people. (For reasons that will become obvious in a moment the letter **m** has not been chosen to represent those people that are married; instead we have used the letter **z**.)

A further set of abbreviations can be defined which relate to information about 'afflictions'. We shall define **d** as representing 'deaf and dumb', **b** as 'blind', **e** as 'imbecile' (since we have already used **i** for 'inhabited') and **l** for 'lunatic'. One other abbreviation that we will make is **o** for 'lodger' (again, because **l** has just been used).

The remaining piece of information that has been left out of the database is only just visible. The sex of an individual is denoted in the original by the two different columns representing age; one for men, the other for women. These entries will be replaced by the abbreviations **m** and **f**.

The first household now looks like this:

```
head$mz/Michael//Morrison/31/Licensed Victualler
   /Islington/Middlesex
relp$fz/Mary/J/Morrison/Wife/29//Cendover/Salop
relp$f/Ellen//Morrison/Daur/7m//Shrewsbury/Salop
relp$fwl/Elizabeth//Morrison/Mother/58/Annuitant/Shrewsbury
   /Salop
relp$fs/Ann//Fox/Serv/28/General Serv/Andover/Hants
relp$fs/Catherine//Doyle/Serv/24/Barmaid///Ireland
```

Notice two things:

- The information regarding the relation to the head of household has been omitted from the group **head** as it can always be inferred from the name of the group.

- It would have been perfectly possible to remove the relation to head of household completely from the group **relp** and include it as **category** type data; but as there are so many possible different relations, in the interests of clarity it was thought wise not to include them. There are already enough 'abbreviations' whose meaning is not immediately apparent by their letter. The following list of potential relations make it clear why this information has not been turned into a **category** date type: aunt; brother; brother-in-law; daughter-in-law; daughter; father; father-in-law; granddaughter; grandfather; grandmother; grandson; housekeeper; lodger; mother; mother-in-law; nephew; niece; servant; sister; sister-in-law; shopman; uncle; visitor. (There are doubtless others.)

Now all the groups and all the elements have been defined. You should now enter the first household into your data file.

FILLING UP THE CENSUS PAPER.

Wife of his Bosom. "Upon my word, Mr. Peewitt! Is this the Way you Fill up your Census? So you call Yourself the 'Head of the Family'—do you—and me a 'Female?'"

4.3.7 More on elements: aspects

Before we look at the two components of the 'model' file, a few further points need to be made about elements.

Entries

Elements can contain more than one entry. Each entry must be separated by a semi-colon (data signal 8). The sample data does not contain any such cases, but an example might be:

```
    farmer;innkeeper
```

which represents an individual who is both a farmer and an innkeeper.

Comments and original material

There may also be data within the source that you do not intend to process analytically but which you want to retain for future reference; and there may be data that is not in the source but which you as creator of the database would like to add. κλειω provides a facility for such additional material in the form of the concept of *aspects*. An entry can have up to two aspects other than the basic one, designated ‘comment’, and ‘original’. They are indicated in the data file by data signals, as follows:

Data signal 4, the hash or number sign ("#") can appear anywhere in an element to denote that what follows it is a comment to the main text. The following examples are correct uses of the data signal 4:

```
Coachman#Domestic servant added by checker
Barrister#rest of entry illegible
#illegible
```

Data signal 5, the percent sign ("%"), can also appear anywhere within an element. It normally denotes the original wording of the source, if the user has decided not to use the original wording or even if there is additional wording that will not be processed analytically. From the sample census page there are certain items that might be managed in this way.

```
Grocer%master, employing 2 men
Corset Maker%apprentice
```

The particular symbols ("%") and ("#") do not have to be used to denote original material and comments but the system is set up to use them. They can be changed using the configuration program.

Fuller details about aspects can be found in Section 4.2.4.1 of the Reference Manual.

4.3.8 An alternative format for data entry

At this stage it is necessary to suggest an alternative format for data entry. The following example shows another way of presenting a data file in a κλειω format. However this format is only recommended where there are likely to be more than, say, seven ‘empty’ elements in sequence. More than seven slashes in sequence are likely to cause confusion and make it harder to sort out mistakes. On the other hand it would take considerably longer to produce a data file in this format. The following example would be an acceptable way of describing data for κλειω to understand:

```
head$status=mz/firstname=Michael/surname=Morrison/age=31/
   occupation=Licensed Victualler/birthto=Islington/
   birthco=Middlesex
relp$status=fz/firstname=Mary/fname2=J/surname=Morrison/
   relation=Wife/age=29/birthto=Cendover/
   birthco=Salop
relp$status=f/firstname=Ellen/surname=Morrison/relation=Daur/
   age=7m/birthto=Shrewsbury/birthco=Salop
```

WARNING. If you use this system of denoting elements, it must be done regularly. The line:

```
relp$status=fs/firstname=Catherine/surname=Doyle/servant/24/
   Barmaid/country=Ireland
```

will result in an error message. Once a name of an element has been used to describe an element in a group all the rest of the names of elements must also be described. Thus,

```
relp$fs/Catherine/Doyle/servant/24/occupation=Barmaid/
   country=Ireland
```

would not result in an error message.

`number` data types

Chapter 6 gives more details on how κλειω manages different types of numbers and the **`number`** data type. In the example above, there is one entry under 'age' which would not fit into the usual pattern. Ellen, the daughter of Michael Morrison, is 7 months old. This could be represented as a decimal, but it is also possible to represent it as a number of months. In the final representation of the data this entry is shown as 7m. This will be explained later.

4.3.9 Conclusion

At this stage it might normally be useful to add some more data to the data file. However, although it was said that the whole of the data file could be constructed before proceeding to the .mod file, it was also said that the construction of a κλειω database was essentially a compromise between two different techniques. Though one might well normally add more data at this stage, we shall instead proceed with the structure declaration and some of the logical environment of the database, as a way of checking whether the data has been entered correctly.

The final list of groups and elements is as follows:

```
reference
   refnum

address
    parish/town/ward/parlb/sandis/ecclp

house
   status/address/name
```

```
schedule
  schednum

head
  status/firstname/fname2/surname/age/occupation/birthto
  /birthco/country

relp

  status/firstname/fname2/surname/relation/age/occupation
  /birthto/birthco/country
```

The structure for the database could also be written down, as follows:

```
reference
    |
 address
    |
  house
    |
schedule
    |
  head
    |
  relp
```

4.4 Structure declarations

Once all the groups and elements relating to a particular database have been chosen it is possible to start to define the commands to allow κλειω to create a database. The usual way to decide what the groups and elements are to be is to construct a sample of the database. Conventionally structure declarations are held in a file with a .mod extension, but there is no reason why they should not be placed immediately above the data in the file that holds the data. This is because the structure declaration, when compiled, creates an empty database, to which data is later added. κλειω will not allow you to put data into a database that does not already exist. We suggest that you open a file called censsamp.mod.

All structure declarations *must* consist of:

- a **`database`** command
- at least one **`part`** directive
- an **`exit`** command.

If a number of elements contain data that is not of the default type (**`text`**) then a number of **`element`** directives are also necessary. In our example we have two elements, status and age, which are not of **`text`** type data; these are respectively of **`category`** and **`number`** type.

4.4.1 The `database` and `exit` commands

Both **`database`** and **`exit`** commands *must* occur in every structure declaration. They are to be found on the first and last lines of every structure declaration. Without them κλειω would not recognise that a database is about to be created.

The **`database`** command indicates to κλειω that a new structure declaration is about to follow.

The **`exit`** command indicates that a structure declaration has been completely defined.

Commands in structure declarations must be formatted in the same way as all other κλειω commands, inasmuch as they must start in the first column of a new line and can continue over as many lines as is necessary as long as the first column of any subsequent line is blank.

The `name=` parameter

The next task to be performed when creating a structure declaration is to choose the name of the database. This forms the parameter value for the **`name=`** parameter for both the **`database`** and the **`exit`** commands. The **`name=`** parameter *must* be specified and *must* contain a user-defined name for the database which can also be found in the data file. (This parameter must contain the same value as the **`name=`** parameter in a **`read`** command.) In this case we have already decided that this database is to be called censsamp, so we must define this structure declaration as belonging to that database.

```
database name=censsamp
...
exit name=censsamp
```

You should type these lines into a file called censsamp.mod.

The **`first=`** parameter

The **`first=`** parameter immediately follows a **`name=`** parameter. It *must* be specified and have as its parameter value the name of the top level of the hierarchy being defined, i.e. the document level.

```
database name=censsamp;first=reference
...
exit name=censsamp
```

The **`overwrite=`** parameter

Another parameter used in the **`database`** directive is **`overwrite=`**. When followed by the keyword **`yes`**, this parameter allows the user to overwrite a previously created database, with the same name. If this parameter is used when there is no database with the same name in existence it will be ignored. However, if this parameter is not specified, and there is already a database with the same name in existence, κλειω will not allow it to be overwritten. This is a particularly useful parameter to use when creating a database, as it allows the user to make corrections to the data file without having to destroy all the structure files (found in the directory with an .s** extension) each time the database is recompiled.

This parameter is usually placed at the end of a **`database`** directive, so that it can be removed once the database is correctly compiled.

```
database name=censsamp;first=reference;overwrite=yes
...
exit name=censsamp
```

Note that the order of the parameters in the **`database`** command, as with every other command, is unimportant (unless one parameter affects the operation of another). This line could have been written like this:

```
database overwrite=yes;name=censsamp;first=reference
```

4.4.2 The **`part`** directive

The **`part`** directive defines the properties of a group. There are two essential properties of any group:

- Its name
- Its connection with any other group within the database.

Every group contained within a database *must* be defined in a **`part`** directive; often this means that each group has its own **`part`** directive, but it is possible for two or more groups to share exactly the same characteristics, so one **`part`** directive could define more than one group. For instance, if one were creating a database using birth certificates as source material, where each individual has (normally) two specified parents, it would be possible to have one group for each parent, which would have the same elements and be related to the same group in the same way:

```
          person

        /        \

  mother          father
```

The **`name=`** parameter

Every **`part`** directive *must* be followed by a **`name=`** parameter. The parameter value *must* be the name of at least one group. Every group within a database *must* be specified in a **`name=`** parameter, and every group in a database specified by a **`name=`** parameter *must* also be specified in a **`part=`** parameter (see below) *EXCEPT* the group defined in the **`first=`** parameter of a database directive. In the case of the censsamp database, the first two lines and the last line of the structure declaration would look like this:

```
database name=censsamp;first=reference;overwrite=yes
part name=reference;
...
exit name=censsamp
```

The **`part=`** parameter

The **`part=`** parameter takes as a value the name of a group, which must also be defined in a **`name=`** parameter of another **`part`** directive. This parameter tells κλειω which groups are dependent on another. This should clarify the matter of why the group in the **`first=`** parameter does not have to be in a **`part=`** parameter; it is because the highest level in a hierarchy cannot be dependent on any other group. So the structure of the imaginary database of birth certificates might look like this:

```
database name=birth;first=person;overwrite=yes
part name=person;
  part=mother,father
...
exit name=birth
```

Compare this with the structure diagram on the previous page. That shows that the groups mother and father are both dependent on the group person.

In the case of the censsamp database the beginning of the structure file would now look like this:

```
database name=censsamp;first=reference;overwrite=yes
part name=reference;
  part=address
...
exit name=censsamp
```

Note that in both of these cases the **`part=`** parameter has been shown on the line following the **`part`** directive. This is for clarity – though of course all parameters relating to a single directive can be contained on the same line.

With the census database all the **`part`** directives needed for the whole database could be added, as follows:

```
database name=censsamp;first=reference;overwrite=yes
part name=reference;
  part=address
part name=address
  part=house;
part name=house;
  part=schedule
part name=schedule;
  part=head
part name=head;
  part=relp
part name=relp
...
exit name=censsamp
```

From the bottom upwards this signifies that the group **`relp`** is dependent on the group **`head`**, the group **`head`** is dependent on **`house`**; **`house`** on **`schedule`**; **`schedule`** on **`address`**; **`address`** on **`reference`**. In this case each group is only dependent on one other. It is perfectly possible to have many groups dependent on another, as in our fictitious database of birth certificates where both mother and father are dependent on the person. Notice that in that example – **`part=mother,father`** – the two names of the groups would be separated by a comma.

The `always=` parameter

It is not mandatory to specify an **`always=`** parameter in a structure declaration. This parameter is used to specify that a group in a **`name=`** parameter must appear a number of times in the database. For instance in this database we might want to specify that every schedule must be followed by at least one head of household. (In reality we don't want to do this because there are sometimes cases of uninhabited houses which obviously would not have to have a head of household.)

```
part name=schedule;
   part=head;
   always=head
```

If it were necessary to specify that at least two heads of household should appear, the line would be written as follows:

```
always=head:2
```

The `only=` parameter

This parameter is similar to the **`always=`** parameter. It is used to specify that a certain group should not have too many instances of a particular group dependent on it. If we wanted to specify that each schedule should have one and only one head of household dependent on it, we could introduce the following line into the structure file:

```
part name=schedule;
   part=head;
   always=head;
   only=head
```

It would be possible to introduce this parameter in this particular database because, even though there are sometimes more than one head of household in a house, each family unit should always be given a schedule number.

The `position=` parameter

The **`position=`** parameter is used to define the order in which elements may appear in a group. It is followed by a list of user-defined element names. This parameter does not have to be specified in a structure definition; however, to facilitate data entry it is often wise to use it, especially when one is using structured data, such as this census material.

In the case of the censsamp database the effect of adding this parameter to the structure definition would be as follows:

```
database name=cennsamp;first=reference;overwrite=yes
part name=reference;
   part=address;
   position=refnum
```

```
part name=address;
  part=house;
  position=parish,town,ward,parlb,sandis,ecclp
part name=house;
  part=schedule;
  position=address,name
part name=schedule;
  part=head;
  position=schednum;
  only=head
part name=head;
  part=relp;
  position=status,firstname,fname2,surname,age,occupation,
    birthto,birthco
part name=relp;
  position=status,firstname,fname2,surname,relation,age,
    occupation,birthto,birthco
exit name=censsamp
```

Note that the element country has been omitted on purpose. The explanation follows below.

Specifying the element in a **`position=`** parameter ensures that the data can be input without using the element name. For example, if there were no **`position=`** parameter for the group **`relp`**, a census entry would have to be represented thus:

```
relp$status=fz/firstname=Mary/fname2=J/surname=Morrison/
  relation=wife/age=29/birthto=Cendover/birthco=Salop
```

With the **`position=`** parameter we can specify the position of the logical columns in the database; for example, the data can be represented as follows:

```
relp$fz/Mary/J/Morrison/wife/2//Cendover/Salop
```

and if necessary we can add extra columns that have not been defined in the **`position=`** parameter:

```
relp$fs/Catherine//Doyle/serv/24/Barmaid/country=Ireland
```

This would be acceptable even though the element country was not defined in the **`position=`** parameter. In this case it would be possible to search a database for all the occurrences of the country in which an individual was born, in the same way as any element defined in the **`position=`** parameter.

The implication of what we have just said is a feature that is highly unusual in data processing. In κλειω, elements can be created and used within a database without formally being defined in a structure declaration.

The **alias=** parameter

The **alias=** parameter, when used within a **part** directive, allows the user to define a number of groups to share similar elements to be accessed in a single command. In the following case we have asked κλειω to assume (for some purposes) that we would like it to understand that both the groups p and relp can be accessed using the group name person.

```
part name=head;
  part=relp;
  position=status,firstname,fname2,surname,age,occupation
    birthto,birthco;
  alias=person
part name=relp;
  position=status,firstname,fname2,surname,relation,age,
    occupation,birthto,birthco;
  alias=person
```

Thus a task like the following would be considered legitimate:

```
query name=censsamp;part=person:surname="Morrison"
write part=:each[]
stop
```

This task would access both groups p and relp and look for all the surnames that contain the character string "Morrison". This would be the same as asking the following task:

```
query name=censsamp;part=:surname="Morrison"
write part=:each[]
stop
```

but there will be times when you want to access both groups and be unable to do so without using this parameter.

4.4.3 The **element** directive

The **element** directive, like the **part** directive outlined in Section 4.4.2, is used in conjunction with the **database** command. The **element** directive is used to specify the properties of elements so that they are different from the default properties. The previous chapter has explained in some detail the different data types that κλειω supports, and as not all of the data in this database is of the default type (**text**) we must demonstrate how κλειω can be told that it is to process data in a way different to that in which it processes textual data. In this database there are two types of data that are not of type **text**. These elements are status, which holds abbreviations like f for female and l for lunatic and is of data type **category**, and age, which is of data type **number**. The properties of these two elements must be defined so that κλειω will not interpret them as text. This is achieved with the use of a number of `element` directives.

The `name=` parameter

With the `element` directive the `name=` parameter takes the name of an element, that is usually not of the default type (`text`). Default data may also sometimes have to be defined.

First, the elements must be listed.

```
   element name=status
   element name=age
```

Though it is not applicable in this database, it would be valid to describe two different elements in the same `name=` parameter. These elements would have to separated from each other with a comma. Any number of elements may share the same `name=` parameter within an `element` directive.

The `type=` parameter

With the `element` directive the `type=` parameter accepts a keyword which defines the data type that relates to the properties of the element defined in the `name=` parameter of an `element` directive. In this case the two data types necessary are `number` and `category`.

```
   element name=status;type=category
   element name=age;type=number
```

Parameters referring to particular data types

Although the elements status and age have been defined as having the properties of elements with `category` and `number` type data respectively, it may also be necessary to define a logical object which gives information to κλειω on how to interpret the data found in these elements. This will become apparent in the next section.

For the censsamp database there are problems with both of these elements. First, though the element status has been defined as of data type `category`, we have not yet explained to the system what those abbreviations mean. Similarly for the element age the abbreviation m (for months) has not been defined. κλειω has not been told how to interpret this information.

If a logical object is to be used to define the parts of an element, κλειω must be told what this logical object is called.

This is done as follows:

```
element name=status;type=category;category=abbreviations
element name=age;type=number;number=ages
```

The names of all data types can be used as parameters within an **`element`** directive. In this case the **`category`** data type has become the **`category=`** parameter and the **`number`** data type has become the **`number=`** parameter. These are followed by user-defined names for the logical objects that will be described.

These **`element`** directives must appear somewhere within the structure declaration and it is usual to place them in the first line following the **`database`** command.

4.5 Logical objects

Logical objects are part of the logical environment of κλειω. Each logical object gives κλειω instructions on how to understand a particular piece of data. (See Section 7.3 of the Reference Manual for more details.) Logical objects will be described fully in Chapter 6.

Logical objects come in two forms. This section describes briefly only one of these, the form of logical object declared in an **`item`** command.

Declarations of logical objects do not have to be kept in the same file as the structure declaration for a database. Some are, as they are immediately needed to interpret the data in the database. Others are created and used on an *ad hoc* basis, and kept in separate files.

Logical objects are made up of three parts:

- an **`item`** command
- any number of directives
- an **`exit`** directive.

We suggest that you should place the logical object above the structure declaration, though it could be placed below.

4.5.1 The `item` command and the `exit` directive

Both the **`item`** command and the **`exit`** directive must have a **`name=`** parameter, which for one logical object must be identical and which must also appear in an **`element`** directive in the structure declaration of a database. In other words, for

```
item name=abbreviations
...
exit name=abbreviations
```

there must be a corresponding **`element`** directive in the structure declaration.

4.5.2 The `usage=` parameter

The **`usage=`** parameter of an **`item`** command must also be specified. It accepts a certain keyword which specifies what sort of logical object is about to be defined. In the case of the two logical objects that are being defined here, the keywords are **`category`** and **`number`**, which relate to the types of data that are being defined.

```
item name=abbreviations;usage=category
...
exit name=abbreviations


item name=ages;usage=number
...
exit name=ages
```

Note that, although it is not coincidence that the names of these keywords are identical to the names of data types, they are not the same thing as data types. A number of other keywords can be used to describe the type of logical object. Many of them will eventually be explained in this book; the remainder can be found in the Reference Manual (see Chapter 6 for references).

Each different type of logical object has a variety of rules which specify exactly how that data in the database should be interpreted. For the time being only those that concern this data will be described here.

4.5.3 The **`category`** declaration

The **`category`** declaration defines how κλειω should interpret data of type **`category`**. (See Section 7.3.1.5 of the Reference Manual for more details.)

For this database there are a number of **`sign`** directives and a number of **`part`** directives.

The **`sign`** directives followed by a **`signs=`** parameter and a **`write=`** parameter tell κλειω to interpret **`category`** type data as user-defined character strings. For instance:

```
   sign signs=m;write="male"
   sign signs=f;write="female"
```

tells κλειω that when it encounters a piece of **`category`** type data with the entry **`m`**, it should interpret it as being the same as the word male.

The whole of the logical object for this particular type of category data is as follows:

```
   item usage=category;name=abbreviations
   sign signs=d;write="deaf and dumb"
   sign signs=b;write="blind"
   sign signs=e;write="imbecile"
   sign signs=l;write="lunatic"
   sign signs=o;write="lodger"
   part
   sign signs=z;write="married"
   sign signs=w;write="widowed"
   sign signs=s;write="unmarried"
   part
   sign signs=i;write="inhabited"
   sign signs=u;write="uninhabited"
   sign signs=a;write="building"
   part
   sign signs=m;write="male"
   sign signs=f;write="female"
   exit name=abbreviations
```

The **`part`** directive in a **`category`** declaration is a rather useful method of verifying data. **`sign`** directives that follow a **`part`** directive are mutually exclusive. Thus while in the case of the first batch of **`sign`** directives any of the abbreviations can be used together (for instance a person can be blind, a lodger and deaf and dumb), the remaining three batches, each of which is preceded by a **`part`** directive, can only be represented by one abbreviation from each. This means that it would not be possible to define an individual as both male and female, or a building as both inhabited and uninhabited.

Note also that in this context the **`part`** directive appears on its own, without any specification.

4.5.4 The **number** declaration

The **number** declaration defines how κλειω should interpret data of type **number**. (See Section 7.3.1.4 of the Reference Manual for more details.)

In the case of the censsamp database there is only one item to define and that is the use of the abbreviation for month. For κλειω to interpret the letter m in the element age as one twelfth of a year, the following logical object must be defined:

```
item name=ages:usage=number
text name="m";number=0.08333333
exit name=ages
```

This means that κλειω will multiply every number with the abbreviation of m in an age element by 0.0833333 (which is incidentally one twelfth).

4.6 Compiling the database

The logical object called ages should complete the .mod file for this data. Earlier we decided that the database should be called censsamp and that the file containing the data to be called censsamp.dat. We would also recommend that you type in the .mod file using the information given in this chapter. It would be usual to call this file censsamp.mod and the data file censsamp.dat. This will make life considerably easier for you when you have a number of databases all running on the same machine. None of this is compulsory. The data file could be called hello and the .mod file goodbye. But for consistency and ease of use you might consider giving obviously related names to these separate files.

When you have two files, one containing the logical environment and structure of the database (which should be called censsamp.mod) and the other containing the data (censsamp.dat), you will be ready to compile the database. This is achieved, as we have seen earlier in this chapter and in Chapter 1, as follows:

First, compile the file censsamp.mod by typing **`kleio censsamp.mod`** at the DOS prompt. If κλειω produces any error messages, ignore them for the time being and compile the file testing.mod which can be found on the demonstration disk. This is a ‘correct’ version of the .mod file for this database. It will also produce a structure for a database called censsamp.

When either (but not both) of these tasks has been achieved, κλειω will allow you to compile the data file. This is done by typing **`kleio censsamp.dat`** at the DOS prompt. In case there are any errors in your database, we would recommend that you send the results of this operation to a file. This is achieved by typing the name of the result file after censsamp.dat, e.g. **`kleio censsamp.dat result`**.

When κλειω has completed this task, use your screen editor to examine this result file. If there are no error messages and the end of the file looks exactly like this:

```
Number of element names defined: 21
Number of group names defined: 7
Number of generic names for elements: 0
Number of generic names for groups: 0
Number of documents contained: 1
Number of groups contained: 36
Number of elements contained: 186
```

you will have successfully created a database, in which all the data you have entered will have been successfully stored. If you have managed to get everything right at your first attempt, see if you can discover an easy way of finding out whether or not every single element that you have input has gone into the correct logical column. You may find for instance that you have a place of birth in the occupation element. It should not be too difficult to fix this type of problem.

Trouble-shooting

If your result file does not look like this, and there are error messages, there must be errors that κλειω has objected to. There are two possibilities here, which we cannot foresee because we do not have your data file or your result file in front of us. First, you may have made a mistake in your .mod file. The quick way to ascertain whether this is the case is to use *our* .mod file, called testing.mod. Run this file (which will overwrite your database) and then run your data again. If this solves the problem, that suggests that you have made some small but important error in copy-typing the .mod file.

If after running testing.mod you still have a number of error messages, these will be errors in the data file. Do not despair, even if it looks as though there are hundreds of them. It often turns out that, if you can identify the first error, you may have solved all of them.

If your error messages look something like this, it should not be too difficult to trace the problem in the original data file:

```
***** Error: You are using an unknown qualifier: Morrison

Part of the input has been ignored due to an error in the data type.
reference (1 = "RG11-5490"),address (1 = "add-1"),house (1 = "hou-1"),schedule (1 = "sch-1"),
head (1 = "*  (Unknown)"),age

***** Error: Surplus POSITION field. Contents ignored.
reference (1 = "RG11-5490"),address (1 = "add-1"),house (1 = "hou-1"),schedule (1 = "sch-1"),
head (1 = "*  (Unknown)"),
```

These two errors relate to the same problem. Errors of this sort should be corrected one at a time, as only one change needs to be made to correct these errors. κλειω does not say what needs to be changed but tells us almost exactly where in the data file this error can be found. In this case the error can be found in the first document (reference (1="RG11-5490"), in the first address, the first house, the first schedule and the first head of household in the element age. κλειω also tells us what it is having problems with. In this case we are using an unknown qualifier in the age group of that individual. This unknown qualifier is the word Morrison. As we have seen above, we have told κλειω that ages must be numbers and that the only textual information we can use within the group age is the letter **m**. If we refer back to the data file to see what is wrong here we may find that this line has been incorrectly typed, for example like this:

```
head$mz/Michael///Morrison/31/Licensed Victualler/Islington
   /Middlesex
```

In this case we have added an extra element delimiter, which has moved the text "Morrison" into the age element. κλειω has reacted to this by displaying an error message.

The second error message displayed above also relates to this problem. As each of the elements in this group have moved along one place, κλειω has been unable to allocate the text Middlesex to a group. In the error message κλειω describes this as having a surplus position field. If we had corrected (by removing one slash) the first error, the second would not have occurred either.

Another example of a common error message is shown below:

```
***** Error: An incorrectly used group was ignored: head
reference (1 = "5490"),address (1 = "add-1"),house (2 = "hou-2"),

***** Error: Following data ignored due to an invalid group identifier:
reference (1 = "5490"),address (1 = "add-1"),house (2 = "hou-2"),

head$mz/Lambert//Newton/39/Grocer%master,employing 2 men/Wigton/Cumberland
```

(If this message was received there would also be a further thirty lines or so of error messages but we have removed the end ones in this example.)

Here κλειω is telling us that we have used a group in the wrong place, and that that group is head. This head can be found in the second house listed in the data file.

We have created this error by removing the line

```
    schedule$5
```

from the data file, and as κλειω expects a group called schedule after a group called house, it is complaining that we are not using a group in a correct position.

The second part of this error message then tells us that the data within that group has been ignored as κλειω has attempted to fit it into a schedule group. Thus it ignores that data.

A final example of an ordinary error message:

```
***** Error: The following Category expression contains illegal character(s): qfu

Part of the input has been ignored due to an error in the data type.
reference (1 = "RG11-5490"),address (1 = "add-1"),house (6 = "hou-6"), schedule (1 = "sch-1"),
head (1 = "hea-1"),relp (2 = "* (Unknown)"),status
```

Having seen the above examples, you should not find it too difficult to discover what this means. Here κλειω has told us that a category expression contains illegal characters. The particular entry that contains the error is made up of the characters **`q`**, **`f`** and **`u`**. In this case, even though κλειω gives us a precise location for this error, it also gives us the whole of the element in which the error occurs. If we were to return to the data file and search for the string **`qfu`** we should find it. And as we have not defined the category code **`q`** we should either delete it from the data file or include it within the logical object defining the element status in the .mod file.

The only real way of learning about interpreting κλειω error messages is to try and work out what they mean. If your particular error messages are not similar to these it would be useful to attempt to sort them out. If you are unsuccessful it may be better to try and use the original material in conjunction with the .mod file to see where you have gone wrong.

There is a further facility which helps you to identify the exact position of an error. If you added the parameter **`repeat=yes`** to the **`read`** command in the data file, κλειω would print out all the data in the datafile as it is 'put' into the database. Any error messages will appear immediately after the input data with an error message in it.

There is at least one other message which you may manage to produce. It might look like this:

```
An ambiguous group identity has been disambiguated. RG11-5490
reference (2 = "RG11-5490"),refnum
---> RG11-5490a
```

κλειω is telling you that you have compiled the database only once but you have added the same data to that database twice. In this case, if you look down your result file you will see lines similar to these:

```
Number of groups contained: 72
Number of elements contained: 372
```

which would convince you that you have exactly twice the number that you would expect of groups and elements in your database.

This error message has occurred because κλειω will not allow you (unless you define otherwise) to have two documents with the same identifier in the same database. The solution to this problem (as you might encounter it here) is to recompile the .mod file and then recompile the .dat file.

This might be the place to explain where how "identifications" are derived. As long as you do not override the default, κλειω will identify each document by the contents of the first element which is encountered in every document (more precisely, by the first twelve characters encountered in that element or the characters up to the first space character, whichever occurs first).

A tip. Correct error messages one at a time, then recompile both the .mod and .dat files. This may be time-consuming, but, as we have said, making one change to the data file might correct a thousand error messages. We hope that you have no problems in successfully compiling this database. When you have done so, try querying it to see if you have managed to input the data correctly. For instance, you may have missed a slash out, thus combining two elements and making a following piece of information occur in the wrong place.

Exercises

Using the censsamp database:

Exercise 4.1

Produce a list of all the related people who have occupations, listing their occupation, surname and first name in that order.

Exercise 4.2

Display all the information about all the women in the database aged 29.

Exercise 4.3

Modify the last task to produce all the information about all the women in the database whose age is greater than 29 (you will find some help in Chapter 3).

Exercise 4.4

Produce an alphabetical list (by surname) of all the married women aged less than 30, with their first names and their ages.

Exercise 4.5

Write a task to find out how many houses were uninhabited. Produce a list of all of the addresses.

Exercise 4.6

Write a task to produce an alphabetically sorted list of all those people whose town of birth is not given as Shrewsbury.

Exercise 4.7

Produce an alphabetically ordered list of all the surnames, first names and occupations of all the unmarried (i.e. widowed or single) men whose occupation does not include the character string "grocer".

Exercise 4.8

Produce an alphabetically ordered list of all the children under 16 who are not scholars. Use the keyword **`not`**.

4.7 Further features

The structure declaration that we demonstrated with the censsamp data was relatively simple. This is because census material is relatively structured material. However κλειω allows the user to define much more complicated structure declarations, which might contain material from more than one source, contain more than one group with the same name, and have many branches in a complex hierarchy. In order to explain κλειω we have necessarily kept the structures of the databases used in this tutorial rather simple. In Chapters 6 to 10 we will be using a database with a slightly more complicated structure, which will put into practice some of the concepts that we will be discussing here.

Earlier in this chapter we described the steps necessary for the construction of a database. We only included parameters and commands which are essential or most useful in database creation. There are a number of other parameters which can be used to help define a database. In the following few pages we shall describe these parameters.

4.7.1 Further parameters for the `database` command

We have said that the **`database`** command tells κλειω that the definition of a database is about to begin. The parameters that follow this command define some of the characteristics of that database. We have already described the **`name=`**, **`first=`**, and **`overwrite=`** parameters as used with this command. There are four other parameters which can also be used. None of them have to be used; they are all optional. Only one will be described here. Two others will be described elsewhere as they refer to the identification and addressability of identifiers.

The `write=` parameter

If we add a **`write=`** parameter to the censsamp structure declaration, thus:

```
database name=censsamp;first=reference;overwrite=yes;
  write=structure
```

when we compile the database we will get more information about its structure. If this parameter is added to the database, the following information will be included at the end of the result:

Schema of data structure for documents named: reference

```
* reference
*
*  - refnum
*
* * address
* *
* *  - parish
* *  - town
* *  - ward
* *  - parlb
* *  - sandis
* *  - ecclp
* *
* * * house
* * *
* * *  - status
* * *  - address
* * *  - name
* * *  - site
* * *
* * * * schedule
* * * *
* * * *  - schednum
* * * *
* * * * * head
* * * * *
* * * * *  - status
* * * * *  - firstname
* * * * *  - fname2
* * * * *  - surname
* * * * *  - age
* * * * *  - occupation
* * * * *  - birthto
* * * * *  - birthco
* * * * *
* * * * * * relp
* * * * * *
* * * * * *  - status
* * * * * *  - firstname
* * * * * *  - fname2
* * * * * *  - surname
* * * * * *  - relation
* * * * * *  - age
* * * * * *  - occupation
* * * * * *  - birthto
* * * * * *  - birthco
```

This is a representation of the database as it has been defined to κλειω. There are six levels of the hierarchy, starting with reference and ending with relp, and a certain number of elements (denoted by a `-`) within those elements.

Currently the **`write=`** parameter takes four different values, all of which provide different information about the database that you have constructed. Three of these values are **`names`**, **`parts`**, and **`structure`**. (The fourth value, **`generic`**, need not concern us here; refer to the Reference Manual, Section 5.2.3.1.)

- **`names`**. Tells κλειω to describe all the elements in the database. Below is some sample output from the censsamp database:

```
Properties of elements named: age
Preferred data type for basic information: Number
Preferred data type for "comment" field: Text
Preferred data type for "original" field: Text
Text data is processed according to the following declaration: text
Number data is processed according to the following declaration: ages
Multiple entries are logically subordinated to aspects.
```

- **`parts`**. Tells κλειω to describe all the groups in the database. Sample output:

```
Properties of groups named: head
The identifying prefix is: hea
Elements appear in order specified in data model.
Groups appear in the order specified in the data model.
In the input, this identifier represents an alias for: person

List of elements (number, {length}, properties):
status (0, Position)
firstname (0, Position)
fname2 (0, Position)
surname (0, Position)
age (0, Position)
occupation (0, Position)
birthto (0, Position)
birthco (0, Position)

List of groups (number, properties):

relp (0)
```

- **`structure`**. Tells κλειω to describe the structure of the database. See above.

The **`write=`** parameter can be specified more than once in a **`database`** command. It would be valid to add the following parameters to the **`database`** command:

```
database name=censsamp;first=reference;overwrite=yes;
  write=names;write=parts;write=structure
```

This command will produce information describing the elements in the database (from the **names** keyword), the groups in the database (from the **parts** keyword) and the structure of the database (from the **structure** keyword).

4.7.2 Further parameters for the `part` directive

There are currently twenty-three different parameters that can follow a **part** directive in a structure declaration. Only one of these must be specified (the **name=** parameter), and we have discussed a number of other useful parameters in the earlier part of this chapter. There are a number of others that may be of (immediate!) use to you.

The `write=` parameter

This parameter can be added to the structure definition, thus:

```
part name=head;write="head of household"
  part=relp;
  position=status,firstname,fname2,surname,age,occupation,
    birthto,birthco;
```

This parameter must be followed by a constant, and thus enclosed in inverted commas. This means that when a task is run, κλειω will print the name of the constant in the **write=** parameter rather than the name of the group as specified in the **name=** parameter. For instance if we ran a simple task without this parameter part of the result would look like this:

```
head (1 = "hea-1")
        age             31.000000
        occupation      Licensed Victualler
        status          married, male
        firstname       Michael
        surname         Morrison
        birthto         Islington
        birthco         Middlesex
```

If we were to change the .mod file by adding the **write=** parameter as above, the same task would provide this result:

```
head of household (1 = "hea-1")
        age             31.000000
        occupation      Licensed Victualler
        status          married, male
        firstname       Michael
        surname         Morrison
        birthto         Islington
        birthco         Middlesex
```

However, this parameter only changes the way in which κλειω displays the information on the screen. It would not allow us to perform a task like this:

```
query name=censsamp;part=head of household
write
stop
```

If you specify the name of a group in a **`part=`** parameter following a command in a task, it must be defined in a **`name=`** parameter of a **`part`** directive in the structure declaration.

The **`source=`** parameter

The **`part`** directive also allows one to use a parameter, **`source=`**, to tell κλειω that the properties of one group are to be identical to those of another. This parameter cannot be demonstrated effectively using the censsamp database. The **`source=`** parameter takes as a value the name of a group already defined in a database. In an imaginary database the following part of a definition,

```
part name=mother;
  part=child;
  position=firstname,surname
part name=father;
  source=mother
```

would mean that the group mother contained two elements, firstname and surname, and had one group, child, that was dependent on it. It would also mean that the group father had two elements, firstname and surname, and had a group, child, dependent on it. As structure definitions of databases do not have to be entered in the same order as the hierarchy of the database, this parameter can be used in what might look a rather odd position. For instance:

```
part name=mother;
  position=firstname,surname
part name=father;
  part=mother;
  source=mother
```

However, the **`source=`** parameter cannot be used in a group which is going to take another group's properties before the group that holds the properties has been defined.

The `guaranteed=` parameter

The **`guaranteed=`** parameter is a useful parameter which takes as a value a list of elements. Each of the elements included in this list must be contained in this group. For instance, in the database censsamp all people in the database must have a surname. Thus it would be possible to change the censsamp .mod file to look like this:

```
part name=head;
   guaranteed=surname;
   part=relp;
   position=status,firstname,fname2,surname,age,occupation,
        birthto,birthco
part name=relp;
   guaranteed=surname;
   position=status,firstname,fname2,surname,relation,age,
        occupation,birthto,birthco
```

Any number of elements can be referred to in a **`guaranteed=`** parameter. The correct format for this involves the use of commas between the elements:

```
guaranteed=firstname,surname,relation
```

The **`guaranteed=`** parameter is particularly useful for data validation.

The `order=` parameter

When using a **`write`** command in a task, you may sometimes wonder why κλειω doesn't preserve the order of the elements as you entered them into the data file. The following result of a task:

```
reference (1 = "RG11-5490") : relp (5 = "rel-5")
        age             24.000000
        status          unmarried, female
        firstname       Catherine
        surname         Doyle
        occupation      Barmaid
        relation        servant
        country         Ireland
```

lists the data in a different order to its input state. This is because κλειω lists data in the order the names of the elements appear in the .mod file. In this case the age and status elements both have logical objects associated with them (i.e are mentioned in an **`element`** directive before the other elements) so they are listed before the other data. If you wanted to preserve the order of the input data, κλειω allows you to use the **`order=`** parameter following a **`part`** directive relating to a group. In the case of the censsamp database, this could be done as follows:

```
part name=relp;order=yes;
  position=status,firstname,fname2,surname,relation,age,
    occupation,birthto,birthco
```

The result of the same task would be:

```
reference (1 = "RG11-5490") : relp (5 = "rel-5")
        status          unmarried, female
        firstname       Catherine
        surname         Doyle
        relation        servant
        age             24.000000
        occupation      Barmaid
        country         Ireland
```

In this case the value for the parameter is **`yes`**. This tells κλειω to keep the elements in the output in the order in which they were input.

The sequence= parameter

The **`sequence=`** parameter performs a similar task to the **`order=`** parameter but deals with the order of groups entered in a database. When κλειω produces a result to a task, the groups may not be in the same order as they were in the original data. This parameter allows the user to change κλειω's default actions in circumstances where this may arise. It is unlikely, however, that this problem will affect you until you start using very complicated hierarchies for their data.

4.7.3 Further parameters for the `element` directive

Currently κλειω knows of twenty different parameters that can be used with the **`element`** directive. Many of these are only used for specialist applications; these will be discussed when they are needed during the course of this volume. There are other parameters which work in a similar way to those which work with the **`part`** directive, but whereas those deal with groups, parameters used with the **`element`** directive deal with elements.

The **write=** parameter

The **write=** parameter works for the **element** directive in exactly the same way as for the **part** directive. It allows the user to define a different name for an element for output than the one named in the **name=** parameter of a **element** directive or in a **position=** parameter. If we wanted to tell κλειω to use the textual string "Town of birth" to replace the element name birthto in output, we would need to add the following **element** directive to the structure declaration. This parameter would usually be used to speed up data entry. An element would be given a single character name for the purposes of data entry, but have a longer constant associated with it.

```
element name=birthto;write="Town of birth"
```

The database as originally created would produce a result which looked like this:

```
reference (1 = "RG11-5490") : head (1 = "hea-1")
          age            31.000000
          status         married, male
          firstname      Michael
          surname        Morrison
          occupation     Licensed Victualler
          birthto        Islington
          birthco        Middlesex
```

If the **write=** parameter described above was added the result would look like this:

```
reference (1 = "RG11-5490") : head (1 = "hea-1")
          age            31.000000
          status         married, male
          firstname      Michael
          surname        Morrison
          occupation     Licensed Victualler
          Town of birth  Islington
          birthco        Middlesex
```

The **source=** parameter

Another parameter which works in a similar way for the **element** directive as for the **part** directive, is **source=**. (See above, p. 83.)

The **`first=`** and **`second=`** parameters

κλειω has two other parameters, **`first=`** and **`second=`**, which allow the user to define what type of data is to be used in the comment and original aspects of an element. See the Reference Manual, Sections 5.4.3.2 and 5.4.3.3.

At the beginning of this section we said that many of the parameters that can be used with the **`part`** directive can also be used with the **`element`** directive. In fact there are parameters which have the same names for both directives but have different functions. None of these parameters are discussed in this volume. However, there is one that you may find it worthwhile to know about. The **`order=`** parameter, when used with the **`element`** directive, allows you to define the way in which κλειω considers aspects as parts of elements. (Consult the Reference Manual, Section 5.4.4.1.)

4.8 Dynamic fixed-field formats

There are times when it is necessary to input statistical data. κλειω was specifically designed to make it as easy as possible to input 'difficult' data rather than the rather 'easy' statistical data. This means that κλειω (until recently) only allowed the user to input statistical data in the same cumbersome fashion as all other data. We shall only mention that κλειω now has the ability to input more 'regular' data in an 'easier' fashion and direct anyone with an interest in using this type of data to Section 4.5 of the Reference Manual which describes in simple terms how to input this form of data.

Appendix: data signals

Each entry in a κλειω database must be composed of a character string made up of characters which are acceptable to the system with ten exceptions.

These ten exceptions are known as the data signals which perform a task that κλειω understands in a different way to the normal characters.

Data signal 1	$	Dollar sign
Data signal 2	/	Slash
Data signal 3	=	Equals sign
Data signal 4	#	Hash sign
Data signal 5	%	Percent sign
Data signal 6	<	Open angled bracket
Data signal 7	>	Close angled bracket
Data signal 8	;	Semi-colon
Data signal 9	:	Colon
Data signal 10	\	Backslash

These characters are always defined in the Reference Manual by their data signal number rather than the symbol they represent because κλειω allows the user to redefine these symbols, if they feel it necessary. In this volume we will almost always refer to these symbols by their English name to prevent initial confusion. To find out how to allocate different symbols to these data signals refer to the Reference Manual, Appendix B. We strongly, indeed emphatically, recommend that you do not redefine these symbols until you are an experienced κλειω user, as much of the information in this volume would then become inaccurate or confusing.

Further reading

For further reading on conventional database design techniques for historians, see:

C. Harvey & J. Press, 'The Business Elite of Bristol: a Case Study in Database Design', *History and Computing*, 3:1 (1991), pp. 1–11.

C. Harvey & J. Press, 'Relational Data Analysis: Value, Concepts and Methods', *History and Computing*, 4:2 (1992), pp. 98–109.

C. Harvey & J. Press, *Database Systems and Historical Research* (Macmillan, London, forthcoming).

On census material:

M. Anderson, C. Stott and B. Collins, 'The National Sample from the 1851 Census of Great Britain. An Interim Report on Methods and Progress', *Historical Methods Newsletter*, 10:3 (1977), pp. 117–121.

(see K. Schürer, S. J. Anderson & J. A. Duncan, *A Guide to Historical Datafiles held in Machine-Readable Form* (Association for History and Computing, London, 1992) pp. 220–225 for further details on this project and a fuller bibliography).

D. V. Glass, *Numbering the People: the Eighteenth Century Population Controversy and the Development of Census and Vital Statistics in Britain* (Saxon House, Farnborough, 1978).

E. Higgs, *Making Sense of the Census* (Public Record Office, London, 1989).

R. Lawton (ed.), *The Census and Social Structure: an Interpretative Guide to Nineteenth Century Censuses for England & Wales* (Frank Cass, London, 1978) (especially the chapters by Armstrong and Lawton).

E. A. Wrigley (ed.), *Nineteenth-Century Society. Essays in the Use of Quantitative Methods for the Study of Social Data* (Cambridge University Press, Cambridge, 1972) (especially the chapters by Armstrong, Drake, Anderson & Tillot).

Chapter 5

Further retrieval and display

5.1 Multiple entries in elements

The last exercise in Chapter 2 asked you to select all the occupations in the burial database which contained the character string "clerke". We suggested that the solution that you would be most likely to use would not bring up the correct answer. The following task would be the natural answer for you to give based on what you have already learned.

```
query name=burial;part=:occupation="clerke"
index part=:occupation
stop
```

However this will bring up a result like this:

```
clerke  p-6

clerke  p-9

petty cannon  p-6
```

Though this retrieves both of the individuals who were clerks, it also brings up a petty canon. This is because one of the two clerks listed in the database has two occupations.

κλειω retrieves both occupations separately, as we have asked for all the occupations of all the people who have the character string "clerke" within the element occupation.

The following task demonstrates both the task above and the 'correct' way of answering the question. This introduces a new command, **`continue`**. This command indicates that the task previously defined should be executed and that further commands are expected. If there are no further commands after the **`continue`** command, κλειω treats it as though it were the command **`stop`**. It is generally used as a connector between tasks that are to be run independently but sequentially, as in this example:

Example 5.1

```
query name=burial;part=:occupation="clerke"
index part=:occupation
continue
query name=burial;part=:occupation
index part=:occupation="clerke"
stop
```

The second part of this task gives this as a result:

```
clerke   p-6

clerke   p-9
```

Why do these two queries bring up different answers? The first could be 'translated' as:

> I am interested in groups in the database burial where the element occupation contains the character string "clerke". When these groups have been found, display the contents of all the elements called occupation.

As there is one person with two entries in the element occupation both of these entries are displayed. The second task 'translates' as follows:

> I am interested in groups in the database burial which have an element occupation. When all those groups have been found display the contents of the element occupation if it contains the character string "clerke".

There is an additional problem here. Why, you may ask, do we not ask κλειω to perform this task?

```
query name=burial
index part=:occupation="clerke"
stop
```

Try running it and see the result. You will notice that only one occupation is retrieved. Why? Use the text editor to examine the data file and see what the difference is between the two people who are described as "clerke".

Whenever κλειω performs a task, it looks for groups. In this case, as we have not specified a group in the **part=** parameter in the **query** command, κλειω assumes that we mean the document level of the database. In this case the document level of the database is the group p, therefore κλειω interprets this line as **query name=burial;part=p**. So, in order to get *all* the people who have occupations, the name of the element must be specified in the **query** command.

Just to show that the two tasks are different, run the following task, putting the result into an output file rather than displaying the results on the screen.

Example 5.2

```
query name=burial;part=:occupation="clerke"
index part=:occupation;
   part=:surname;
   part=:firstname
continue
query name=burial;part=:occupation
index part=:occupation="clerke";
   part=:surname;
   part=:firstname
stop
```

However there is one reason why one should not use the second task in **ex5.1** to get the result of the question posed at the end of Chapter 2. The reason will be made clear in the following section.

5.2 The element function **:query[]**

When tasks become longer and more involved, as they soon will, it will rapidly become clear that it is better practice to specify exactly what you want to be found in the parameter following the **query** command as well as in either the **index** or **write** commands. Thus the following task would produce the desired result.

```
query name=burial;part=:occupation="clerke"
index part=:occupation="clerke"
stop
```

However this is a rather messy way to specify the same condition twice. Instead of repeating a condition one can use the element function `:query[]`.

The element function `:query[]` means the entry in an element which fulfils the condition specified in the previous `query` command. So the following task is a more precise (and stylish) way of posing the previous task.

Example 5.3

```
query name=burial;part=:occupation="clerke"
index part=:query[]
stop
```

5.3 Output

In the following section we will be considering ways of formatting the output from tasks, both to make them more attractive and to sort the output. The commands described below only work with the command `index` and not with the command `write`. Some of these commands refer to all the columns of the output, some only refer to one at a time and yet others only refer to those columns immediately after the parameter has been reached.

Earlier we said that elements were similar to columns in traditional data processing. In a κλειω database there are no columns but there are logical columns. The term logical column is used in κλειω because there are no 'real' columns inherent within the database. The user defines columns as and when they are needed.

5.3.1. The `position=` parameter

In the following task κλειω displays the index in a tabular form. Using the parameter `position=` with the keyword `yes`, κλειω transfers each logical column into a printed column of the same width beginning at the same place on each line.

Example 5.4

```
query name=burial;part=:surname
index position=yes;
   part=:surname;
   part=:firstname;
   part=:title;
   part=:occupation
stop
```

Part of the result of this task is shown here:

```
Alderley     Robert                    singing man    p-30
Alexander                  Dr                         p-4
Alexander    Lucee                                    p-4
Appleford    Frances       Mrs         widow          p-64
Atkins       Luce                      widow          p-32
Badger       William       Mr                         p-51
```

If the keyword **yes** which follows the parameter **position=** is changed to **no**, the task will be displayed as usual. This parameter obviously affects all the columns of the output. (N.B.: When you run this task send the results to a file and scroll across the screen to see the whole result. If you send the result directly to the screen it will look awful, because the table of columns is wider than the screen!)

5.3.2 The **signs=** parameter

By using the parameter **signs=** with a number as a parameter value, one can arrange the output for that particular logical column to fit into a printed column of a fixed length of that number.

Example 5.5

```
query name=baptism;part=relp
index part=:surname;signs=12;
   part=:firstname;signs=20;
   part=:occupation;signs=12
stop
```

Here is some sample output:

```
Kercher                            p-59
Kercher                Dr of Divini p-20
Kercher                prebendary  p-20
Kercher     Robert                 p-18
Love                               p-48
Love                               p-54
.
.
.
Love                   prebendary  p-42
Love        Nicholas               p-28
Mason       Robert     councellour p-74
More        John                   p-1
Oglander    William                p-16
Pawlet      William                p-19
Perin                  prebendary  p-21
```

In the example above we have decided that the width of the printed column for the element surname is to be 12 characters long, for firstname 20 characters and for occupation 12. These are arbitrary figures, and any number could be used. If an entry is larger than the user defined length of the printed column the surplus characters are cut off.

5.3.3 The `without=` parameter

κλειω also allows the user to choose which of the logical columns defined are to be sorted. If, in this particular case, one wanted to sort by occupation, but keep the personal information displayed before it, one could set the following task:

Example 5.6

```
query name=baptism;part=relp
index part=:surname;signs=12;without=yes;
   part=:firstname;signs=20;without=yes;
   part=:occupation;signs=12
stop
```

The `without=` parameter expects either of the keywords `yes` or `no` to follow it. It allows the user to specify to κλειω that it should disregard logical columns for sorting purposes. If it is followed by `yes`, it means that the column is output by κλειω but not sorted. In this particular case the element occupation is the only one to be sorted. So the result of the task and the secondary search is to be sorted by occupation. A problem may arise here as κλειω sorts the empty logical columns first (in the order entered in the database).

5.3.4 The `write=` parameter

The parameter `write=` allows the user to sort an index without displaying the information. Try running the following example:

Example 5.7

```
query name=burial;part=relp
index part=:surname;write=no;
   part=:firstname;
   part=:occupation
stop
```

This task sorts the selected data by surname but does not display the contents of the element surname. (As a consequence it looks rather a mess.)

5.3.5 The `form=` parameter

The parameter **`form=`** is a simple parameter to sort entries and justify the output to the left or right.

The parameter **`form=`** accepts the keywords **`left`** and **`right`**. **`left`** is the default.

The following task demonstrates the usage of the parameter **`form=`**:

Example 5.8

```
query name=burial;part=relp
index part=:surname;form=right;
   part=:firstname;form=right;
   part=:occupation;form=right
stop
```

Here is some sample output:

```
  Love                            prebendary  p-25
  Love                            prebendary  p-59
 Beely                      fellow of ye Coll  p-50
 Cowse     Richard                             p-31
 Friar         Tho                             p-23
 Frost        John         one of the choire  p-45
```

Note that this has sorted in a rather curious way. It has right-hand justified the contents of each column, but sorted with the number of spaces before each word. You might ask why this has not sorted by surname. The answer to this is that it has; κλειω has put the surnames in order "most spaces first". This parameter is most effectively used to display numeric data.

5.3.6 The `first=` parameter

By default, when using the **`index`** command, κλειω sorts in ascending alphabetical order. If one wanted to sort a logical column in descending alphabetical order one could use the **`first=`** parameter followed by the keyword **`limit`**. By default κλειω acts as though the keyword **`start`** follows the **`first=`** parameter.

Example 5.9

```
query name=burial;part=:surname
index first=limit;position=yes;
   part=:surname;
   part=:firstname;
   part=:title;
   part=:occupation
stop
```

The parameters **`first=`** and **`position=`** both relate to the whole of the index. As such they have only been defined once and they are usually placed before the first column-defining parameter, as in the above example. But they can be positioned anywhere within the **`index`** command, as the following example demonstrates:

```
query name=burial;part=:surname
index part=:surname;
   part=:firstname;
   part=:title;
   part=:occupation;
   first=limit;
   position=yes
stop
```

5.3.7 The **`substitution=`** parameter

The parameter **`substitution=`** acts rather like a ditto mark. If two entries in successive lines in an index are the same, a user-defined string can be substituted to give emphasis to the index.

Running the following task demonstrates how this parameter may be used:

Exercise 5.10

```
query name=burial;part=:surname
index part=:surname;substitution="---//---";
   part=:firstname;
   part=:title;
   part=:occupation
stop
```

We had some puzzled looks when we first demonstrated this to our class of guinea-pigs, so here follows a digression on the parameter **`substitution=`**.

Some output from the following task is shown below it.

```
query name=burial;part=:surname
index part=:surname;
  part=:firstname;
  part=:title;
  part=:occupation
stop
```

```
Budd Alice     1_bur
Budd William Mr alderman 1_bur
Budd William Mr alderman 1_bur
```

With a **`substitution=`** parameter added to the end of the first line of the **`index`** command the following output is given:

```
query name=burial;part=:surname
index part=:surname;substitution="--||--";
  part=:firstname;
  part=:title;
  part=:occupation
stop
```

Part of the output from the previous task is shown below:

```
Budd Alice     1_bur
--||-- William Mr alderman 1_bur
--||-- William Mr alderman 1_bur
```

Notice that because the name Budd is repeated in the index it is replaced by the string --||-- which is the string defined in the **`substitution=`** parameter. If a further **`substitution=`** parameter was added after the next **`part=`** parameter, as in the following task:

```
query name=burial;part=:surname
index part=:surname;substitution="--||--";
   part=:firstname;substitution="==//==";
   part=:title;
   part=:occupation
stop
```

the result would be as follows:

```
Budd Alice     1_bur
--||-- William Mr alderman 1_bur
--||-- ==//== Mr alderman 1_bur
```

This is because we have asked κλειω to substitute the string "==//==" where any first names are the same as the previous entry. This works even if the first item in an index is not subject to a **`substitution=`** parameter.

```
query name=burial;part=:surname
index part=:surname;
   part=:firstname;substitution="==//==";
   part=:title;
   part=:occupation
stop
```

```
Budd Alice      1_bur
Budd William Mr alderman 1_bur
Budd ==//== Mr alderman 1_bur
```

Here we have only asked κλειω to replace the first names with the character string "==//==", so only where these match does κλειω reproduce that character string. If however we were to change the data so that Elizabeth Bunckley and her father, the following two people in the index, were both to have the first name William, the following output would be the result of the task. Notice how, even though the last four people in this list all have the first name of William, the string "==//==" is repeated. This is because the **`substitution=`** parameter only operates if *all* the previous elements in the index are the same.

```
Budd Alice      1_bur
Budd William Mr alderman 1_bur
Budd ==//== Mr alderman 1_bur
Bunckley William      1_bur
Bunckley ==//== Mr   1_bur
```

Obviously you will not get the same result as this unless you alter the data in the data file and recompile the database.

5.3.8 The `cumulate=` parameter

This parameter only works in conjunction with the **`substitution=`** parameter (or following another **`cumulate=`** parameter, which must in turn follow a **`substitution=`** parameter). Its purpose is to repeat the effect of the **`substitution=`** parameter. If the following example is run, the entries for surname *and* firstname, if they are the same, will be replaced by the character string previously specified in the **`substitution=`** parameter.

Example 5.11

```
query name=burial;part=:surname
index part=:surname;substitution="---//---";cumulate=yes;
    part=:firstname;
    part=:title;
    part=:occupation
stop
```

Part of the result of this task should look like this:

```
Budd Alice     1_bur
---//--- William Mr alderman 1_bur
---//--- Mr alderman 1_bur
```

In this case the last two items displayed both include the surname Budd and the first name William. If we were to add further **cumulate=** parameter, as below:

```
query name=burial;part=:surname
index part=:surname;substitution="---//---";cumulate=yes;
  part=:firstname;cumulate=yes;
  part=:title;
  part=:occupation
stop
```

part of the result would look like this.

```
Budd Alice     1_bur
---//--- William Mr alderman 1_bur
---//--- alderman 1_bur
```

Notice that in this case the single user-defined string would replace all items that were identical. One further addition of the **cumulate=** parameter should make this absolutely clear.

```
query name=burial;part=:surname
index part=:surname;substitution="---//---";cumulate=yes;
  part=:firstname;cumulate=yes;
  part=:title;cumulate=yes;
  part=:occupation
stop
```

The following result would appear.

```
Budd Alice     1_bur
---//--- William Mr alderman 1_bur
---//--- 1_bur
```

5.3.9 The `limit=` parameter

`limit=` is another useful parameter to help redefine output. If you want to display output in a contextual form, this parameter can be used. The value given with it is then printed at the point where it occurs. The parameter must be followed by a character string, i.e. as usual, it must be enclosed within inverted commas.

To print information from the baptismal database in the following form:

<surname>, *<firstname>* the *<relation>* of *<title>* *<surname>*

the parameter **`limit=`** is used as follows:

Exercise 5.12

```
query name=baptism;part=p
index part=:surname;limit=", ";
   part=:firstname;limit=" the ";
   part=:relation;limit=" of ";
   part=relp:title;
   part=:surname
stop
```

Run this exercise.

There is a vital new piece of syntax introduced into this task, showing how one can manœuvre between two different groups in the database. This will be explained in the following section.

5.4 Moving between groups in a database

The last example demonstrated the simplest way of moving between two different groups in a database. When one specifies a particular group in a **`part=`** parameter following a **`query`** command, κλειω operates as though it can only access data from that particular group. In order to produce an index of all those people who were baptised and the name of the person related to them, one has to move from one group to another – in this case from p to relp. To move down one level in the hierarchy all one needs to do is to specify the name of the group in a subsequent **`part=`** parameter.

However, if one wanted κλειω to produce a result in the following format:

<surname>, *<firstname>* the *<relation>* of *<title>* *<firstname>* *<surname>* was baptised on *<bapdat>*

one would encounter a problem, as the date of baptism of the individual is not part of the group relp.

If one added a **`limit=`** parameter after the **`part=`** parameter in the last example, thus:

```
part=:surname;limit=" was baptised on";
```

and then added the following line:

```
part=p:bapdat
```

κλειω would not understand the task because the group p is *not subordinate* to the last group that was encountered. These changes have been made to the following exercise. Run it to see that it will not work.

Example 5.13 (This will not work)

```
query name=baptism;part=p
index part=:surname;limit=", ";
   part=:firstname; limit=" the ";
   part=:relation;limit=" of ";
   part=relp:title;
   part=:surname;limit=" was baptised on ";
   part=:firstname;
   part=p:bapdat
stop
```

In order to understand why this will not work we must consider the structure of the database. In the first line of the task we asked for only those people who were in the group p; in the fourth **`part=`** parameter within the **`index`** command we moved down one level of the hierarchy to the group relp.

The structure of the database looks like this:

```
doc
 |
 p
 |
relp
```

If we move down the hierarchy of the database from p to relp we can not simply move back up to p again. This is vitally important to remember, because it is perfectly permissible to have two groups with the same name in a hierarchy; for example:

```
x
|
y
|
x
```

Obviously κλειω would not know which of the two groups called x to look at in order to access the information. In order to tell κλειω that we wish to move up the hierarchy we could use one of two different group functions. The one we will be using in this example is **`back[]`**.

5.4.1 The group function `back[]`

In the following example the number 1 is enclosed in the square brackets after **`back[]`**. This denotes to κλειω that we want to move *back* one level in the hierarchy from the group is is currently processing. If we wanted to move back two levels in the hierarchy we would enclose the number 2 within the square brackets, etc.

Example 5.14

```
query name=baptism;part=p
index part=:surname;limit=", ";
   part=:firstname; limit=" the ";
   part=:relation;limit=" of ";
   part=relp:title;
   part=:firstname;
   part=:surname;limit=" was baptised on ";
   part=back[1]:bapdat
stop
```

As an alternative to using figures to denote which level of the hierarchy to which one wants to move back, one could enclose in the square brackets in **`back[]`** the name of the group that one wished to access. In this case the following alternative example would be acceptable:

Example 5.15

```
query name=baptism;part=p
index part=:surname;limit=", ";
   part=:firstname; limit=" the ";
   part=:relation;limit=" of ";
   part=relp:title;
   part=:firstname;signs=1;
   part=:surname;limit=" was baptised on ";
   part=back[p]:bapdat
stop
```

Generally, though, it is recommended to use the first solution as there may be two groups above the group κλειω is currently accessing with the same name – which κλειω would not understand. Moreover, if one had a data structure like the following:

```
    person
      |
    person
      |
relatedperson
```

if κλειω was accessing the group relatedperson and **`back[person]`** was asked for, κλειω would only move back to the first group called person it encountered. (We would suggest that you do not try using structures like this until you know more about κλειω.)

Exercises

Exercise 5.1

Using the database burial, select in alphabetical order the surname, first names and occupations of all those people whose occupations contain the string "organist". Ensure that this brings up only two entries. (Remember that the function **`:query[]`** refers to whole of the previous command, *not* the previous segment of a command.)

Exercise 5.2

Produce a task which retrieves all occurrences of people whose surname is Love and who are recorded as being baptised. Link to this task a second task which retrieves all occurrences of people whose surname is Love and who are recorded as being buried.

Exercise 5.3

The baptism database includes the element bapdat which gives the date of baptism of the children. In this source there are two possible pairs of twins. Use κλειω to find out their names. When you have found them, create an index which includes their surnames, first names, and date of baptism (in that order), and which is sorted by date of birth. Note that this database has been set up in such a way that κλειω only understands dates in the format:

```
   14 Dec 1614
```

Note also that within a condition dates must be enclosed within inverted commas.

Exercise 5.4

Amend the previous task to produce an index which has the first names sorted in reverse alphabetical order. Note that this task will need two of the parameters introduced in this chapter.

Your result should look like this.

```
Ridley Thomas 1600.09.24 1
Alexander Robert 1624.03.03 1
Alexander Frances 1624.03.03 1
Ridley Elizabeth 1600.09.24 1
```

Exercise 5.5

Using the database burial, produce an index of all the women who died who are listed as "wife". The output should be sorted by their surname and the format should be as follows:

<*firstname*> <*surname*> the wife of <*title*> <*firstname*> <*surname*>

Exercise 5.6

Using the database baptism produce an index in the following format:

<*relp:firstname*> <*relp:surname*>'s <*p:relation*> <*p:firstname*> was baptised on <*p:bapdat*>

This should be sorted first by surname and then by date of baptism.

Exercise 5.7

Using the database censsamp, produce an alphabetical index (by surname) of all the related people in the following format:

<*relp:firstname*> <*relp:surname*> the <*relp:relation*> of <*p:firstname*> <*p:firstname*> lived
in <*house:address*> in <*address:town*>

5.5 Elementary counting functions

κλειω has two elementary counting functions. One allows you to count the items that you have specified in a specification command, the other provides basic statistical information about the data in the database.

5.5.1 The `type=` parameter in an index command

When the parameter **`type=`** is followed by the keyword **`count`** after a **`part=`** parameter in an **`index`** command, it means that the data will be output in numerical form. Essentially it means that the number of occurrences of any specified parameter will be counted.

In the case of the following example, κλειω will count the number of occurrences of each surname:

Example 5.16

```
query name=baptism;part=:surname
index part=:surname;type=count
stop
```

Part of the result of this task is shown below:

```
Alexander appears 24 times.
Barlow appears 12 times.
Brown appears 2 times.
Browne appears 6 times.
Butler appears 2 times.
Colson appears 2 times.
Cradock appears 2 times.
Darrell appears 10 times.
Foell appears 4 times.
```

5.5.2 The `cumulate` command

The **`cumulate`** command provides simple statistics. It only works on data of the type **`number`**. It provides the value of the minimum value of a specified entry, the maximum value of that entry, the median value of all the chosen entries, the standard deviation of all the chosen entries and the sum of all the chosen entries.

The following task would give these simple statistics for the age of women under 21 in the censsamp database created in the last chapter:

Example 5.17

```
query name=censsamp;part=:status="f" and :age="21" less
cumulate part=:age
stop
```

The results of this task are shown below:

```
Minimum value:       0.583333
Maximum value:       18.000000
Mean value    :      10.895833
Standard deviation:  7.883814
Number of values:    4.000000
Total:               43.583332
```

For the actions of this command when encountering interval numbers, see the Reference Manual, Section 8.3.5.3.

Before we quiz you on these parameters we will explain some further built-in functions.

5.6 More element functions

5.6.1 The `:day[]`, `:month[]`, and `:year[]` functions

These three functions all take the following format:

```
:year[element specification]
```

where "element specification" refers to either an element or a path to an element. The three functions all let you use part of a date in an element defined as of data type **`date`**. For example if we wanted to know the names of all the people in the baptism database who were born in the month of May we might set the following task:

Example 5.18

```
query name=baptism;part=:month[:bapdat]="5"
index part=:surname;
  part=:firstname;
  part=:bapdat
stop
```

Notice that the square brackets of the function contains the name of the element in the database baptism that contains data of type **`date`**.

Similarly we could ask for information about all people born in or before 1620.

Example 5.19

```
query name=baptism;part=:year[:bapdat]="1620" less equal
index part=:surname;
  part=:firstname
  part=:bapdat
stop
```

5.6.2 Tagging; the `:status[]` function

κλειω allows the user to change the *status* of an entry. Either or both of the tagging characters ("!") and ("?") can be used to denote whether the entry to which the character is added is deemed ʻpeculiarʼ or ʻuncertainʼ in some way. Within historical computing this may be very important as there are frequently times when historians are unable to decide on the validity of a piece of information in the source they are using.

The use of these characters within an entry affects the way in which κλειω produces output. For instance, if during the transcription of this census data we had been unable to read the piece of information about the occupation of Catherine Doyle, we might have recorded this difficulty in the database as follows:

```
relp$fs/Catherine//Doyle/servant/24/Barmaid#illegible
  /country=Ireland
```

This uses the ʻcommentʼ data signal which we met in Section 4.3.7. We could also have transcribed it thus:

```
relp$fs/Catherine//Doyle/servant/24/Barmaid?/country=Ireland
```

If we were to query the database with an entry of this kind, for example:

```
query name=censsamp;part=:occupation="Barmaid"
write
stop
```

a result like this would be produced:

```
reference (1 = "RG11-5490"): relp (5 = "rel-5")
        age             24.000000
        status          unmarried,female
        firstname       Catherine
        surname         Doyle
        occupation      Barmaid? (?)
        relation        servant
        country         Ireland
```

By default, κλειω includes the tag ("?") with the entry. This causes the entry to be put into the database as **`Barmaid?`**. The second question mark, enclosed in brackets, denotes that the status of this entry is a question mark. κλειω allows the user to tell it *not* to include the tag in the core information of the entry by adding a logical object to the .mod file (and adding a new **`element`** command). If it were added to the censsamp database, this logical object might look as follows:

```
item name=dummy;usage=text
signs signs=no
exit name=dummy
```

The new **`element`** command might look like this:

```
element name=occupation;type=text;text=dummy
```

If the logical object of the **`element`** command were added, the task shown earlier in this section would produce this result:

```
reference (1 = "RG11-5490"): relp (5 = "rel-5")
        age             24.000000
        status          unmarried,female
        firstname       Catherine
        surname         Doyle
        occupation      Barmaid?
        relation        servant
        country         Ireland
```

These two tagging characters are important not only because you can add a sort of comment, but because using tools described in Chapter 15 you can search databases ignoring data marked by these characters.

If you wanted to search through a database for all those occupational entries which contained a tagging character, you could use the **`:status[]`** function. The following task:

```
query name=censsamp;part=:status[:occupation]="(?)"
index part=:occupation;
  part=:status[:occupation]
stop
```

would produce the following result (if the database had been altered as on the previous page):

```
Barmaid (?) RG11-5490
```

Obviously if the logical object **`element`** command had not been added, the result would be:

```
Barmaid ? (?) RG11-5490
```

5.6.3 The `:original[]` and `:comment[]` functions

There are two different element functions, **`:original[]`** and **`:comment[]`**, which allow the user to search on information in either the original or comment aspects of an element (described in Section 4.3.7).

In the censsamp database there are no comments, only material which was in the original source. We would not usually want to process this material. This task:

Example 5.20

```
query name=censsamp;part=:occupation
write part=:original[:occupation]
stop
```

produces a result like this:

```
master, employing 2 men
apprentice
out of employ
```

The element function **`:comment[]`** works in exactly the same way. We have already come across one element function **`:each[]`** in Chapter 2, but the six element functions we have seen in the second part of this chapter all work in a different way to **`:each[]`**. **`:each[]`** only ever appears in exactly that form, and is defined in the manual as the set of all elements contained in the last group to be activated. The square brackets at the end of **`:each[]`** are never filled. However, in the case of many element functions (and there are currently forty-nine different element functions) the brackets at the end of the function are filled with the name of an element (or an element function), to which the element function is to refer. More precisely, the contents of the brackets of most element functions contain what is known as an element specification, which may be the name of an element, but may also be another element function.

To demonstrate the last point, let us alter the original data that went into the censsamp database, thus:

```
head$mz/Lambert//Newton/39
   /Grocer%master,employing 2 men?/Wigton/Cumberland
```

Here the tagging character ("?") has been added to the original aspect of the occupation element. This is perfectly legitimate, as the original aspect of an element is treated as an entry.

The following task asks κλειω to display information relating to the original material relating to the occupation of an individual. This task asks κλειω to produce the status of the original aspect of the element occupation and the contents of that original aspect, for all parts of the database where there is an occupation:

```
query name=censsamp;part=:occupation
write part=:status[:original[:occupation]],
   :original[:occupation]
stop
```

This produces the following result:

```
(?)
master, employing 2 men(?)

apprentice

out of employ
```

where the first line and the second are related to the same element. Many of the other element functions will be referred to later in this tutorial.

5.6.4 The `:collect[]` function

The element function **`:collect[]`** is useful as it allows one to 'collect' together all the entries of an element. If the following task is run:

Example 5.21

```
query name=burial;part=:surname="Colson"
index part=:surname;
   part=:firstname;
   part=:occupation
stop
```

it will produce a result like this:

```
Colson Richard clerke p-6

Colson Richard petty cannon p-6
```

because the original data in the database looks like this:

```
p$p-6/dm/Mr/Richard/Colson///clerke;petty cannon//20 Jan 1619
```

In order to 'collect' together these two separate entries within the occupation element, the `:collect[]` function can be used. An element specification must occur within the square brackets following collect, as in the following task:

Example 5.22

```
query name=burial;part=:surname="Colson"
index part=:surname;
   part=:firstname;
   part=:collect[:occupation]
stop
```

The result of this task is shown below:

```
Colson Richard clerke;petty cannon p-6
```

Exercise 5.8

Using these new commands, find out in which month boys were more likely to be born and in which month girls were more likely to be born. Your answer could take a number of different forms. The answer in the appendix is only one that might help to solve this type of problem.

Chapter 6

Logical objects and knowledge bases

6.1 Introduction

Logical objects are the third component of any κλειω database, the data and the structure definitions being the other two. Logical objects tell κλειω how to interpret a particular type or piece of data. It is usual for each logical object defined by the user to refer to just one element within a database, but it is perfectly possible for one to refer to a number of different elements. A number of independent logical objects exist whenever κλειω is activated. This creates a *logical environment*. When one makes a new database this logical environment is stored permanently in one's database. κλειω also allows one to create one's own logical objects, which can, but need not, be integrated into the permanently stored knowledge of a database. Every logical object in either part of the logical environment can be redefined, modified, or deleted.

Logical objects can be defined in a number of ways. The most usual is with an **`item`** command, but they can also be defined using other commands. We will be discussing the creation of a codebook, a particular kind of logical object, using the **`create`** command in Chapter 10. Logical objects can be deleted using a **`delete`** command and described using the **`describe`** command.

6.2 Basic information about logical objects

In order to explain the logical environment we shall demonstrate the use of the **`describe`** command. In Chapter 3 we said that κλειω allows the user to define how κλειω interprets dates in a historical source. We said that by default κλειω accepts dates in the format "Day.Month.Year", only allowing numerical information, and that all years are supposed to start on 1 January. κλειω will let us know at any time while we are using it what logical object(s) are in force at the moment. We can achieve this using the **`describe`** command, thus:

Example 6.1

```
describe name=date;usage=date
stop
```

This is a task and as such should be placed in a file and run in the usual fashion. It means that we would like κλειω to describe a logical object called date which has been defined as of type **`date`**. This task produces the following result:

```
Represents a currently local declaration.
Name of logical unit: date
Class of logical unit: Date

Style      Order               Beginning of year      Names of months

Default    Day  Month  Year    Circumcision           Ordinal numbers
```

Note that the feast of Circumcision falls on 1 January.

This description of a logical object does not refer to any particular database; it is part of the permanent logical environment. This is how κλειω treats dates unless it has been specifically asked to treat them differently.

To make things slightly more complicated κλειω also has a logical object called date which is of data type **`number`**. To see the description of it, run the following task:

Example 6.2

```
describe name=date;usage=number
stop
```

This produces a longer result, so we have not reproduced it here. To summarize the result, however: it says that this is a local declaration, referring to a logical object called date, which is of data type **`number`**. With this type of data we are allowed to use the following operators: "+", "-", "*", ":" (representing the four basic arithmetical operators). We can use

"(" and ")" as logical brackets. Both decimal numbers and triplets (Days, Months and Years) are separated by a ".". κλειω permits spaces between numbers and qualifiers while spaces between fractional values are interpreted as addition operators ("+"). All of the operators have the same status. Words as qualifiers are interpreted as 1 and uncertainty indicators are allowed. By default we are allowed to use the qualifiers "year", "y", "month", "m" "week", "w", "day" and "d". These are given specified weights.

Most of this is rather technical and need not really concern us here. But it should demonstrate why κλειω expects certain data types to be in a certain format.

If a user defines his or her own logical object to be used with a certain element, it overrides κλειω, by telling it to use this logical object rather than the default logical object for that type of data. We have seen this in practice in Chapter 4, while designing the database. In that database we used an **`element`** directive within the structure declaration of the database to tell κλειω that we wanted the element age to be operated on by rules contained in a logical object called ages. This logical object is reproduced here:

```
item name=ages;usage=number
text name="m";number=0.0833333
exit name=ages
```

The task below must be run in the same directory as the compiled censsamp database.

Example 6.3

```
describe name=ages;usage=number;source=censsamp;type=permanent
stop
```

If that task is run, we will get a result that looks rather like the result we obtained earlier, but there would be a slight difference at the end, as follows:

```
List of primary qualifiers

Weighting  Maximum    Minimum    Qualifier

0.083333                         m
```

The first point to mention here is that the task we have run is slightly different, as it refers to the name of a user-defined logical object which is related to the database censsamp. The result shows that we have defined the letter "m" to mean 0.083333 relating to the other numbers in that element. (Thus one month is 0.083333 of a year, even though we have not explicitly defined a year.)

Here we have shown how κλειω lets the user change some of the rules relating to data of **`number`** type. Notice that as we have only changed one part of the default rules we retain all the other rules as originally stated. (Here however we have only changed the way in which κλειω interprets a single element in a database.)

κλειω also allows the user to replace any of the default logical objects with their own logical object, but at the moment we do not recommend that you do this. The rules and regulations of this process can be found in the Reference Manual, Section 7.3.4.

So far we have seen three ways in which logical objects control the way in which κλειω interprets source material:

- default logical objects relating to defined data types within a database
- modified logical objects which keep some of the rules from the default logical objects but with added user-defined rules
- user-defined logical objects which completely override κλειω's default logical objects.

There is a fourth type of logical object. These relate to other types of information within one's database. This type of logical object works in the same way as those which define how κλειω should interpret different data types, but κλειω does not contain default definitions for these rules. The logical objects we have discussed take a keyword in the **`usage=`** parameter which is the same as the name of a data type. These other logical objects take other names because they define something different. A list of all the different types of logical object follow. Some of these will be demonstrated more fully in this chapter, some will be demonstrated in other chapters of this book, while a few will not be demonstrated at all. Full details of how to use all logical objects can be found in Chapter 7 of the Reference Manual.

- **`text`**: logical objects of this class define how κλειω behaves when it is dealing with continuous text, i.e. how it converts such texts into **`text`** type data in the database (for instance, as we have seen in Chapter 5, how κλειω interprets the flags "?" and "!" in the database). This logical object is also used to define to κλειω the symbol(s) to be used to separate entries within elements.
- **`date`**: logical objects of this class define how κλειω should behave when it is dealing with dates, i.e. how it should convert calendar texts into **`date`** type data and process constants (like years) for comparisons with such elements. For example, κλειω allows us to tell it what abbreviations we consider necessary for the months of the year. This class of logical object will be discussed briefly later in this chapter.
- **`number`**: logical objects of this class define how κλειω behaves when it is dealing with numerical data, i.e. how it should convert numerical expressions into **`number`** type data and process constants intended for comparisons with such elements. For an example, see Section 4.5.4 above.

- `category`: logical objects of this class define how κλειω behaves when it is dealing with categorical abbreviations. This has also been demonstrated in Chapter 4.

- `relation`: logical objects of this class define how κλειω behaves when it is dealing with non-hierarchical relationships. This logical object is used to make κλειω work in a similar fashion to a relational database. This will be fully demonstrated in Chapter 13.

- `conversion`: logical objects of this class define systematic changes which κλειω should make to a character string before processing the latter any further. (This kind of logical object is almost always used in conjunction with another.) This will be discussed briefly in Chapter 12.

- `soundex`: logical objects of this class define how κλειω should convert any character strings into a 'phonetic' coded value. A logical object of this class should contain an explicitly specified coding scheme. This logical object will also be discussed in Chapter 12.

- `skeleton`: logical objects of this class define how κλειω should reduce words to a 'framework of essential characters' with a skeletonising algorithm. This logical object will not be discussed in this volume. Consult the Reference Manual, Section 7.3.1.9.

- `guth`: logical objects of this class define how κλειω should use a Guth algorithm to quantify the similarity between two character strings in numerical terms. This class of logical object will not be discussed in this volume. Consult the Reference Manual, Section 7.3.1.10. For information on the Guth algorithm see G. J. A. Guth, 'Surname Spellings and Computerized Record Linkage', *Historical Methods Newsletter*, 10 (1976), pp. 10–19, and D. De Bron & M. Olsen, 'The Guth Algorithm and the Nominal Record Linkage of Multi-Ethnic Populations', *Historical Methods*, 19 (1986), pp. 20–24.

- `substitution`: logical objects of this class tell κλειω how to understand substitution paths (i.e. access paths which κλειω uses if it cannot find an element or group specified by the user in the required location while it is processing a database). This class of logical object will not be discussed in this volume. Consult the Reference Manual, Section 7.3.1.11.

- `codebook`: logical objects of this class are used to set up a 'translation table' which translates sets of character strings into numerical variables. This class of logical object will be more fully discussed in Chapter 10 of this volume.

- **`location`**: logical objects of this class are used to set up and modify directories of topographical coordinates or allocate graphical attributes to topographical objects used in geographical representation of maps. This class of logical object will be discussed more fully in Chapter 14 of this volume.

- **`order`**: logical objects of this class are used to tell κλειω about sorting sequences (for example, allowing the user to sort alphabetically telling κλειω that the letters V and U are to be considered identical). This class of logical object will not be discussed in this volume. Consult the Reference Manual, Section 7.3.1.14. There is a default logical object of class order. To see the order in which κλειω sorts by default run the following task.

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

- **`catalogue`**: logical objects of this class are used to process catalogues that have already been created. This class of logical object will be discussed more fully in Chapter 8 of this volume.

- **`connection`**: logical objects of this type 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 to be processed. This class of logical object will not be discussed in this volume. Consult the Reference Manual, Section 7.3.1.16.

- **`image`**: logical objects of this type are used to tell κλειω how to set up a linkage with a database of digitally stored images. For further information consult the Reference Manual, Section 7.3.1.17, and G. Jaritz, *Images. A Primer of Computer-Supported Analysis with* κλειω *IAS*, Halbgraue Reihe zur historischen Fach-informatik, A22 (St. Katharinen, 1993).

- **`classification`**: logical objects of this type are used to provide embedded classifications within a text administered within a κλειω database. If you consider using full-text and want either to add linguistic knowledge to this text or to apply a hierarchical structure to your text you should consult the Reference Manual, Section 7.3.1.19. The **`classification`** definition was added to the system too late for us to integrate it more fully into this volume.

- **`chronology`**: logical objects of this class are used, for example, to define a calendar of saints' days, i.e. to define the rules to deal with references to a locally prevailing calendar. Logical objects of this class will be discussed later in this chapter.

Even though you may feel that you may never want to use the chronology class of logical object we suggest very strongly that you work through that section of this chapter as it gives valuable information about using logical objects generally.

6.3 Logical objects; calendars

To create a logical object which defines how κλειω interprets 'non-standard' dates in the database is relatively easy. There are however a number of interpretational problems which we hope to make clear in this section.

We consider that the best way in which to explain how κλειω understands calendar dates is to create a dummy database which contains all the ingredients necessary for κλειω to process saints' days and religious festivals. The database that we will be using here is provided on the sample disk (dummy.mod), but we would suggest that you create files which you will be able to use with this chapter. These should be based on the files shown below.

The database we shall use is very simple. Its two basic elements are shown below:

```
database name=dummy;first=festival;overwrite=yes
part name=festival;position=id,date
exit name=dummy

read name=dummy
festival$1/Lady Day 1613
festival$2/Quadragesima 1613
festival$3/Maundy Thursday 1613
festival$4/Advent Sunday 1613
festival$5/3 days before Easter 1613
festival$6/Week after Pentecost 1994
```

We have included six dates which could occur in a historical source. We shall construct a logical object of type **`chronology`** to let κλειω interpret these dates. The first stage in this process is to tell κλειω that the element date in this database should be of type **`date`**. This line would usually be added after the **`database`** command and before the first **`part`** directive, thus:

```
database name=dummy;first=festival;overwrite=yes
element name=date;type=date;date=tutorial
part name=festival;position=id,date
exit name=dummy
```

This **`element`** directive explains to κλειω that there is an element in this database called date, which is of data type **`date`**, which refers to a logical object called tutorial (which is a user-defined name).

This logical object needs to be constructed according to the rules laid down for **`date`** data type rather than religious festival or saints' days.

A logical object is normally placed above the structure declaration of a database. The following logical object would be appropriate for this database:

```
    item name=tutorial;usage=date
    type ...
    exit name=tutorial
```

Logical objects that refer to dates must contain a **`type`** directive, which gives κλειω information about the type of date that it is to process. As we want κλειω to be able to process dates that refer to feast days, we must define that requirement to κλειω. The parameter necessary to achieve this is **`name=`** followed by the keyword **`saints`**. Whenever the **`saints`** parameter is used, there must also be a **`date=`** parameter in this **`type`** directive. The **`date=`** parameter should contain the name of a chronology definition as a value. This tells κλειω which definition to use to process the contents of this element. We have given this (as yet undefined logical object) the name english.

```
    type name=saints;date=english
```

We are now ready to construct this logical object.

If you are creating these files as you go you will need at this point to save the .mod file and open a new file which for simplicity we shall call dummy.cal (as it contains the calendar definition). Do not compile this file yet as you will receive error messages telling you that κλειω is unable to access the logical object english – which is reasonable as we have not created it yet!

We can start to define this logical object by giving it the following heading and **`exit`** commands:

```
    item name=english;usage=chronology;source=dummy;overwrite=yes
    ...
    exit name=english
```

This tells κλειω that we are about to construct a logical object called english which is of class **`chronology`**, and that we would like it to be overwritten whenever we make any changes.

Remember, all definitions of logical objects must begin with an **`item`** command and end with an **`exit`** command.

The **`name`** directive is the only important directive for use with a **`chronology`** definition. It introduces a number of parameters, the first of which must be **`name=`**. This takes as a parameter value a character string that we want κλειω to interpret as a component of a date. Another parameter used with the **`name=`** parameter is **`date=`**. This takes the value of the date in days and months. For instance, if we wanted to define a fixed feast day, Lady Day, as 25 March, we would define it as follows:

```
name name="Lady Day";date=25.3
```

Dates can also be defined as having a relationship with Easter. This is valuable as κλειω knows about the Paschal calendar and automatically calculates the date of Easter in any year. To relate a date to Easter, the **`name=`** parameter is needed together with a parameter **`easter=`** which takes as a value a number of days before or after Easter Sunday. For example:

```
name name="Quadragesima";easter=-42
name name="Easter";easter=0
name name="Pentecost";easter=49
```

For this particular database we will need to add all three of these lines as to understand the entry "the week after Pentecost" we should need to define when Pentecost is. We must also define "Easter" in a **`name=`** parameter of a **`name`** directive, as above, since the date of Maundy Thursday is dependent on the date of Easter.

Notice that feast days which are related to Easter Sunday, but which occur before it, need to be allocated a negative number of days (i.e. a number of days before Easter Sunday).

Any number of dates which are related to Easter Day can be defined in this way.

Moveable feast days are slightly more difficult to define, and are defined as follows.

In the next example we shall demonstrate how to define Maundy Thursday. This day is usually defined as the first Thursday before Easter Day; it could also be defined as the day before Good Friday. In order to put this into a form that κλειω will understand, let us start by defining Good Friday:

```
name name="Good Friday";form="Friday before Easter"
```

and then define Maundy Thursday:

```
name name="Maundy Thursday";form="day before Good Friday"
```

The **`form=`** parameter must contain terms that are also defined in the logical object. Whenever κλειω comes across terms in the **`name=`** parameter it converts them into the value of the **`form=`** parameter and then interprets this with other definitions in the logical object. In this case we have introduced three terms that we must define elsewhere. The following three **`name`** directives are incomplete, but it should be clear that they represent the missing terms from this logical object:

```
name name="Friday";
name name="day"
name name="before";
```

If we take the last directive, we can see that κλειω allows one to define a date by its proximity to another date. κλειω uses one of two parameters to define a date that occurs either before or after another date. These parameters are **`before=`** and **`after=`**. They almost always take the value **`exclusive`**. (The other possible parameter value is **`inclusive`** which should be used when defining periods of time: see below.)

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

κλειω also assumes that you have defined days of the week. Days of the week must be followed by a **`weekday=`** parameter which should contain a value representing one of the days of the week. (This parameter also accepts the values **`absolute`**, **`octave`** and **`previous`**. The first tells κλειω that this date refers to a day, the second to a day 7 days after a feast, and the third to the day before a feast.) In this case we have to define Sunday as a weekday which is a **`sunday`** and say that a day can be any day of the week:

```
name name="Friday";weekday=friday
name name="day";weekday=absolute
```

It should be clear that as long as all the terms that are contained in the original data can be defined for κλειω to understand them, κλειω is able to process them. For instance, if the term "Sunday after All Saints' Day" occurred in some original data, all that would be necessary for κλειω to understand it would be the following lines in a **`chronology`** logical object:

```
name name="Sunday";weekday=sunday
name name="after";after=exclusive
name name="All Saints' Day";date=1.11
```

κλειω also copes with periods of time. According to the rules of κλειω every date in a database may potentially be a *terminus ante quem* or a *terminus post quem*. κλειω does not yet support period dates between two dates that need to be interpreted, such as "from All Saints' Day 1623 to Christmas Day 1623". It can, however, cope with dates in the format "Week after Pentecost 1567". This is achieved by using **`first=`** and **`second=`** parameters after a **`name`** command. In this case κλειω would need to know the following information:

```
name name="week after";first="1 day after *";
  second="7 days after *"
name name="Pentecost";easter=49
name name="day";weekday=absolute
name name="days";weekday=absolute
name name="after";after=inclusive
```

There are three points to mention here.

- First, we should describe the **`first=`** and **`second=`** parameters. These two parameters take for any date a *terminus ante quem* and a *terminus post quem*. They use a similar format to the **`form=`** parameter but also take an asterisk ("*") to denote the date. In the above example, then, the phrase "week after" means from one day after any date to seven days after that same date.

- Second, we should also alter the dates allowed by κλειω; by default κλειω only accepts dates between 1.1.1500 and 1.1.1994. In order to change this we are allowed to use the **`date`** command. This **`date`** command should be placed above the structure declaration of the database, i.e. before the **`database`** command.

- Third, the **`after=`** parameter has been given the keyword **`inclusive`**, which in this case tells κλειω that "the seventh day after" a feast day should also be displayed in the output.

6.3.1 The `date` command

If you wanted to use data that pertained to the time scale 1200 to 1994, you would use the date parameter with the following parameters.

```
date minimum="1.1.1200";maximum="1.1.1995"
```

κλειω also allows for the use of both the Julian and Gregorian calendars. By default κλειω understands the date for the introduction of the Gregorian calendar as 15 October 1582. However as not all countries followed the lead of the Papal States, Spain and Portugal, κλειω allows the user to set their own date for the introduction of the Gregorian calendar. For Great Britain and Ireland this would be shown:

```
date first="14.9.1752"
database name=<name of database>
```

For France it would be as follows:

```
date first="20.12.1582"
database name=<name of database>
```

For Great Britain and Ireland all dates in a database with such a command before 14 September 1752 would be assumed to be in Julian style, and all dates following this date would be considered to be in the Gregorian style (including 14 September 1752).

The abbreviations **OS** and **NS** can be used in the data to override this function. The contraction **NS** can be used to tell κλειω that a date before the introduction of the local Gregorian calendar is in fact in the New Style, while the contraction **OS** can be used to tell κλειω that a date after the introduction of the local Gregorian calendar should in fact be interpreted as a date in the Old Style.

The files you have created should look like this:

```
note dummy.mod
item name=tutorial;usage=date;overwrite=yes
type name=saints;date=english
exit name=tutorial

date maximum="1.1.1995"

database name=dummy;first=festival;overwrite=yes
element name=date;type=date;date=tutorial
part name=festival;position=id,date
exit name=dummy

item name=english;usage=chronology;source=dummy;type=permanent;
  overwrite=yes
name name="Sunday";weekday=sunday
name name="Sundays";weekday=sunday
name name="Friday";weekday=friday
name name="day";weekday=absolute
name name="days";weekday=absolute
name name="before";before=exclusive
name name="after";after=inclusive
name name="Lady Day";date=25.3
name name="Christmas";date=25.12
name name="Quadragesima";easter=-42
name name="Easter";easter=0
name name="Pentecost";easter=49
name name="Adventstart";form="5 sundays before Christmas"
name name="Advent Sunday";form="Sunday after Adventstart"
name name="Good Friday";form="Friday before Easter"
name name="Maundy Thursday";form="day before Good Friday"
name name="week after";first="1 day after *";
  second="7 days after *"
exit name=english

read name=dummy
festival$1/Lady Day 1613
festival$2/Quadragesima 1613
festival$3/Maundy Thursday 1613
festival$4/Advent Sunday 1613
festival$5/3 days before Easter 1613
festival$6/The week after Pentecost 1994
```

Once each of these files have been compiled in the correct order, i.e. dummy.mod, dummy.cal and dummy.dat, a task like the one below can be run to see how κλειω interprets these dates.

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

Part of the result of this task would be as follows:

```
festival (1 = "1")
          date    25.3.1613
          id      1

festival (2 = "2")
          date    12.5.1613
          id      2
```

Chapter 7

More complex database structures

7.1 Introduction to the probate database

For most of the remainder of this book, a different database will be used to demonstrate some of the more interesting and specifically historical functions of κλειω. This database is called probate and can be found on the tutorial disk. We recommend that you compile this database now, while you read this introduction, as the compilation of this database takes rather a long time.

The database is made up of a number of probate inventories and wills from the city of Winchester in the early seventeenth century. They are dated between 1620 and 1626. This is not the place to enlarge upon the importance of probate material for historical research; there is a burgeoning literature on the subject, some of which is given at the end of this chapter.

The importance of probate material is one reason for our choice of database. Another is that it illustrates particularly well why one might want to use κλειω for a complex and rich historical source which needs to be kept intact if one is to be able to do anything useful with it. You will need to acquire a good understanding of the database in order to follow the features of κλειω that are demonstrated later.

The structure of the database

The structure of the database is relatively straightforward:

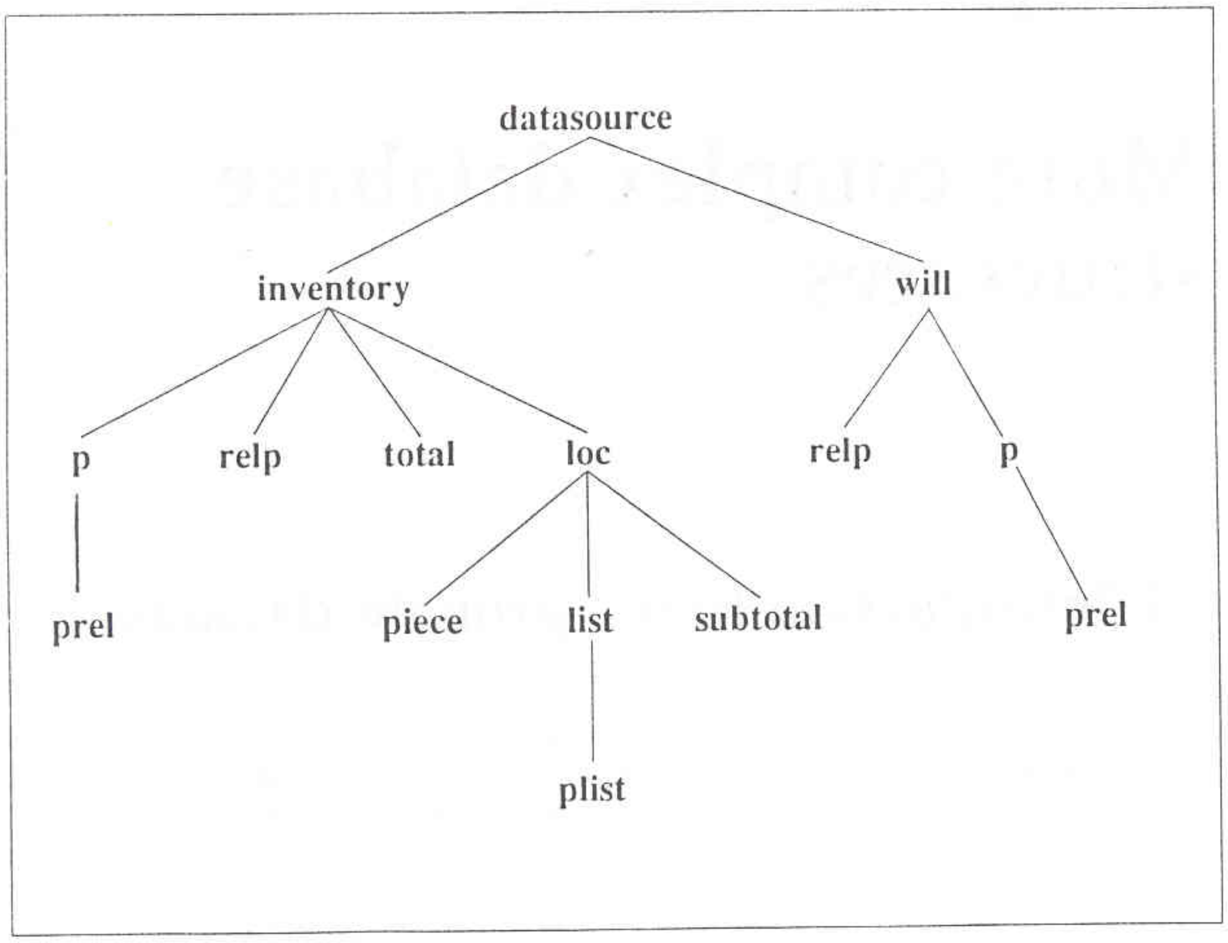


Figure 7.1

The group datasource is made up of one element, refnum, which is an artificial identifier. The groups inventory and will are subordinate to it.

The group inventory is made up of two elements, invref and invnum, which refer to the Hampshire Record Office reference number for the inventory and the date on which the inventory was made. The groups p, relp, total and loc are subordinate to it.

The information group p (subordinate to inventory) contains data about the person who owned the goods in the inventory. The elements within this group are status, firstname, surname, occupation, house (name of), street, parish and town. The group prel is subordinate to it.

The group prel contains data about the ‘relations’ of those people who owned the goods listed in the inventory. Its elements are similar to those for the group p. No groups are subordinate to this group.

The group relp contains data about other people mentioned in the inventory. This only contains the names of the appraisers of the inventory. No groups are subordinate to this group.

The group total contains data given in the inventory as the total worth of the goods and chattels listed in the inventory. Beware of this as it is not always an accurate sum of all the items listed in the inventory. No groups are subordinate to this group.

The group loc contains the physical location of the items given in the inventory. The groups piece, list and subtotal are subordinate to this group.

The group piece refers to the individual items listed in the inventory. The elements contained within this group are item, material, quality, form, colour, quantity and value. item is the name of the property, material refers to what it is made of, quality to the size or age (small, old, little etc.) of the item, form to the mode of production (turned chair, joyned stool) and colour refers to the colour of the object. value contains data about the value of the item. No groups are subordinate to this group.

The group list refers to a group of items with only one value. Often goods are listed in an inventory as a list of items with a single value. The elements contained within this group are the same as those for the group piece except that the element value has been subordinated to a different group. The group plist is subordinate to this group.

The group subtotal contains information about the total given in the original inventory which relates to the total value of the goods and chattels in that particular location.

The group plist contains a single element – value, which refers to the value of the goods in the previous list. No groups are subordinate to this group.

The group will contains the full text of the will of the same person who left an inventory. The groups p and relp subordinate to this group contain similar elements to the groups with the same name subordinate to the group inventory.

Before we start to look at this database there is one parameter within the .mod file of this database that warrants our attention now.

7.1.1 The **`arbitrary=`** parameter

In the .mod file there is one parameter that has not yet been encountered:

```
part name=inventory;
   arbitrary=p,relp,loc,total;
   position=invref,invdat
```

This parameter appears in the same position as a **`part=`** parameter in a **`part`** directive. It performs a similar function to the **`part=`** parameter but it releases the assumption that all the groups listed in that parameter must occur in the same order. In the original inventories these items do not always occur in the same order. In order to prevent κλειω from misinterpreting the data the **`part=`** parameter must be replaced by the **`arbitary=`** parameter.

An example from the database should demonstrate the effects of this parameter. Notice the sequence in which the group names occur and compare this with an extract from the .mod file shown below.

```
inventory$HRO1623A26\/2/20 Jan 1623
relp$ma/William/Marrilyent%Marryland////St Maurice/Winchester
relp$ai/Francis/Smith////St Maurice/Winchester
p$md/Thomas/Hayward////St Maurice/Winchester
loc$chamber over the Hale
list$standine bedsted///Joine worke//1
list$matts
list$cords
plist$13s 4d
```

```
part name=inventory;
   arbitrary=p,relp,loc,total;
   position=invref,invdat
```

Exercises

Using the probate database do the following exercises:

Exercise 7.1

Count the contents of the element material.

Exercise 7.2

Select all objects in the database made out of wood and display all the elements which describe them.

Exercise 7.3

What is the mean value of objects made out of wood?

Exercise 7.4

Count the number of occurrences of the word "old" (or variants)

7.2 Relationships between groups

We have demonstrated two ways of moving between groups, the **`back[]`** function (to move back a determined number of groups) and the use of a subordinate group name (to move forward one level). E.g.:

Example 7.1

```
query name=probate;part=p:surname
write part=prel:surname
stop
```

However, if one wanted to move forward more than one level of the hierarchy , for example from datasource to p, we need to use a forward slash to indicate to κλειω which path to follow through the database. E.g.:

Example 7.2

```
query name=probate;part=datasource
index part=:refnum;
  part=inventory/p:surname
stop
```

The / indicates to κλειω which path to follow.

Similarly we could use it in conjunction with the **back[]** function, thus:

Example 7.3

```
query name=probate;part=relp
index part=:surname;
  part=back[1]/p:surname
stop
```

Using the diagram of the hierarchy (page 130), discover what this does. κλειω is asked first to go to all the groups called relp, then to print out in alphabetical order all the surnames in that group, then to go back one group (to *both* will and inventory), to go forward one group to p, and to display those surnames too.

We could specify that we only wanted to do this for the group will by asking:

Example 7.4

```
query name=probate;part=relp
index part=:surname;
  part=back[will]/p:surname
stop
```

However, this result produces all the surnames of all the people in both of the groups called relp. If we only wanted a list of all those people who were mentioned in the group relp dependent on the group will we might ask:

Example 7.5

```
query name=probate;part=will/relp
index part=:surname;
  part=back[1]/p:surname
stop
```

Exercises on moving between groups can be found in the next chapter.

Further reading

There is a very large literature on probate records. The following literature represents only a small fraction, but includes much written on computing and the analysis of inventories.

Collections of inventories and wills

It may be worth reading the introduction to any one of the following collections.

J. M. Beastall & D. V. Fowkes, 'Chesterfield Wills and Inventories, 1521–1603', *Derbyshire Record Society* (1977).

P. C. D. Brears, 'Yorkshire Probate Inventories, 1542–1689', *Yorkshire Archaeological Society Record Series*, 134 (1972).

J. S. W. Gibson (ed.), 'Banbury Wills and Inventories, 1591–1620', part 1, *Banbury Historical Society*, 13 (1985).

J. A. Johnston, 'Probate Inventories of Lincoln Citizens, 1661–1714', *Lincoln Record Society*, 80 (1991).

M. A. Havinden, 'Household and Farm Inventories in Oxfordshire, 1550–1590', *Oxford Record Society*, 44, and Historical Manuscripts Commission Joint Publication, 10 (1965).

H. C. F. Lansberry, 'Sevenoaks Wills and Inventories in the Reign of Charles II', *Kent Records*, 25 (Maidstone, 1988).

J. S. Moore, *The Goods and Chattels of our Forefathers: Frampton Cotterell and District Probate Inventories, 1539–1804* (Phillimore, Chichester, 1976).

L. Munby, *Life and Death in King's Langley: Wills and Inventories, 1498–1659* (King's Langley Library, King's Langley, 1981).

E. Roberts & K. Parker, 'Southampton Probate Inventories, 1447–1575', *Southampton Records Series*, xxxiv and xxxv (1992).

F. W. Steer, 'Farm and Cottage Inventories of Mid-Essex, 1635–1749', *Essex Record Office Publications*, 8 (Chelmsford, 1950 and revised edition, Phillimore, Chichester, 1965).

D. G. Vaisey, 'Probate Inventories of Litchfield and District, 1568–1680', *Collections for a History of Staffordshire*, fourth series, 5 (1969).

General works on Probate Records

A. van der Woude & A. Schuurman (eds.), 'Probate Inventories. A New Source for the Historical Study of Wealth, Material Culture and Agricultural Development', *AAG Bijdragen*, 23 (1980).

P. Riden (ed.), *Probate Records and the Local Community* (Alan Sutton, Gloucester, 1985).

Computers and probate records

L. G. Carr & L. S. Walsh, 'Inventories and the Analysis of Wealth and Consumption Patterns in St. Mary's County, Maryland, 1658–1777', *Historical Methods*, 13 (1980), pp. 81–104.

T. Grotum & T. Werner, *Sämtlich Hab und Gut ... Die Analyse von Besitzstandslisten*. Halbgraue Reihe zur historischen Fachinformatik, A2 (St. Katharinen, 1989).

A. Hanson Jones, 'Estimating the Wealth of the Living from a Probate Sample', *Journal of Interdisciplinary History*, 13 (1982), pp. 273–300.

T. J. King, 'The Use of Computers for Storing Records in Historical Research', *Historical Methods*, 14 (1981), pp. 59–64.

M. Overton, 'Computer Analysis of an Inconsistent Data Source: the Case of Probate Inventories', *Journal of Historical Geography*, 3 (1977), pp. 317–326.

M. Overton, 'Estimating Crop Yields from Probate Inventories: an Example from East Anglia, 1585-1735', *Journal of Economic History*, 39 (1979), pp. 363–378.

M. Overton, 'English Probate Inventories and the Measurement of Agricultural Change', *AAG Bijdragen*, 23 (1980).

M. Overton, 'Computer Analysis of Probate Inventories: from Portable Micro to Mainframe', in *History and Computing*, eds. P. Denley and D. Hopkin (Manchester University Press, Manchester, 1987).

M. Overton, 'Computer Standardization of Probate Inventories', in *Standardisation et échange des bases de données historiques*, ed. J.-P. Genet (CNRS, Paris, 1988).

B. Pöttler, 'Modelling Historical Data. Probate Inventories as a Source for the History of Everyday Life', in *Storia & Multimedia*, eds. F. Bocchi and P. Denley (Grafis Edizioni, Bologna, forthcoming, 1994).

Chapter 8

Catalogues

8.1 Introduction

A catalogue as used by κλειω can be described as an index of locations, rather like an index to a book. An index to a book contains a number of entries with a reference to the page where those entries can be found. To use a book the reader will use the index to speed up the process of finding out information. One does not read through a whole book in order to find a small piece of information. A catalogue in κλειω performs a similar task. The contents of each (or any) element can be sent to a catalogue, and κλειω will then add to these terms a 'pointer' which directs it to that place in the database. This means that all databases can be accessed much more quickly than sorting through the whole of the data as the location of this information in the database can be found almost instantly.

As an example, consider asking κλειω to search for all the occurrences of a word in all of the wills in the probate database. In order to complete this task, κλειω would have to go through every word in the database to find the particular word. If this database was much larger it would take a considerable amount of time to find that particular word. If on the other hand a catalogue of all the words in all the wills was made first, κλειω would only have to search through the catalogue to find the specified word, and, using the information referring to the location of that word, go immediately to all the places in the database where that word occurs. Thus the use of a catalogue considerably speeds up the querying of a database especially for larger databases.

There is another important aspect to catalogues. The use of catalogues introduces us to the menu-driven part of κλειω. Almost all your use of the menu system will depend on the fact of your having previously created a catalogue or catalogues. This part of κλειω will be touched on in this chapter, and will be consolidated in the following chapter. However, the complete range of tasks that can be performed by the menu-driven system are not all within the scope of this tutorial.

8.2 Creating a catalogue

The creation of a catalogue is an easy process. Consider the following task:

Exercise 8.1

```
query name=probate;part=:surname
index part=:surname
stop
```

This task produces a index of all the surnames found in the probate database for the user to use. We have said that a catalogue is rather like an index, but for κλειω's use. To create a catalogue we need to use a task like this:

Exercise 8.2

query name=probate;part=:surname
I am interested in the database called probate and I am only interested in those groups which contain the element surname.
catalogue part=:surname;name=people
With the contents of those elements called surname create a catalogue. This catalogue should be called people.
stop

The result of this task may be disappointing. κλειω simply tells you that it has created a catalogue without showing it to you. If you wanted to see what κλειω has made you could run a task using the **describe** command:

Exercise 8.3

```
describe name=people;source=probate;usage=catalogue;
   type=permanent
stop
```

Once you have created a catalogue it is a straightforward task to query it. Consider the usual method of querying a database. Say we wanted to know all the details about those people in the database probate with the surname Crosse. If we ran the following task κλειω would search through the database sequentially to find all that information and then display it on the screen.

Example 8.4

```
query name=probate;part=:surname="Crosse"
write part=:each[]
stop
```

If you run this task try timing it.

The next task should produce exactly the same results but use the catalogue to speed up the enquiry process. Try timing this task too.

Example 8.5

```
query name=probate;part=catalogue[people,complete,"Crosse"]
write part=:each[]
stop
```

The three parts of the built-in function **`catalogue[]`** refer to the following information:

```
part=catalogue[people,complete,"Crosse"]
                      /          |         \
   name of catalogue to use   how to look it up   what to look up
```

The keyword **`complete`** here tells κλειω that the constant that follows it should be the whole of the entry within the catalogue. If we had said:

```
part=catalogue[people,start,"Crosse"]
```

κλειω would have produced any entry within the catalogue with the string "Crosse" at the beginning of an entry. Similarly the keyword **`limit`** can also be used in this position to call all the entries which contain the following string at the end of a word.

One can also formulate a task which searches for more than one name at a time. Try running the following task:

Example 8.6

```
query name=probate;
  part=catalogue[people,complete,"'cowlson' or 'crosse'"]
write part=:each[]
stop
```

Using the command language is not the only way to access a catalogue. Now that we have a catalogue in place, it would be worth considering some of the basic features of the menu driven system of κλειω. As we have said, not all of the features of this system will be described in this tutorial.

8.3 Using the menu system

out-of-date

To enter the menu system, you need to type **`kleio`** at the DOS prompt.

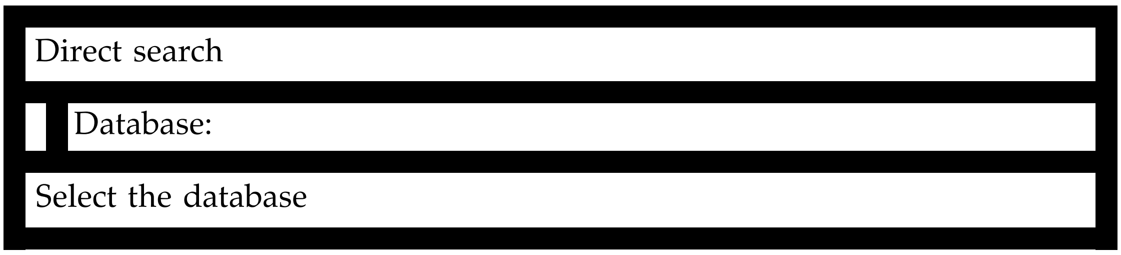
Once you have done this, a menu similar to this should appear on the screen. We say similar for two reasons; first, we cannot reproduce the colour of your screen in this book, and second, we cannot effectively reproduce the highlighted areas on the screen. However, we feel that you should have no difficulty in following what we demonstrate.

```
Kleio Version 5.1.1

  Systematic processing of a database
  Interactive processing of a database
  Creating a new database
```

Using the ↑ and ↓ keys, move the highlighted bar down so that the line **`Interactive Processing of a database`** is highlighted. Once you have done that activate that field by pressing F1. This will bring up another menu, looking similar to this:

```
Interactive processing of databases

  Processing a catalogue interactively
```

The field **`Processing a catalogue interactively`** will be highlighted. You should activate this field by pressing F1. This will bring up another menu, which looks like this:



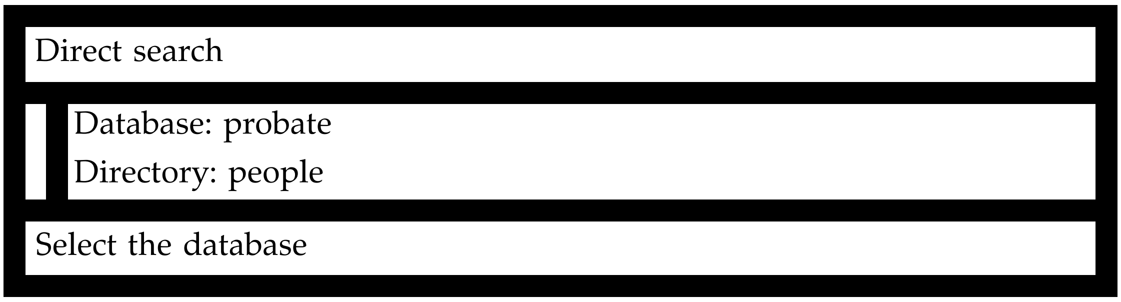
Direct search

Database:

Select the database

The field **`Database`** will be highlighted. The line at the bottom of this menu is a sort of help message; it prompts you to do something. There will always be a line in the bottom of a menu when it is necessary for you to do something other than activate the current field. However, this message will remain on the screen even if you have already made a decision. In this menu, you are prompted to choose a database. In this case, we have made a catalogue with the database called probate. So type the word probate in that field. Then press F1.

κλειω will then produce another menu, looking rather like this:

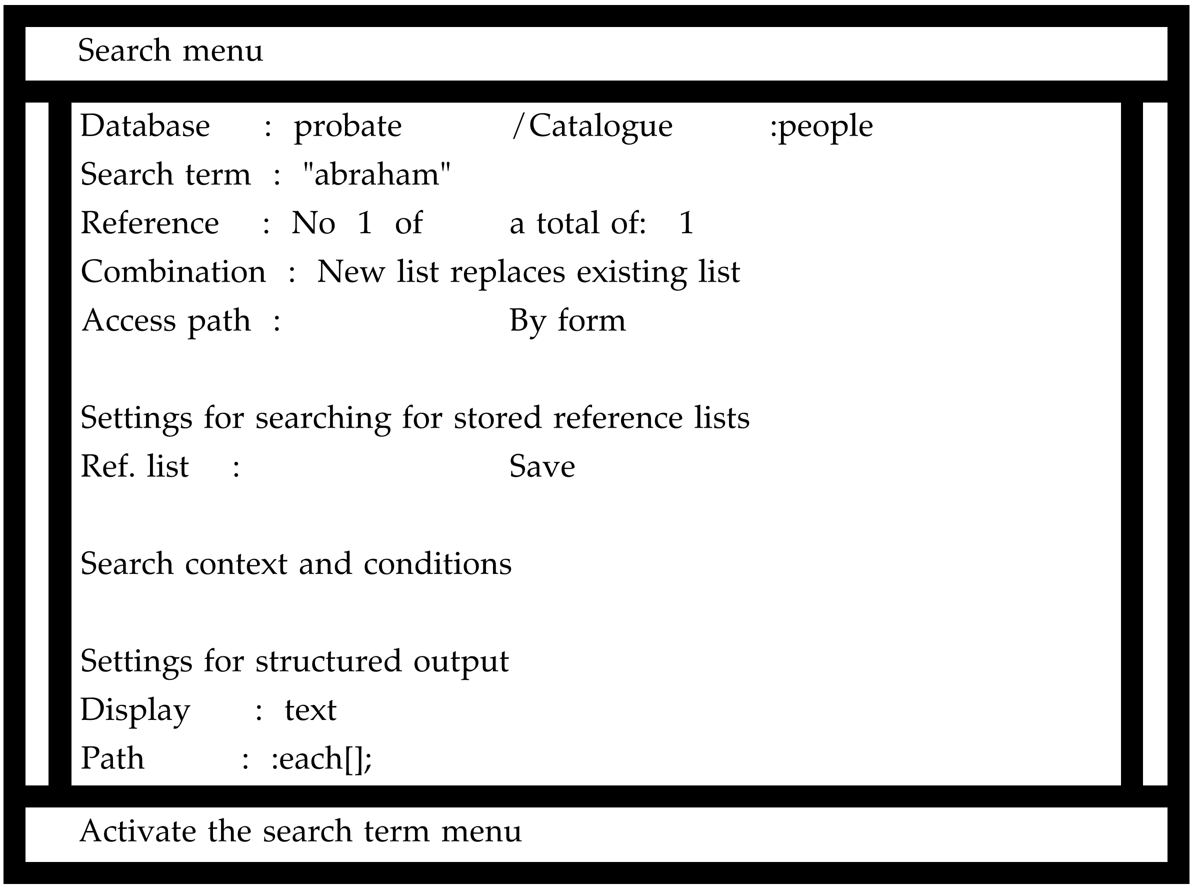


Direct search

Database: probate
Directory: people

Select the database

The highlighted bar will still be in the first field, but there will be an additional field, called **`Directory`**. If you use the ↓ key the message in the help field will change to **`Choose the directory for storing locations`**. Here κλειω is asking for the name of the catalogue that you wish to use. As there is only one catalogue in existence for that database, the entry **`people`** will automatically appear there. Press F1 to activate that menu, and κλειω will produce the following menu:

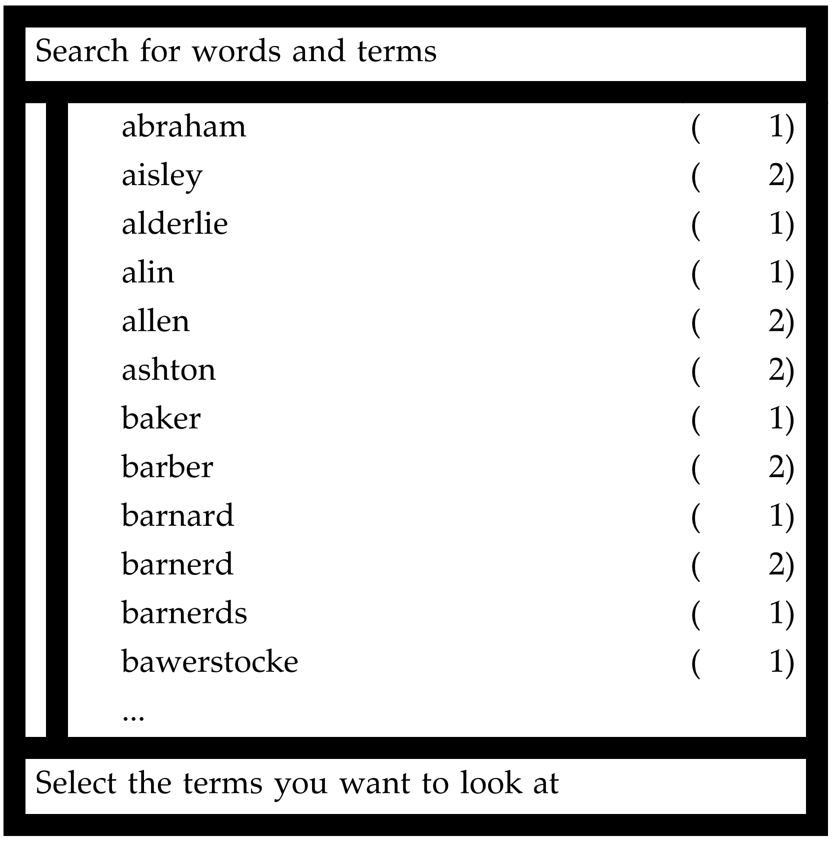


```
Search menu

Database     :  probate          /Catalogue          :people
Search term  :  "abraham"
Reference    :  No  1  of        a total of:   1
Combination  :  New list replaces existing list
Access path  :                   By form

Settings for searching for stored reference lists
Ref. list    :                   Save

Search context and conditions

Settings for structured output
Display      :  text
Path         :  :each[];

Activate the search term menu
```

This menu is usually used for searching full text material, but it is perfectly possible to use it to search through other material within a database. As you enter this menu, the **`Search term`** field should be highlighted. There should also be a term within that field. Look through this menu. As you can see, it is saying that we are using the probate database and the catalogue called people. Within that catalogue there is a term abraham, which occurs once. Ignore the rest of this menu for the time being.

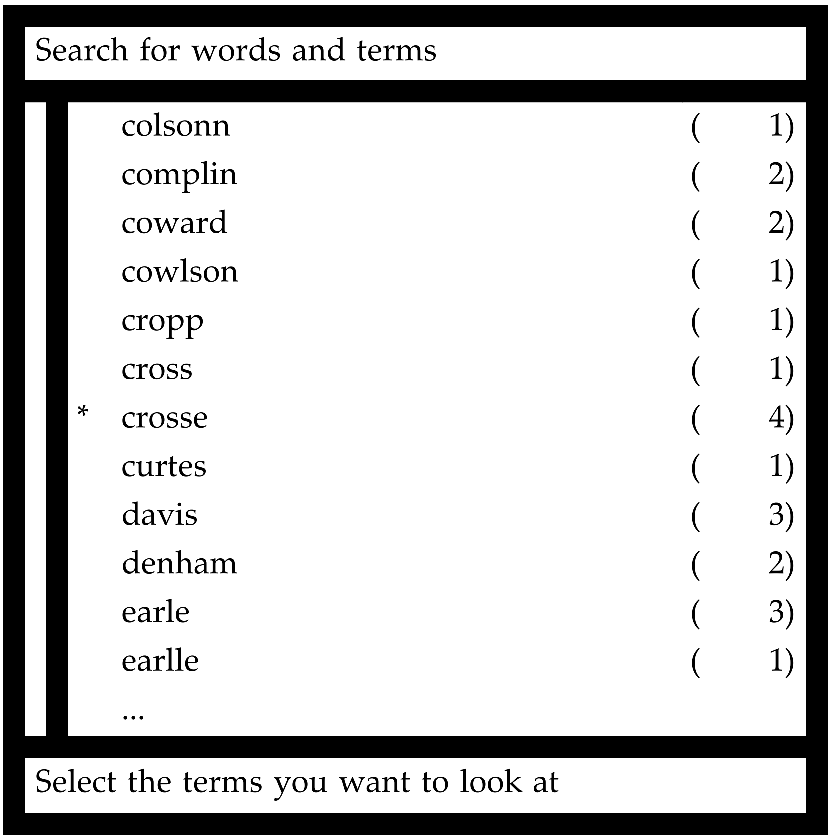
Keeping the highlighted bar on the **`Search Term`**, press either of the ← or → keys and you will be moved to a list of terms contained in the catalogue.

Note that if you press F10 while you are in the menu-driven system you will frequently get some assistance in the role of the function keys at that particular point in your session.



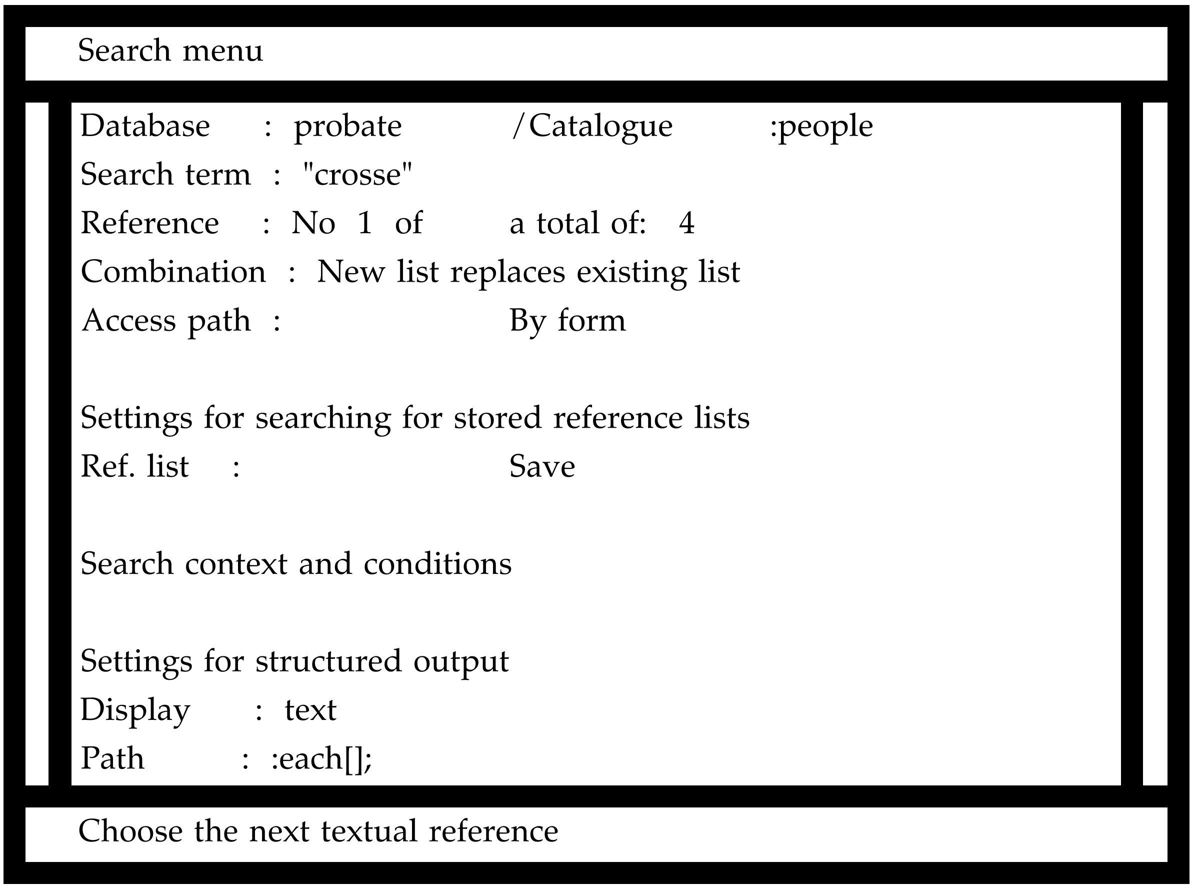
```
Search for words and terms

    abraham                        (   1)
    aisley                         (   2)
    alderlie                       (   1)
    alin                           (   1)
    allen                          (   2)
    ashton                         (   2)
    baker                          (   1)
    barber                         (   2)
    barnard                        (   1)
    barnerd                        (   2)
    barnerds                       (   1)
    bawerstocke                    (   1)
    ...

Select the terms you want to look at
```

This is not the complete menu. To save space a few lines have been left out. This menu, however, contains a list of all the words or terms within the whole of the catalogue, with their frequency on the right hand side. You can move up and down this menu using the ↑ and ↓ keys. You can also move up and down using the PgUp and PgDn keys. A third way of looking through this list is to press the first letter of the word you want, followed by the second. κλειω will move through the list to find the word you are looking for. Move down this list to get to the term Crosse. Then press the Enter key. This should result in an asterisk being placed next to that term. This signifies that that particular term has been ‘marked’. If you accidently mark something, position the highlighted bar on it again and press Enter a further time. This will ‘unmark’ the term.



```
Search for words and terms

     colsonn                        (     1)
     complin                        (     2)
     coward                         (     2)
     cowlson                        (     1)
     cropp                          (     1)
     cross                          (     1)
  *  crosse                         (     4)
     curtes                         (     1)
     davis                          (     3)
     denham                         (     2)
     earle                          (     3)
     earlle                         (     1)
     ...

Select the terms you want to look at
```

Once you have 'marked' the term Crosse with the use of the Enter key, press F1 which should return you to the search menu. This will have changed to look as follows, and the first **`reference`** field will be highlighted:



```
Search menu

Database     :  probate          /Catalogue         :people
Search term  :  "crosse"
Reference    :  No  1  of        a total of:   4
Combination  :  New list replaces existing list
Access path  :                   By form

Settings for searching for stored reference lists
Ref. list    :                   Save

Search context and conditions

Settings for structured output
Display      :  text
Path         :  :each[];

Choose the next textual reference
```

Now that κλειω has some results for you to see, press F1 which will take you to an output screen. This should look something like this:

```
datasource (15 = "15") :
 Crosse
```

Depending on the way in which the system that you are using has been configured, the term Crosse will in some way be highlighted. If you have not changed the configuration (refer to the notes distributed with the software) it will flash.

The information displayed here is not particularly exciting. This is because κλειω is only displaying contents of the element surname where the surname is Crosse. However this does tell us that there is a person called Crosse in the fifteenth document in the database.

To see the result of this task with some context, press F3. This will display slightly more information, namely the whole of the group in which Crosse is the surname:

```
datasource (15 = "15") : relp (1 = "rel-1")
        status          appraiser, male
        firstname       James
        surname         Crosse
```

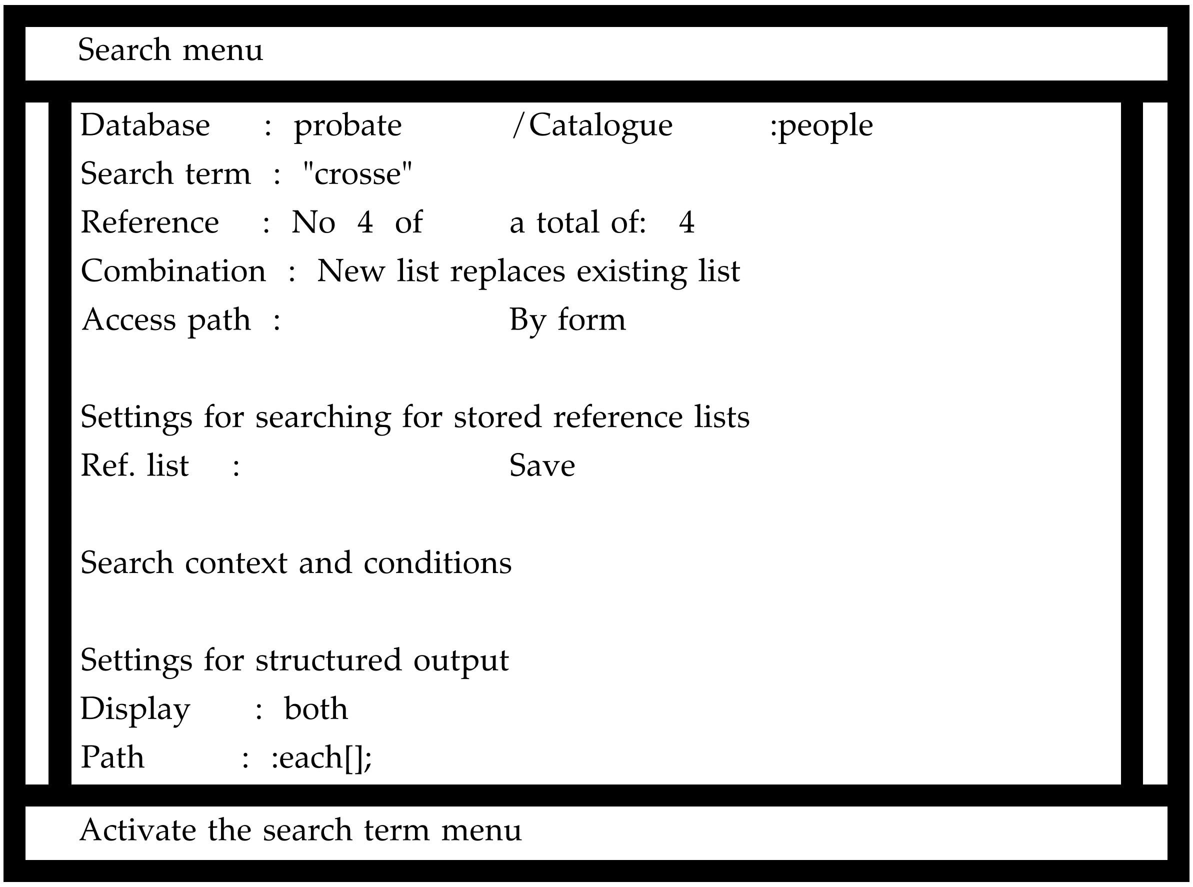
This is slightly more informative. You can press F3 to return to the original format of this result, or press F5 to get a result which looks like this:

```
datasource (15 = "15") :
 Crosse
*****************************************************
datasource (15 = "15") : relp (1 = "rel-1")
          status          appraiser, male
          firstname       James
          surname         Crosse
```

Pressing F5 asks κλειω to display both formats of the result. The 'other' format of the result is defined in the bottom section of the **`Search`** menu. In this case the other format is the result of the same task with the element function **`:each[]`**.

You can also ask κλειω to display information about the other people who have the surname "Crosse" by pressing the PgDn key. This will move you along to the second person in the database with that surname. A further press will move you to the third and so on. If you press it a total of four times, you will be returned to the **`Search term`** menu, as there are only four people in the database with that surname.

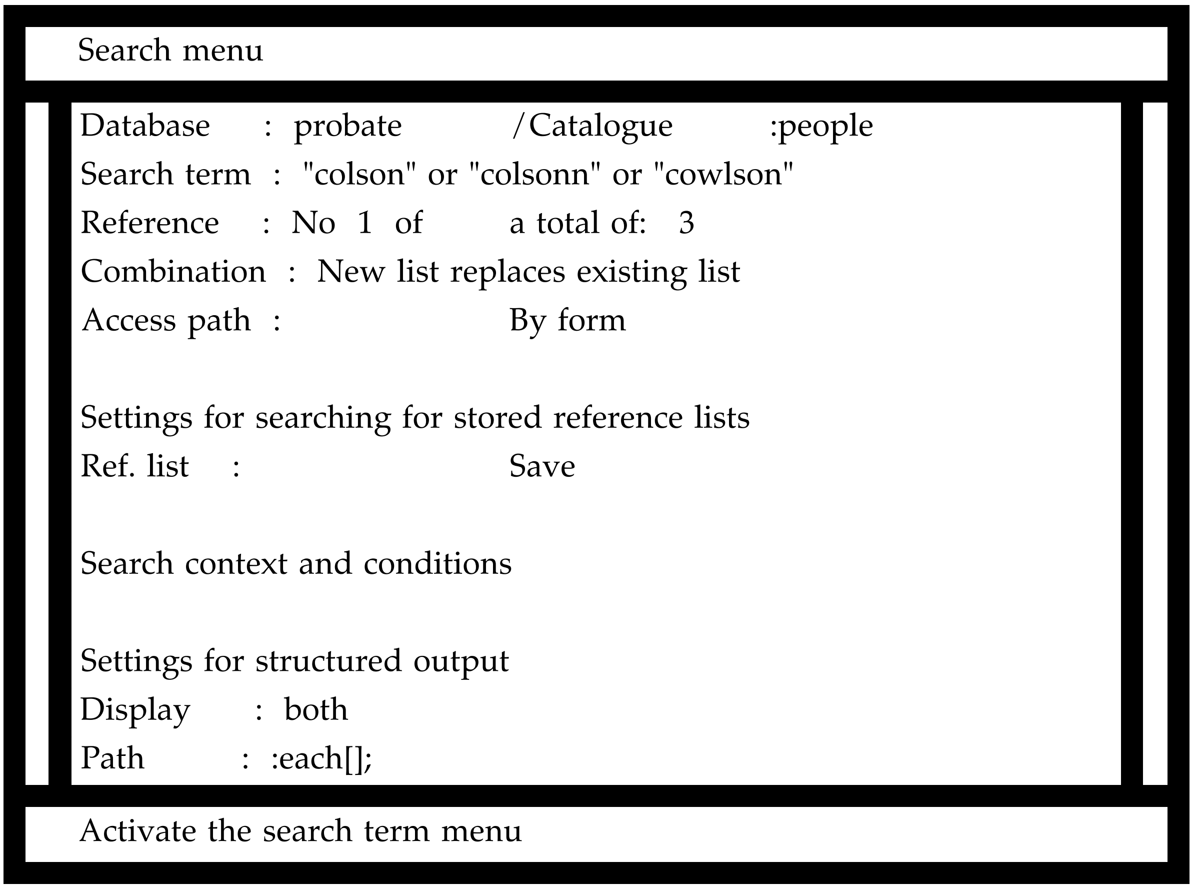
The menu that you will see should look like this:



```
Search menu

Database     :  probate          /Catalogue         :people
Search term  :  "crosse"
Reference    :  No  4  of         a total of:   4
Combination  :  New list replaces existing list
Access path  :                    By form

Settings for searching for stored reference lists
Ref. list    :                    Save

Search context and conditions

Settings for structured output
Display      :  both
Path         :  :each[];

Activate the search term menu
```

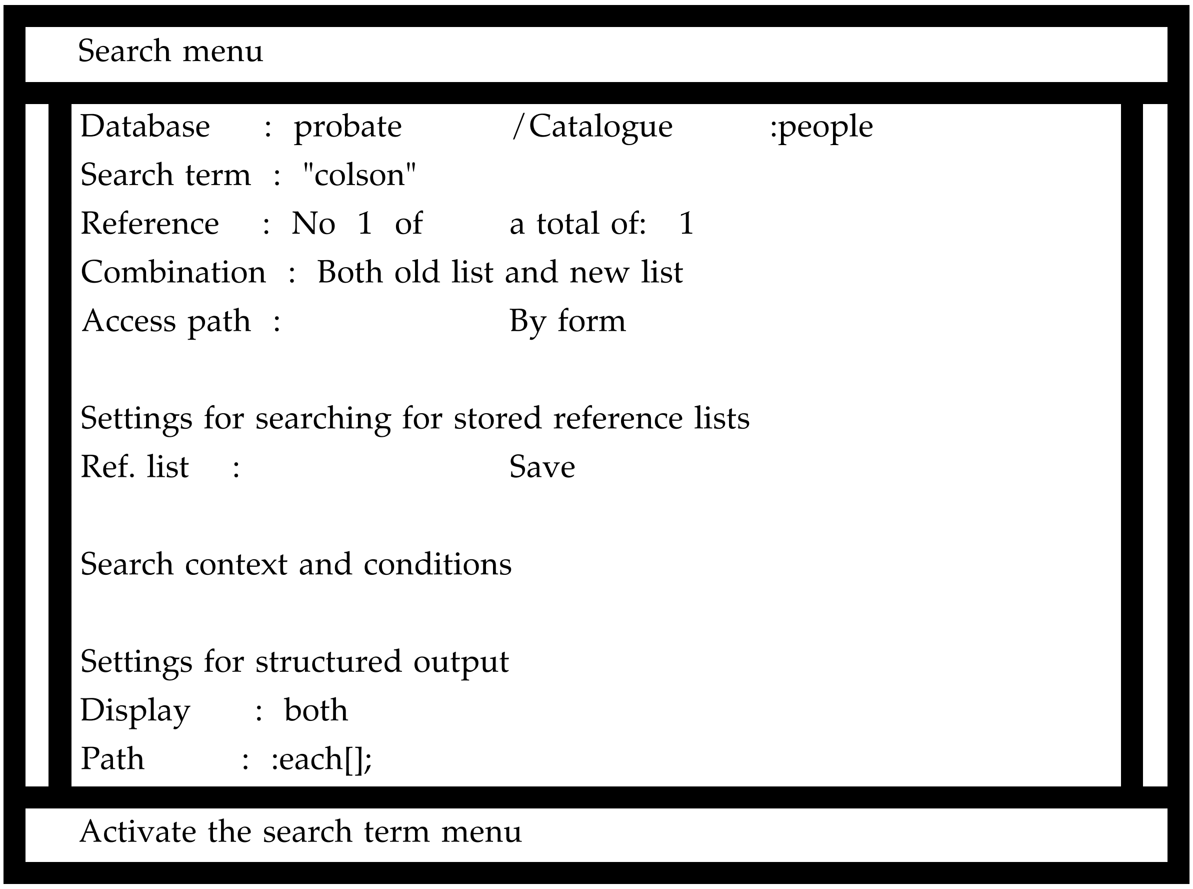
There is a slight change from the previous version of this menu. The **`reference`** field now refers to the fourth occurrence of the term. If you move the highlighted bar to that field and change that 4 to 2 and then press F1 you will get the details of the second person called Crosse in the database. When there are a large number of entries in a catalogue, this is often a quick way of navigating through the database. To return to the **`Search`** menu press Esc.

Imagine that you wanted to see the records of all the people in the database who have a surname like Colson. You should now be at the **`Search`** menu, in the **`Search term`** field. Press either the → or ← key, to move you to the **`Search for words and terms`** menu, and move down the menu to find the names that are like Colson. In this list there are three names which look like Colson. In this menu notice that the name Crosse is no longer highlighted. We will discuss this later. Move down to the three names similar to Colson (Colson, Colsonn and Cowlson) and mark each of them by pressing the Enter key. When you have done this, press the F1 key to return you to the **`Search term`** menu. It should look like this:

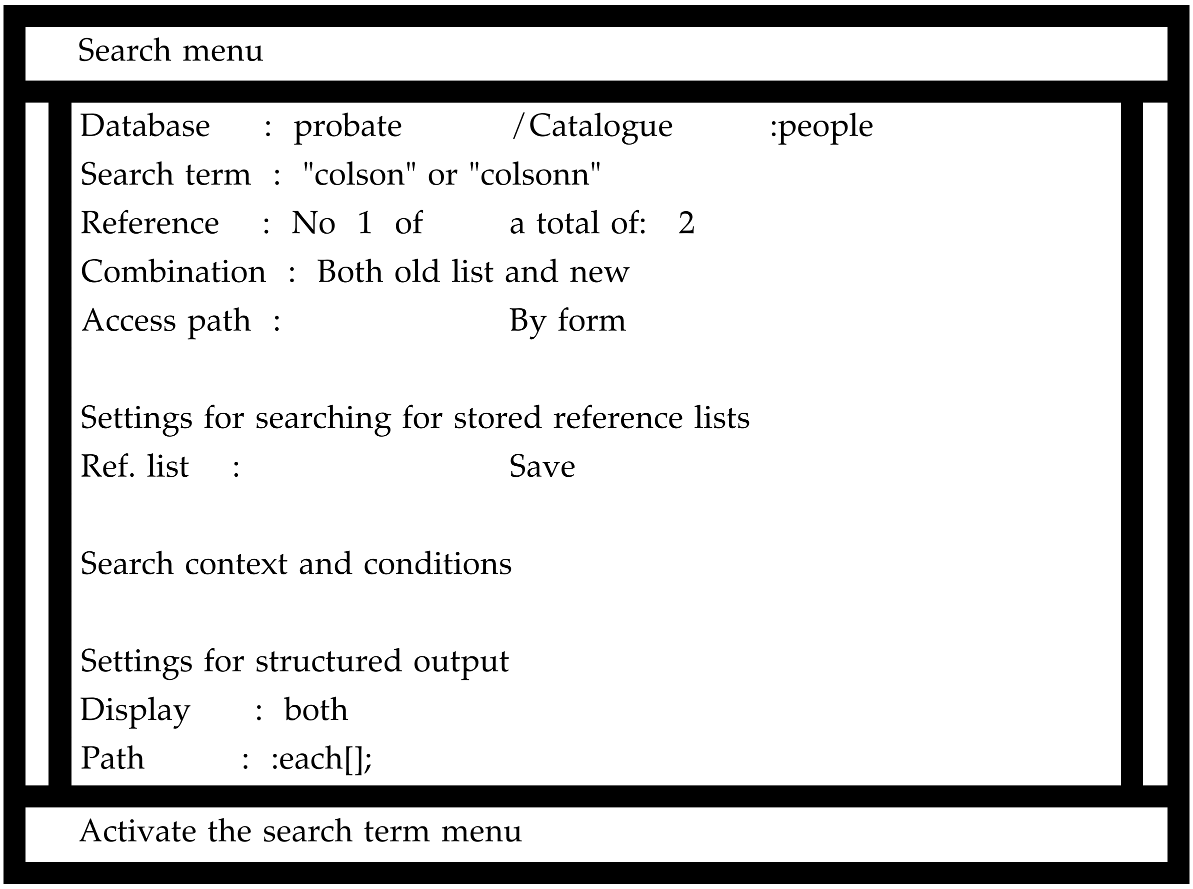

```
Search menu

Database     :  probate          /Catalogue        :people
Search term  :  "colson" or "colsonn" or "cowlson"
Reference    :  No  1  of        a total of:   3
Combination  :  New list replaces existing list
Access path  :                   By form

Settings for searching for stored reference lists
Ref. list    :                   Save

Search context and conditions

Settings for structured output
Display      :  both
Path         :  :each[];

Activate the search term menu
```

You can now look at the details of these people by pressing F1. There seem to be three different Colsons — John, Martin and William.

Earlier we remarked that when we returned to the **`Search for words and terms`** menu, the surname Cross was no longer highlighted. This was because of the **`Combination`** field in the **`Search`** menu. The default for this field is **`New list replaces existing list`**. This means that once we return to the **`Search for words and terms`** menu and instigate a new search, the old search terms have been lost. To demonstrate the other possible 'combinations' available, return to the **`Search words and terms`** list, select a name, say Colson again, return to the **`Search`** menu by pressing F1, move the highlighted bar down to the combination field and press the → key. This should result in the following:


```
Search menu

Database     : probate        /Catalogue      :people
Search term  : "colson"
Reference    : No 1 of        a total of:  1
Combination  : Both old list and new list
Access path  :                By form

Settings for searching for stored reference lists
Ref. list    :                Save

Search context and conditions

Settings for structured output
Display      : both
Path         : :each[];

Activate the search term menu
```

Then move the highlighted bar back up to the **`Search form`** field, press → to go back to the **`Search for words and terms`** menu and mark a different variation of the spelling of Colson. Press F1 and return to the **`Search`** menu which should look like this:



```
Search menu

Database     :  probate          /Catalogue        :people
Search term  :  "colson" or "colsonn"
Reference    :  No  1  of        a total of:   2
Combination  :  Both old list and new
Access path  :                   By form

Settings for searching for stored reference lists
Ref. list    :                   Save

Search context and conditions

Settings for structured output
Display      :  both
Path         :  :each[];

Activate the search term menu
```

Notice that by changing the combination you can ask κλειω to produce conditions. In this case using the **`Both old list and new list`** option causes κλειω to operate as though each list of terms is linked by an ′or′ operator. If you move down to the combination field again and use the → key, you will see that there are a total of four different operations that can be performed. We have seen two. The other two, **`Textual references common to both lists`** and **`textual references in old list but not in new list`**, are used to represent the conditions ′and′ and ′not′ respectively.

With the catalogue that we have at the moment, however, the ′and′ operator is useless as none of the entries in the elements contain more than a single word.

We suggest that before you move on you should play with what we have already demonstrated.

Before returning to the command driven part of κλειω we shall demonstrate a couple of the other features of the menu-driven system.

Pressing the Esc key a few times should return you to the DOS prompt. Re-enter the menu system, interactively process the catalogue people again and return to the **`Search`** menu. Then move to the **`Search for words and terms`** menu (by pressing → on the **`Search term`** field in the **`Search`** menu). Choose three or four different names and mark them by pressing Enter. Press F1 to return to the **`Search`** menu and F1 again to see the results of your task. If you wanted to send these results to another file so that you could look at them together rather than singly on the screen, press F2 when you have some result on the screen. (We suggest that you press F5 before pressing F2.) Once you have pressed F2 you should get a menu which looks rather like this:

```
Data Output

Output file : result.lis
Number      : 1
Replace     : Save
Print file  : No

Please enter the name of your output file
```

If you were to press F1 now (please don't!), κλειω would produce an output file called result.lis which contained one reference without adding page breaks. Each of these fields can be changed. For instance if you have more than one reference in the database that you wish to save you should move to the **`number`** field and alter the number. You can ask κλειω to output more entries than you have, so if you are unable to remember the total number of entries that you have it is perfectly possible to ask for a much larger number. The **`replace`** field has two other possible answers, **`overwrite`** and **`extend`**. **`Overwrite`** tells κλειω to overwrite a file that already has the same name as the file you intend to save your list to, and **`extend`** will add your list to the end of a file which you have already created. The print file field allows the keywords **`yes`** and **`no`**. **`yes`** inserts a special form feed character which will cause almost all printers to start a new page.

When you are ready to create the output file, press F1 and it will be made. Leave κλειω and edit the file you have created.

Now that we have covered some of the easier points of using the menu-driven system, we shall create another catalogue containing different material from the same database.

8.4 Creating further catalogues

The probate database contains almost 1700 entries found in the element item. To keep an overview of all these terms would be very hard if it had to be done 'manually'. This section introduces new ways to organise these terms using a catalogue.

A catalogue of all the items in the database can be created like this:

```
query name=probate;part=:item
catalogue name=objects;part=:item
stop
```

That task makes a catalogue of all the words in the element item of the database, not all the entries. By default κλειω assumes that catalogues are used for full-text analysis, so a further parameter must be used to ensure that terms found in the database are catalogued rather than just the words. This parameter is **`type=`** and should have the keyword **`terms`** as a value. For example:

brewing vates	**`type=words`**	(default)
brewing vates	**`type=terms`**	

If have already created a catalogue using the task above, the following task will not work.

Example 8.7

```
query name=probate;part=:item
catalogue name=objects;part=:item;type=terms
stop
```

This is because a catalogue called objects already exists. There are two solutions to this problem; either to change the name of the new catalogue or to add an **`overwrite=yes`** parameter to the end of the **`catalogue`** directive, as follows:

```
query name=probate;part=:item
catalogue name=objects;part=:item;type=terms;overwrite=yes
stop
```

The parameter **`overwrite=`** is a general parameter which can be used in a variety of places. This parameter will be used increasingly in different places. Remember the basic principle that κλειω will not overwrite anything unless it is specifically told to.

The **`catalogue[]`** group function can now be used to query this catalogue.

Exercises

Exercise 8.1

Formulate a command which looks in the catalogue objects for the term "beere" and displays each element belonging to the same group.

If you were interested in the terms in the database that refered to brewing equipment, you would find that there are many more terms than just those which contain the string "beere". (Forget for a moment that you have read about the menu-driven system.)

The following command might start to help but as there would be no perfect match on the word "brewing" the keyword **`complete`** would have to be changed to **`starts`**, which gives all the terms beginning with "brewing".

```
query name=probate;part=catalogue[objects,starts,"brewing"]
write part=:each[]
stop
```

Exercise 8.2

Which people left brewing equipment when they died? (Do not display information relating to the equipment for the time being.) Look at the diagram displaying the hierarchy of the database.

Exercise 8.3

Modify the above task to make an index of the brewing equipment, followed by the names of the people who owned them. E.g.

brewing fate Guy Thomas 22

8.5 The group function `root[]`

With this database, the task in exercise 8.3 works because the groups piece and list, which both contain the element item are both one group lower in the hierarchy than the group p. If the bottom of the database were changed to this (with everything else the same as before):

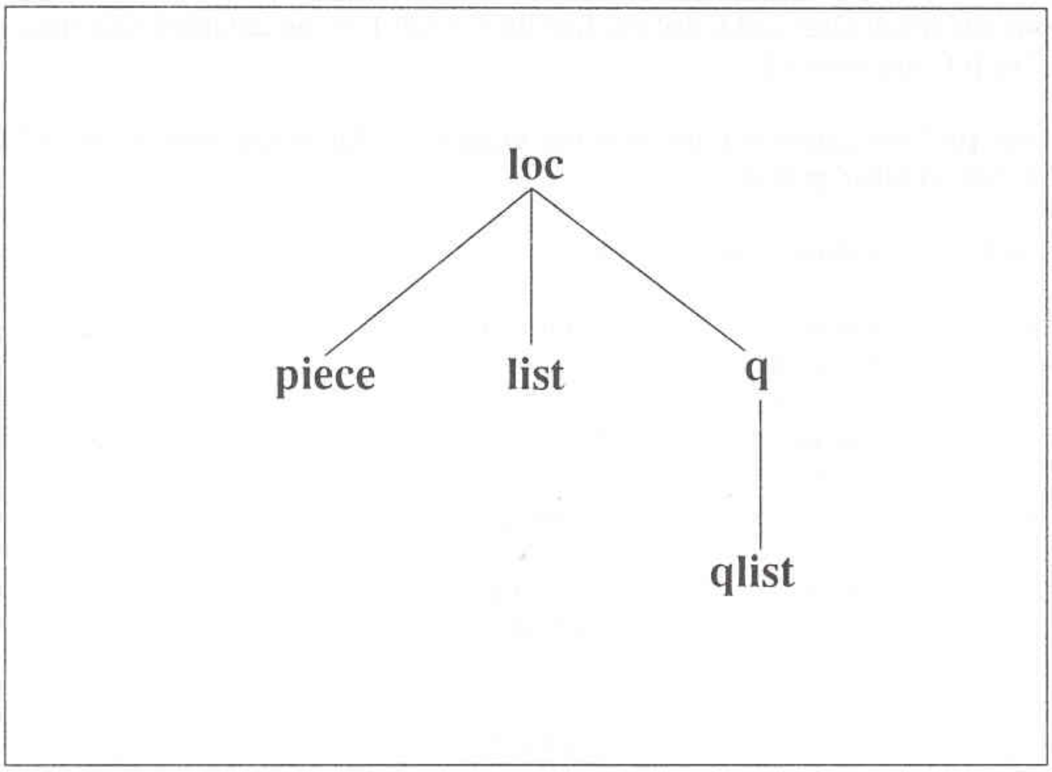


Figure 8.1

and the group qlist also contained the element item, κλειω would not be able to produce information about the entries in p relating to those items in qlist using the same task given above. Why not? Because to move back two levels in the hierarchy from list would be the equivalent of moving back to the group inventory, and the forward slash would move us down one level from there to the group p (as specified). In the case of qlist, to go back two levels would take us to the group loc, but there is no group called p down the hierarchy from loc. This means that a command which moves us to an absolute level rather than a relative level is needed. It would be possible (if qlist existed in the database) to represent the task shown in Exercise 8.2 as follows:

Example 8.8

```
query name=probate;part=catalogue[objects,starts,"brewing"]
write part=back[2]/p:each[],back[3]/p:each[]
stop
```

But this would be a rather inelegant way of solving this problem. It would be much better to use the group function **`root[]`** to refer to absolute levels in the database rather than relative levels as shown above.

Look at the diagram showing the hierarchy of the whole database. On the left hand side is a list of levels. Every single database in κλειω has these levels built in. In this database the groups list and piece are both found on level 3 of the database and p is found on level 2. To get from list to p in the database it would be possible to go back (using **`back[]`**) and then down one level. One could also go directly to level 1 of the database and then down one level to p using **`root[]`**.

Thus in the probate database these selected groups the following uses of **`back[]`** and **`root[]`** relate to other groups.

group	corresponds	to
list	**`back[3]`**	datasource
list	**`back[2]`**	inventory
list	**`root[2]`**	loc
list	**`back[1]`**	loc
list	**`root[1]`**	inventory
list	**`root[0]`**	datasource
loc	**`back[3]`**	not defined
loc	**`back[2]`**	datasource
loc	**`root[2]`**	loc
loc	**`back[1]`**	inventory
loc	**`root[1]`**	inventory
loc	**`root[0]`**	datasource

Thus the following task might replace the task shown in Exercise 8.2:

Example 8.9

```
query name=probate;part=catalogue[objects,starts,"brewing"]
write part=root[1]/p:each[]
stop
```

Exercise 8.4

Create a register of people owning brewing equipment, displaying the surname in the first column and the first name in the second. Do not include any information about the equipment itself. Use the group function **`root[]`**.

The correct solution to the problem is:

```
query name=probate;part=catalogue[objects,starts,"brewing"]
index part=root[1]/p:surname;
         part=:firstname
stop
```

A common misunderstanding with the **`root[]`** function might have led you to use the following line in your task:

```
index part=root[2]/p:surname;
```

This line is wrong because it is necessary to move all the way to the first level of the database and then down one level to p. That command would ask κλειω to go to the second level of the database and then down one level to the group p. As there are no groups called p in the third level of the database, κλειω would create a list with no items in it.

Secondly, the second line of the task should not read:

```
index part=root[1]/p:surname,:firstname
```

as it would not produce the first given name of any individual. It would work if the **`index`** command was replaced by the **`write`** command, but this would not sort the individuals alphabetically.

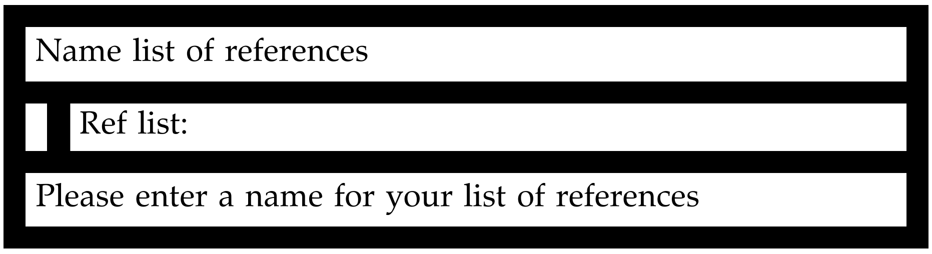
Thirdly, it would be unnecessary to write the whole of the **`index`** command thus:

```
index part=root[1]/p:surname;
         part=root[1]/p:firstname
```

because **`root[1]/p:firstname`** only reinvokes the previous **`part=root[1]/p:`** part of the parameter. κλειω assumes that the user wishes to stay in the same group that is currently being accessed. Thus once in the group p (on the second level of the database), κλειω assumes that the next **`part=`** parameter refers to the same group unless instructed otherwise.

8.6 Creating a reference list

We will now return briefly to the menu-driven system. Enter the system and interactively process the catalogue called objects, which should appear first on the menu. (If you wanted to use the people catalogue again with the menu-driven system, you should toggle on the directory field (using the → key) until the people catalogue appears.) Then get to the search menu and on to the list of terms and words. Select all the terms relating to brewing (e.g. "malte", "barlie", "beare", "beere", and terms beginning with "brewing(e)", "bruing(e)" and "brueing(e)". When you have selected all these return to the search menu. Move to the field below the line `Settings for searching for stored reference lists`. This field, entitled ref. list, tells κλειω that you want to save a reference list (i.e. a list of search terms). Once the highlighted bar is on the Save message, press F1. The following menu should appear:


Name list of references

Ref list:

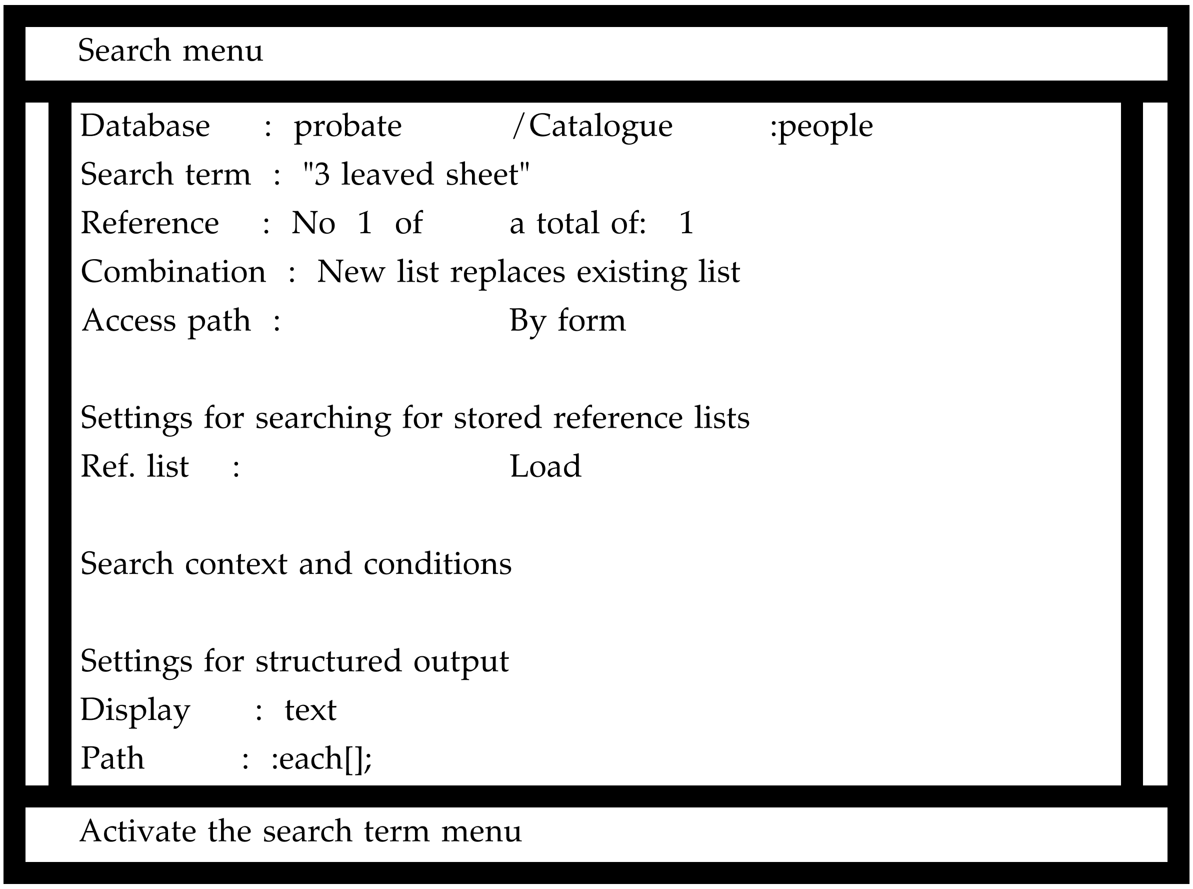
Please enter a name for your list of references

The field in the middle should be highlighted. Enter a name for your list of references. You might try brewingequip. (As this is not a DOS-limited name, it can be more than eight characters long.) Then press F1 to save these search terms.

A message will appear on the screen telling you that κλειω has performed this task. You will need to press ESC to acknowledge this.

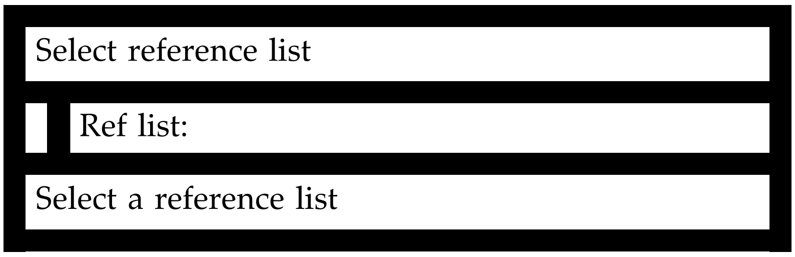
Now exit the menu-driven system by pressing ESC as many times as is necessary to take you back to the DOS prompt, and then return to the menu-driven system and proceed as before until you get to the Search menu.

Then move down to the Ref. list field and toggle on the entry (using the → key) until the word Load appears in this field. The menu should now look like this:



```
Search menu

Database      :  probate          /Catalogue          :people
Search term   :  "3 leaved sheet"
Reference     :  No  1  of        a total of:   1
Combination   :  New list replaces existing list
Access path   :                   By form

Settings for searching for stored reference lists
Ref. list     :                   Load

Search context and conditions

Settings for structured output
Display       :  text
Path          :  :each[];

Activate the search term menu
```

Press F1 to activate this menu, and a new menu will appear:



```
Select reference list
Ref list:
Select a reference list
```

The field entitled Ref. list should have the name of the reference list that you saved. If you then press F1 to activate this menu, you will be returned to the search menu with your old search term in place.

This function is particularly important to users doing analysis on full text where it might take a great deal of experimentation to get all the search terms wanted. This function allows one to fine-tune any task and build up complex search patterns.

8.6.1 The group function `keyword[]`

A reference list created with the menu-driven system can also be used in the command-driven version of κλειω using the **`keyword[]`** built-in function. This is useful when one is interrogating a section of a catalogue, though when one is interrogating the whole corpus of a text it is more effective to use the menu-driven system.

The built-in function **`keyword[]`** must contain two keywords within its square brackets. The first must refer to the name of a catalogue relating to the database named in the **`name=`** parameter of a **`query`** command; the second must refer to a list of references created in the menu-driven system.

Example 8.10

```
query name=probate;part=keyword[objects,brewingequip]
write part=:each[],root[1]/p:each[]
stop
```

This produces a result like this:

```
datasource (2 = "2") : list (2 = "lis-2")
        item            beere

datasource (2 = "2") : p (1 = "p-1")
        status          dead, male
        firstname       Thomas
        surname         Stobbington
        parish          St Johns in the Socke
```

This reference list is part of the logical object called objects. It is possible to design a new reference list without using the menu-driven system, but this process can be rather cumbersome as it is necessary to type out all of the names of the entries that you want to make part of that new reference list. The format for creating a reference list is shown in the task below:

Example 8.11

```
item name=objects;usage=catalogue;source=probate;
   type=permanent
keyword name=spices;form="anneseed" or "anneseede"
   or "bollarmericke" or "carawaye seede" or "case
   nuttmegges" or "case peper" or "cenimon seede"
   or "licorishe" or "mace" or "muster seede" or "salt"
   or "salte" or "sinimon"
exit name=objects
```

This is not an exhaustive list of all the spices in the probate database, but the problems involved in making this type of list should demonstrate the benefits of using the menu system for this sort of problem.

Example 8.12

```
query name=probate;part=keyword[objects,spices]
write part=:each,root[1]/p:surname
stop
```

8.7 Fine-tuning a task in the menu system

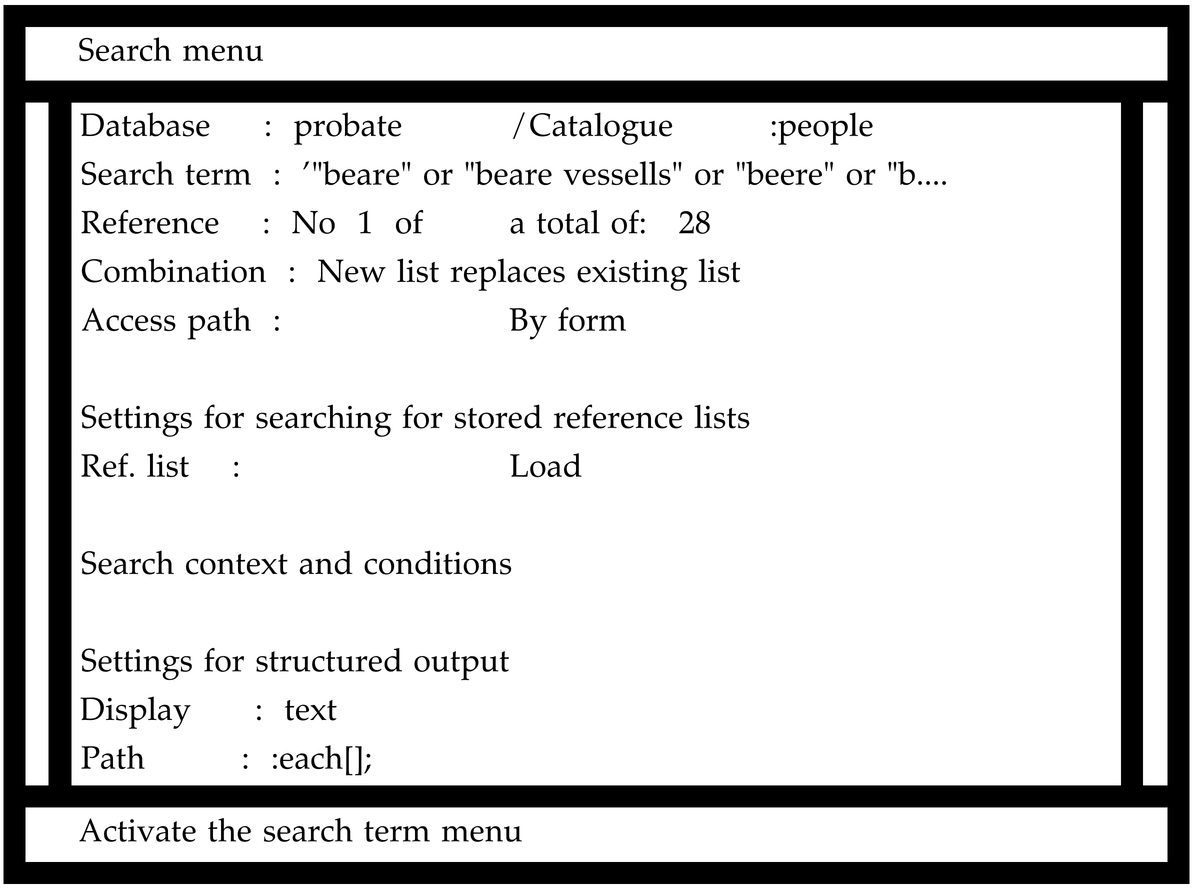
out-of-date

To give a little more idea of what the menu-driven system can do, let us try and repeat the task performed in Exercise 8.4, adding some further information.

The task in Exercise 8.4 asked for the surnames and first names of all those people who owned brewing equipment. In the following task we shall attempt to get the same information together with information about the goods that they possessed.

Leave the menu system and return to it. If you have saved a list of all the search terms which contain information about brewing equipment, load it again. (There should be around 28 terms.)

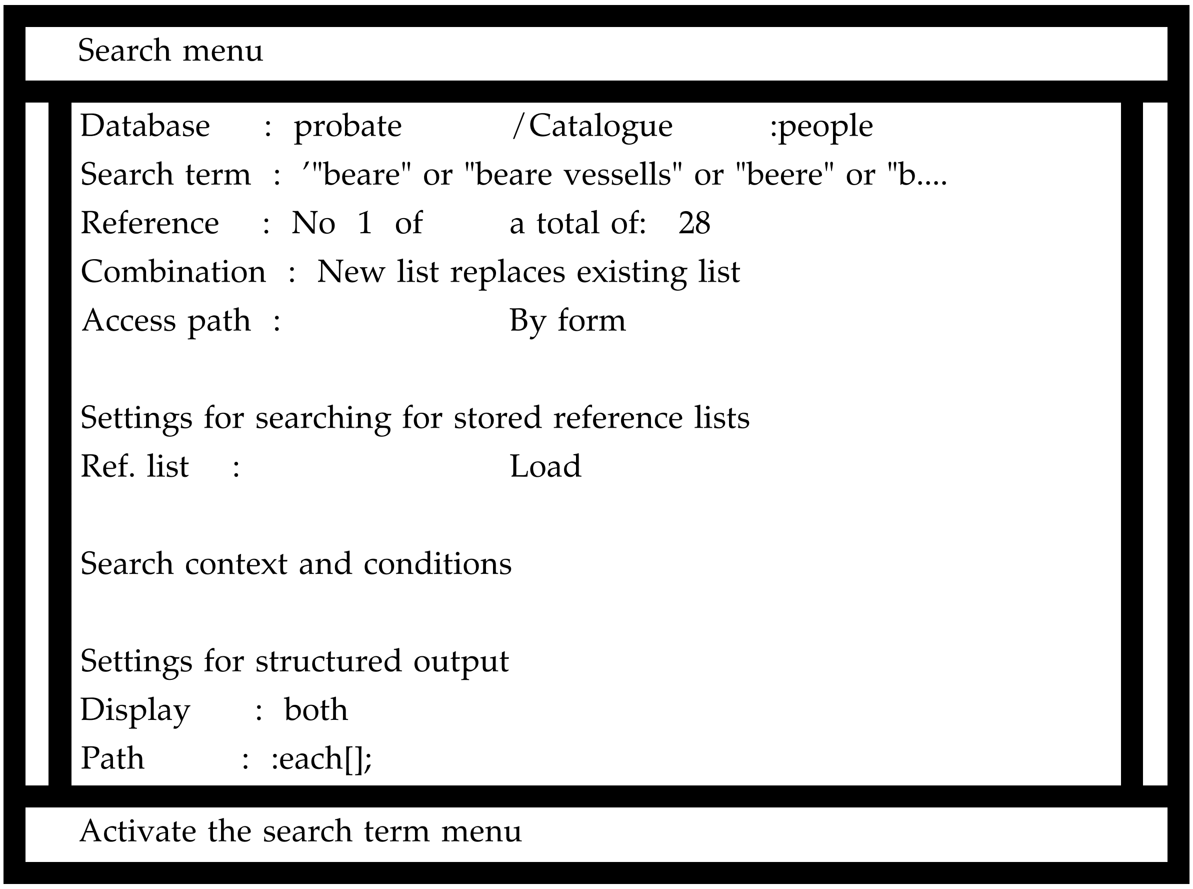
Your **`Search term`** menu should look like this:



```
Search menu

Database     :  probate          /Catalogue        :people
Search term  :  ′"beare" or "beare vessells" or "beere" or "b....
Reference    :  No  1  of        a total of:   28
Combination  :  New list replaces existing list
Access path  :                   By form

Settings for searching for stored reference lists
Ref. list    :                   Load

Search context and conditions

Settings for structured output
Display      :  text
Path         :  :each[];

Activate the search term menu
```

Press F1 to see the information, then press F5 to see both the information and the structured output. Return back to the **`Search`** menu (by pressing Esc).

Your **`Search`** menu should now look like this:



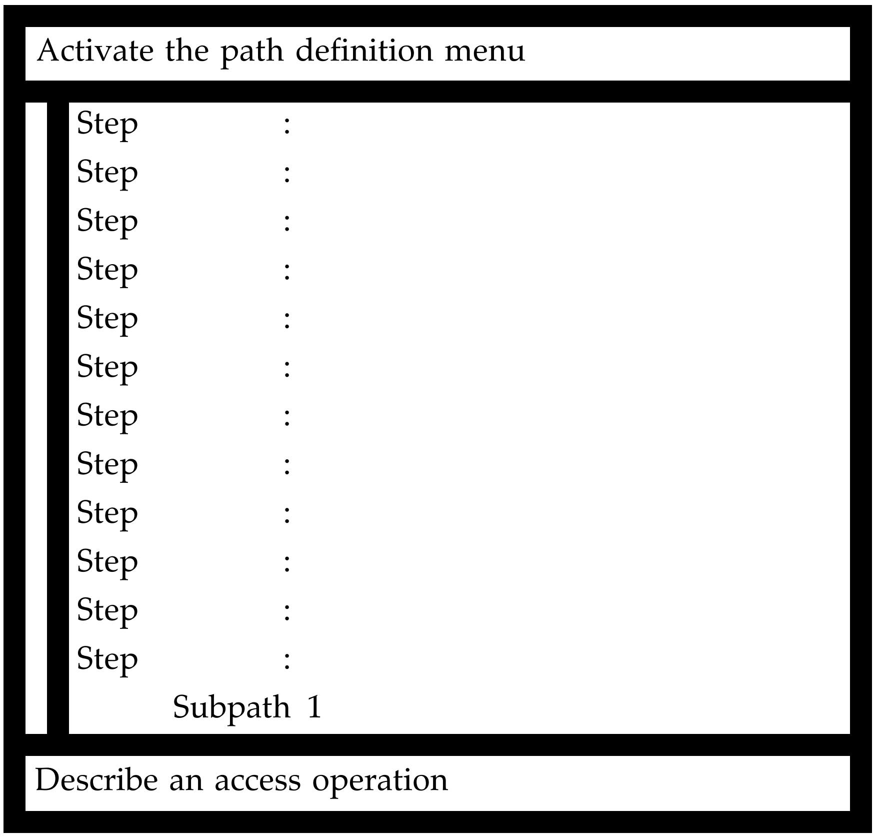
```
Search menu

Database      :  probate          /Catalogue        :people
Search term  :  ′"beare" or "beare vessells" or "beere" or "b....
Reference    :  No  1  of         a total of:   28
Combination  :  New list replaces existing list
Access path  :                    By form

Settings for searching for stored reference lists
Ref. list    :                    Load

Search context and conditions

Settings for structured output
Display      :  both
Path         :  :each[];

Activate the search term menu
```

Notice that the contents of the display field has changed to both. If you move the highlighted bar down to that field and toggle (using the → key) you will see that you are allowed to choose the following terms:

- **`text`** default output (the contents in the catalogue)
- **`structure`** ‘structured output’ (based on the rules shown below)
- **`both`** both textual and structured output

Return to the output screen, so that you have both displayed.

Move down to the path field. Activate it by pressing the → key. Your menu should look like this:



```
Activate the path definition menu

Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
       Subpath 1

Describe an access operation
```

You can fill in this menu to allow you to access parts of the database other than the catalogue with which you have been working. This particular menu allows you to define a path in exactly the same way as you would after a **`write`** command. At first, however, you will find this method of specifying paths much more complicated than using the command system, but after time and with a fuller knowledge of κλειω some of the benefits of using the menu-driven system will become apparent.

Let us use this menu to add the first path. Let us start by adding the path **`:each[]`**, which you should remember as an element function which stipulates all the elements within the last group that we specified. As κλειω is using the catalogue as the last group that we specified, we would use the **`:each[]`** element function to find out about all the other characteristics about that item. To add this element function to this menu, start by pressing F5 (which is the key to select element functions). This will produce a menu like this:

```
Select an element function

Selecting entries
Selecting aspects
Selecting elements
Selecting parts of entries
Defining constants
Converting on the basis of declarations/rules for computing
Converting on the basis of codebooks
Selecting/converting data types
Properties of the elements
Properties of the group
```

This menu may seem rather daunting at first. Do not be put off by it. Think about what the **`:each[]`** function represents. It represents the set of all elements within a previously specified group. Therefore, move the bar down to **`selecting elements`** and activate it by pressing F1.

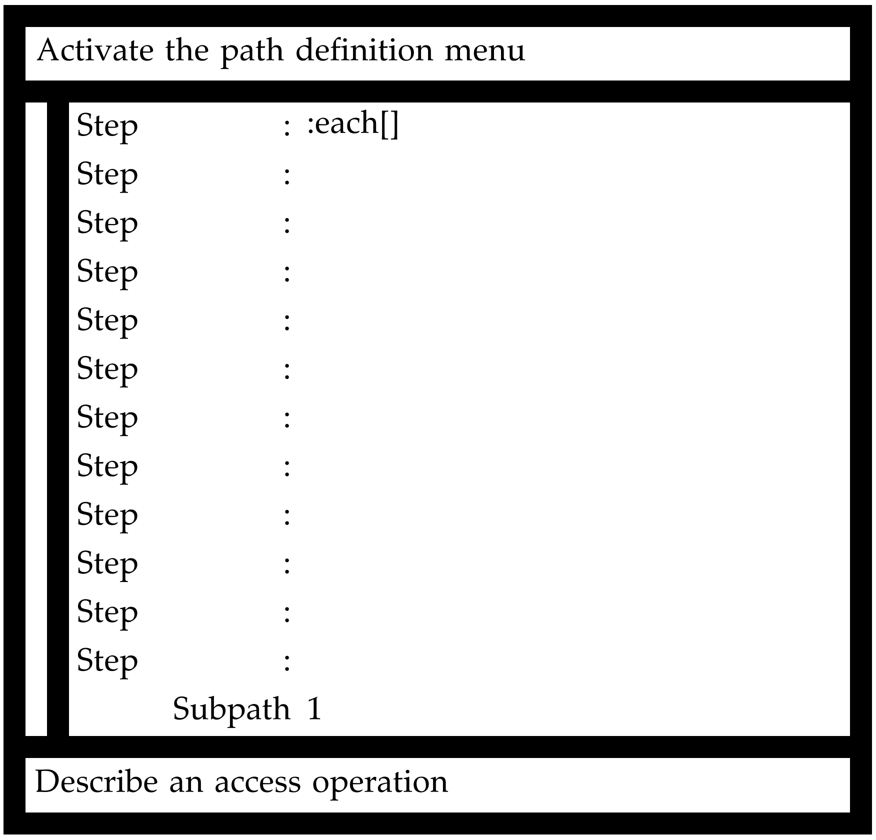
This produces another menu:

```
Select an element function

All elements in a group
All elements in and dependent on a group
Elements with the same name, regardless of structure
Selection of elements in a group, regardless of name
```

As we want to have all the elements within the group in which the catalogue entry is found, we must keep the highlighted bar on the first field and press F1. A further menu will appear:

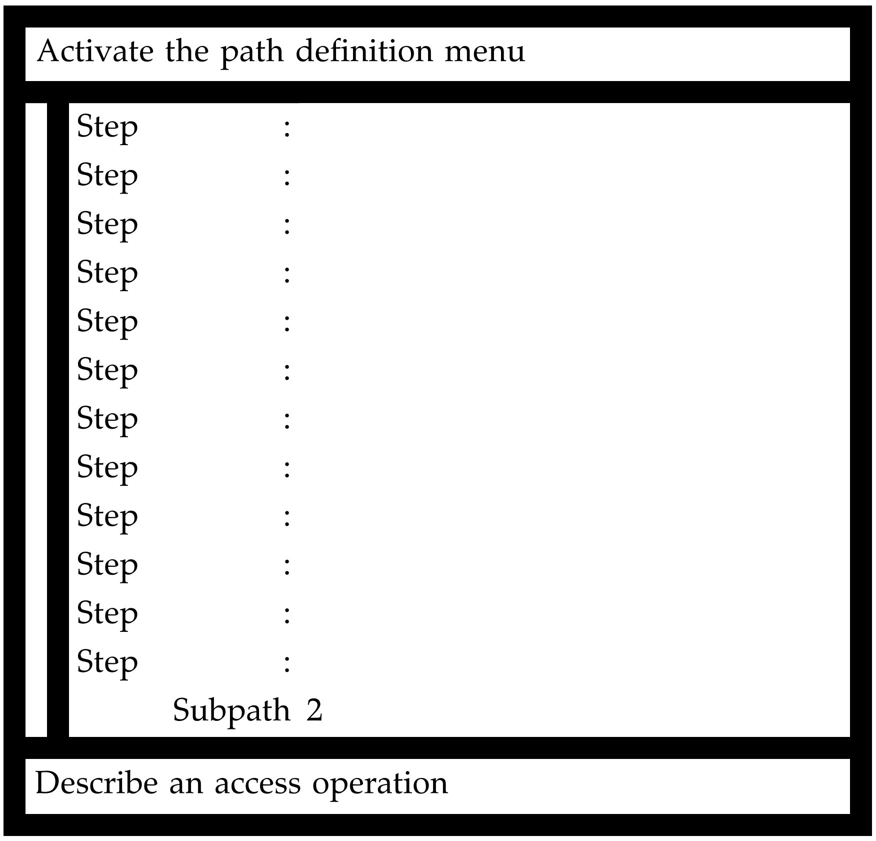
```
Function definition

All elements in a group
(Empty) :

This function has no parameters
```

This tells us that the **`:each[]`** function has no parameters, therefore we need add nothing, and F1 can be pressed again. This should return us to the path definition menu which now looks like this:



```
Activate the path definition menu

Step          : :each[]
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
       Subpath 1

Describe an access operation
```

We have now made the first of our path definitions. Press F7 to move to a second path definition menu. This should look like this:


Activate the path definition menu

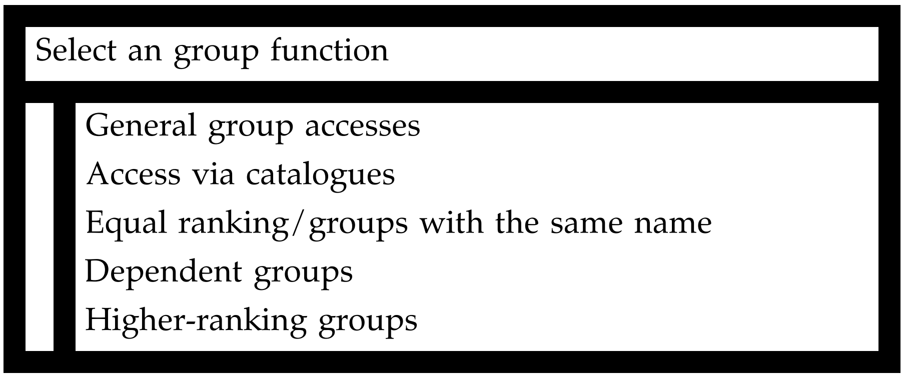
Step :
Step :
Step :
Step :
Step :
Step :
Step :
Step :
Step :
Step :
Step :
Step :
Subpath 2

Describe an access operation

In this path we want to tell κλειω to go back to the top of the hierarchy of the database, then move down to the group p and produce the surnames of that group. In the command language this would be expressed like this:

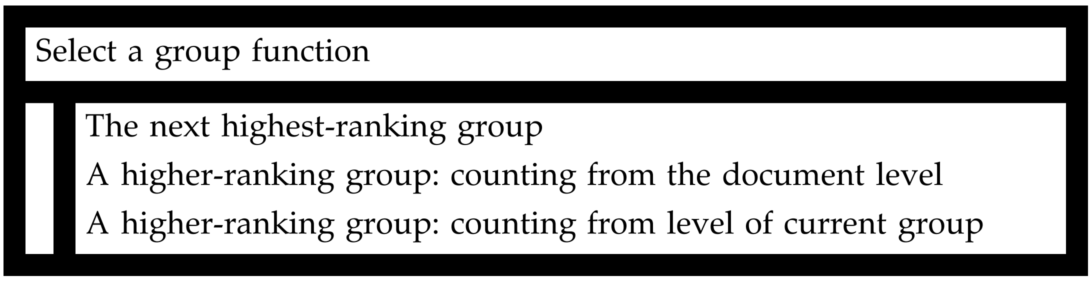
```
/root[1]/p:surname
```

Start by pressing F4. This brings up a menu which allows us to invoke a group function:


Select an group function

General group accesses
Access via catalogues
Equal ranking/groups with the same name
Dependent groups
Higher-ranking groups

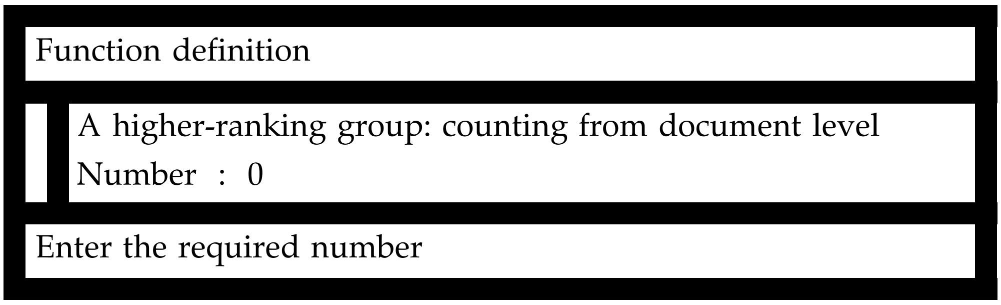
As the group p is a higher-ranking group than the groups which contain the element item, we need to activate the higher-ranking groups field. Move the cursor to the appropriate field and press F1. A further menu appears:



Select a group function

The next highest-ranking group
A higher-ranking group: counting from the document level
A higher-ranking group: counting from level of current group

As the element item is in two different groups we can only use the **`root[]`** function. Therefore activate the field containing the text **`A higher-ranking group: counting from document level`**. A further menu appears:



Function definition

A higher-ranking group: counting from document level
Number : 0

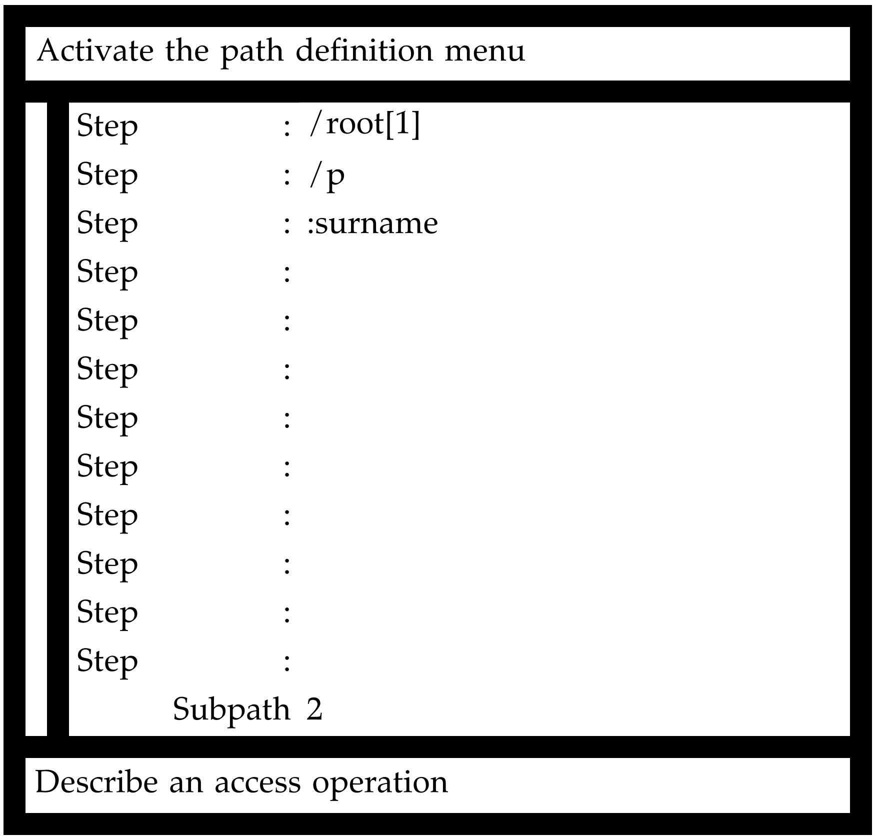
Enter the required number

Here, as in the command system, we need to specify at which level of the hierarchy of the database we want κλειω to leave us. In this case it is the first level. So enter 1, in the Number field and activate this menu by pressing F1.

Look at the path definition menu. It has changed. There are two other parts to this path. First, press F2 to select a group. This will produce a menu of all the available groups in the database. However, κλειω does not know which groups would be acceptable from the position in the database that you will be left in after the **`root[1]`** function has been performed, so ensure that you know the structure of the database you are using when you attempt this. An error can leave κλειω very confused and mess up the task on which you are working.

Move down this menu until you reach the group p. Select it by pressing F1. You will be returned to the path definition menu. Notice how it has changed.

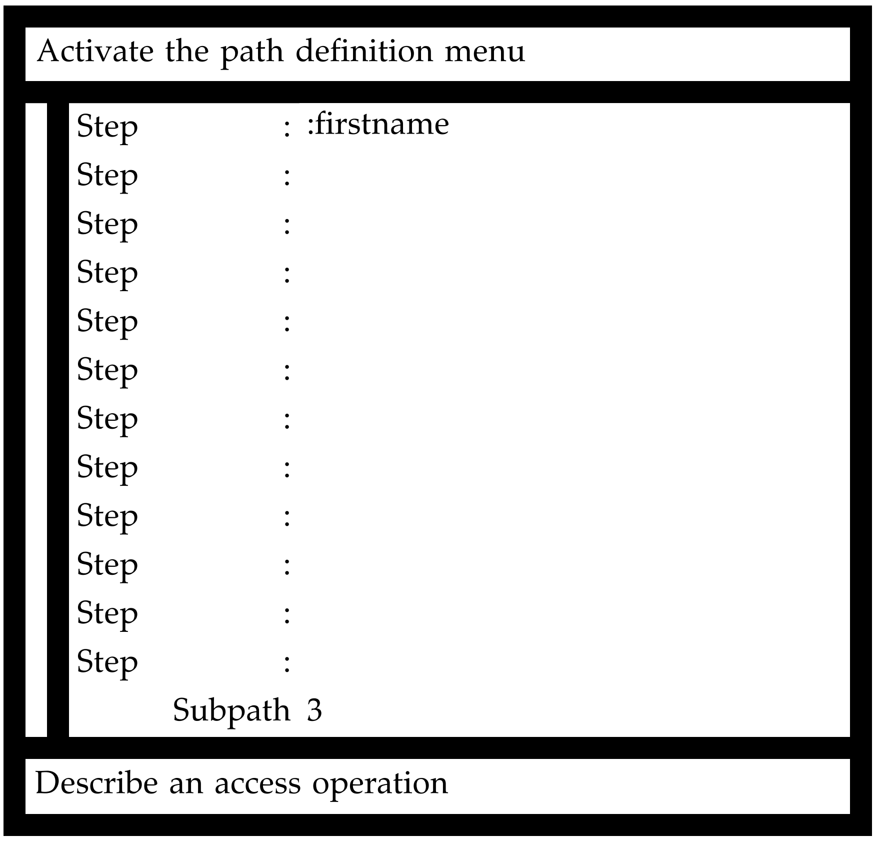
Now press F3. This will allow you to select an element. A menu will appear containing all the elements in the database probate. Choose the element surname, and then press F1. Your path definition menu should now look like this:



```
Activate the path definition menu

Step        : /root[1]
Step        : /p
Step        : :surname
Step        :
Step        :
Step        :
Step        :
Step        :
Step        :
Step        :
Step        :
Step        :
       Subpath 2

Describe an access operation
```

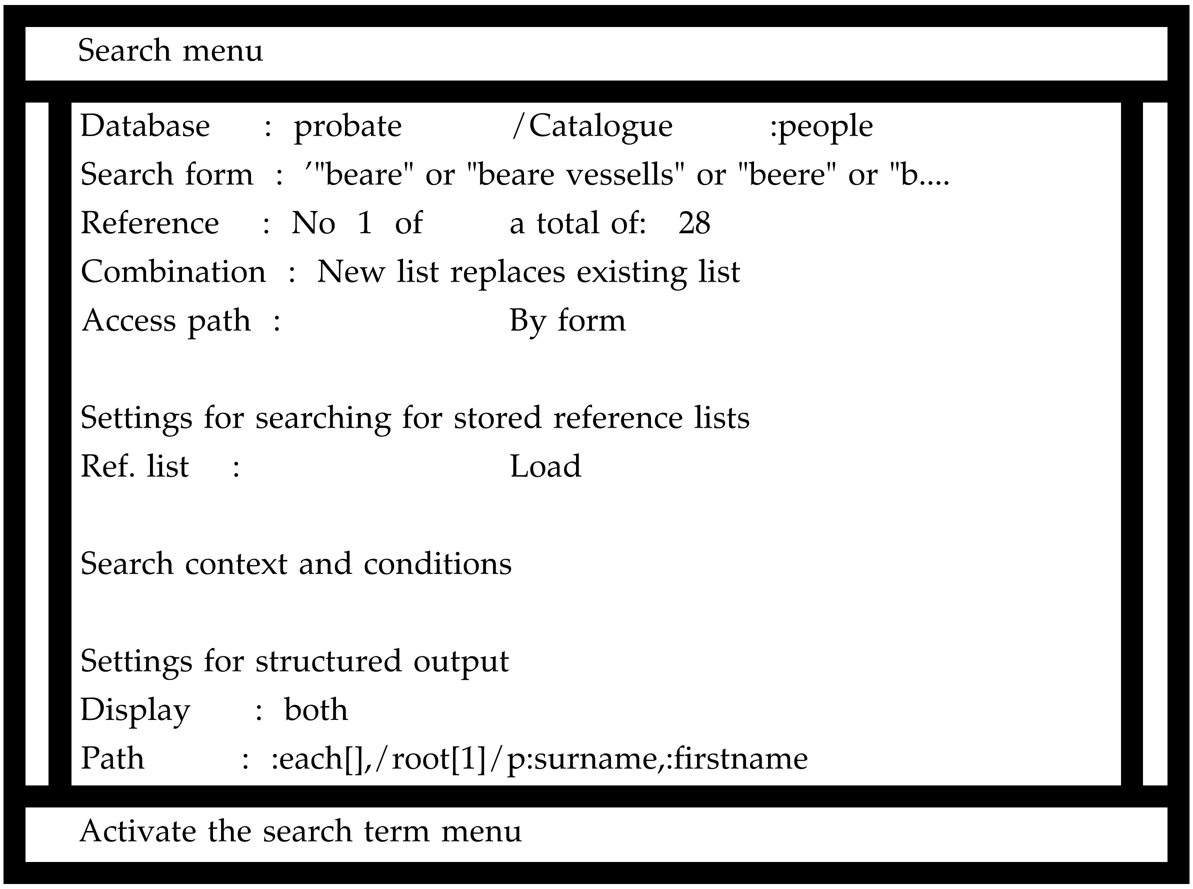
To get the first name of this person as well, we need to specify the element firstname. Press F7 to define a third path. A new path definition menu should appear. Press F3 to produce a list of all the elements in the database again. This time choose firstname. Press F1 to select this element. Remember we do not need to specify the **`root[]`** function or the group p a second time as κλειω is already working at that group level.

This should produce a third path menu which looks like this:



```
Activate the path definition menu

Step          : :firstname
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
       Subpath 3

Describe an access operation
```

Now press F1 a further time to activate these three paths that you have defined. This will produce a search menu that looks like this:



```
Search menu

Database     :  probate          /Catalogue         :people
Search form  :  ′"beare" or "beare vessells" or "beere" or "b....
Reference    :  No  1  of        a total of:   28
Combination  :  New list replaces existing list
Access path  :                   By form

Settings for searching for stored reference lists
Ref. list    :                   Load

Search context and conditions

Settings for structured output
Display      :  both
Path         :  :each[],/root[1]/p:surname,:firstname

Activate the search term menu
```

If you press F1 again you should see some results.

```
datasource (31 = "31") :
 mollt

****************************************************

datasource (31 = "31") : piece (3 = "pie-3")
        quantity
         original          vi Boshells
        value              144.000000
        item               mollt
datasource (31 = "31") : surname         Earlle
datasource (31 = "31") : firstname       Edman
```

This is just one example of the items retrieved from the database. The top half is the part generated directly from the catalogue, the second half is the information taken from the whole of the database (as we defined with the path definitions). If you do not see all this output, press F5 to see the material produce from the paths you have defined.

As you will have noticed, a large number of features have been touched on but not described. Once you are fairly familiar with the command system of κλειω most of the functions of the menu-driven system will be moderately clear. We suggest trial and error as the best way to find your way through it, but once you have the confidence gained by using a database containing your own material, performing simple queries should make using the menu system a much more pleasant task.

Conclusion

There are three main uses of catalogues in κλειω:

- to speed up the querying process
- to facilitate full-text queries
- to categorise information (also known as 'keywording').

When you are using the material from the probate database, catalogues could be very important looking up categories of household items. If one were studying clothes or dyeing or cheese manufacture, a catalogue relating to each of those areas could easily be made to facilitate analysis. However, if one wanted to relate all of the items within a household to an area, catalogues might be an unwieldy way of managing this. The solution to this is to create a codebook which allows the user to designate a code to every item found in the database in order to amalgamate them. A codebook is rather like a look-up table in traditional data processing. The codebook allows one to categorise all the materials in the database rather than just a selection of terms created interactively.

Exercise 8.5

Create an index from the probate database with four columns:

<relp:surname> <p:surname> <relp:firstname> <p:firstname>

Many people occur more than once. Why? How can this be solved? The solution can be found but it contains one item that has not yet been explicitly discussed! See the answers at the end of this book.

Chapter 9

out-of-date

Interactive text processing

9.1 Introduction

This chapter follows up the work done in the previous chapter. It adds to the understanding gained there of interactive processing of a catalogue. However, in this chapter we will be dealing with the interactive processing of full text rather than the processing of simpler material. In the probate database there are a number of wills or testaments which relate to the same people who had inventories made of their goods and chattels after death. It is this material that we shall be considering in this chapter.

We should also mention that there is very little more to say about the processing of full text with κλειω, as κλειω processes most data in a similar fashion. Therefore whatever we have already said about processing structured textual information with a catalogue also holds for fuller text.

9.2 Cataloguing words

In order to process full text effectively, it is necessary to create a catalogue of all the terms in the database in order to use the menu-driven system to achieve some results.

As we saw in the last chapter, the method of creating a catalogue is easy. In this case we want to make a catalogue of all the words in the wills in the probate database. The following task will do this:

Example 9.1

```
query name=probate;part=:contents
catalogue name=text;part=:contents;type=words
stop
```

However, what you have not seen yet is the part of the .mod file which refers to the structure of the probate database. Within that file is a logical object which refers to the way in which κλειω is asked to handle data of **`text`** type:

```
item name=willmaterial;usage=text
signs part=".?!";without=no
exit name=willmaterial
```

There is of course also an **`element`** directive which links this with the element called contents:

```
element name=contents;type=text;text=willmaterial
```

This logical object is necessary as it tells κλειω to understand the text in the database as **`text`** type data. The **`signs`** directive is the only directive used within a **`text`** definition. This directive is used to tell κλειω how to treat different characters which might be found within the text. The main parameter used within this directive is the **`part=`** parameter, the contents of which define to κλειω which character(s) which are used to define entry separators. By default κλειω only accepts the semi-colon (";") as an entry separator, but when using large blocks of text it is wise to define other characters as potential entry separators. Within the **`part=`** parameter above the characters "?", "!" and "." have been defined as entry separators as these are the usual end of sentence characters. The parameter **`without=no`** has been added to tell κλειω to include these characters within the core information of the database. By default, κλειω does not include entry separators into the core information of the database. In this case, as these symbols are vital pieces of punctuation we ask κλειω to include them in the database. (There are other parameters which can be added to a **`text`** definition: see the Reference Manual, Section 7.3.1.2.)

9.3 Text processing

If you have run the task at the beginning of this chapter, you should have a catalogue called text in operation and ready for processing using the interactive processing of a catalogue feature of the menu system.

Enter the menu system, as usual by typing **`kleio`** at the DOS prompt. This should bring you to a menu. Select interactive processing of a database and press F1.

Kleio Version 5.1.1

Systematic processing of a database
Interactive processing of a database
Creating a new database

This will bring up a further menu:

Interactive processing of databases

Processing a catalogue interactively

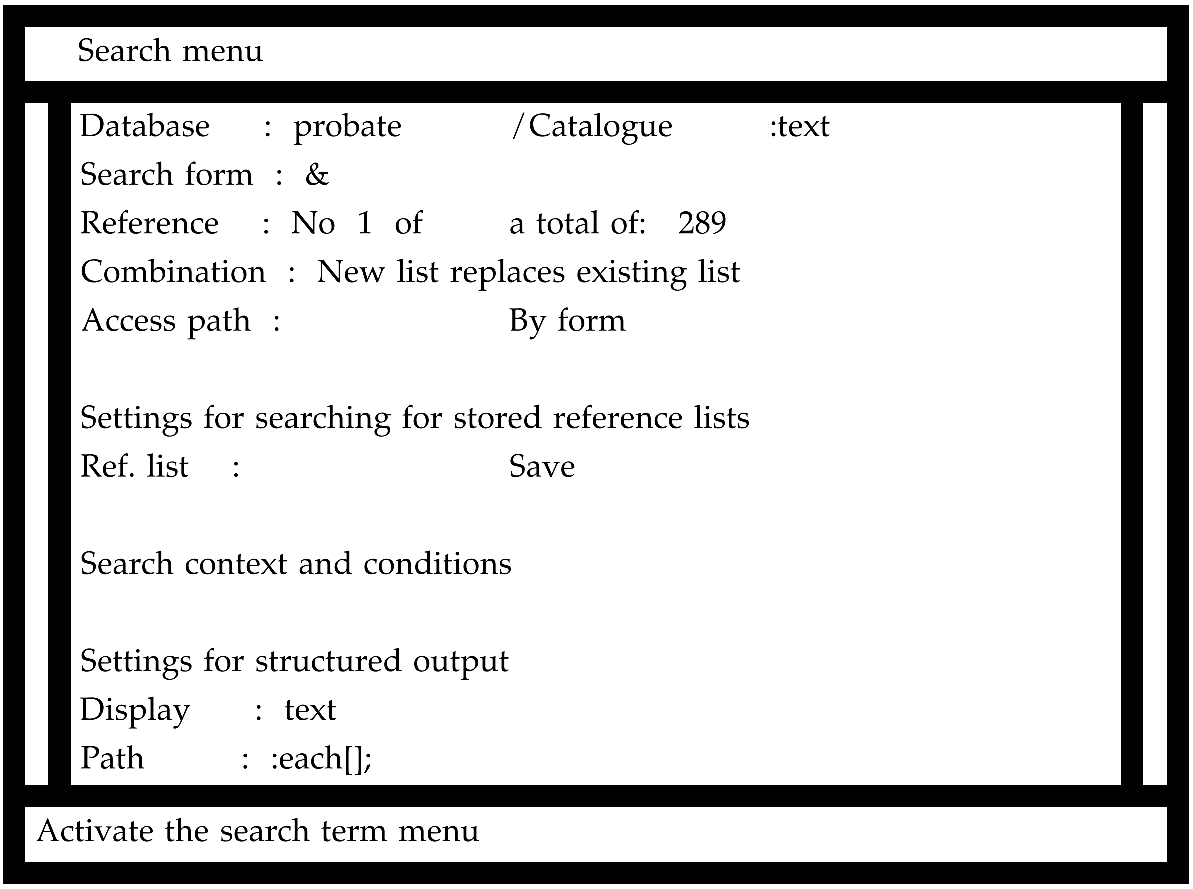
As in the last chapter, press F1 a further time to bring up this menu:

Direct search

Database:

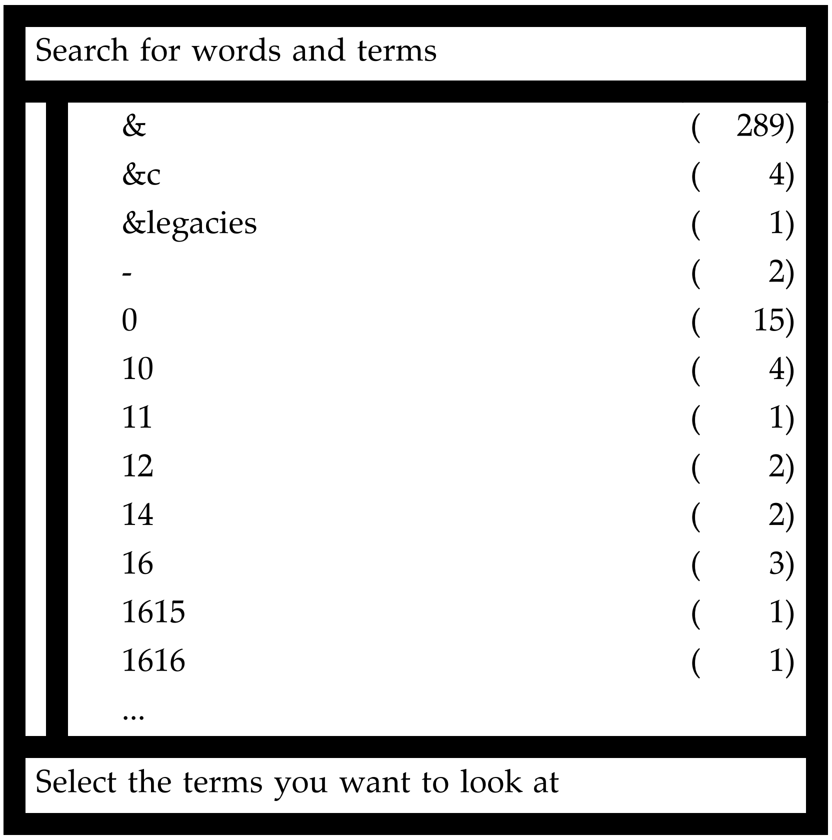
Select the database

and type **`probate`** in the highlighted box. Then F1 to activate this menu. κλειω will produce a further menu with the names of all the possible catalogues available for this database. Using the cursor keys (← and →), toggle until you reach the catalogue you have just created. This should be called text. Activate this choice by pressing F1 a further time. A search menu should appear:



```
Search menu

Database     :  probate          /Catalogue         :text
Search form  :  &
Reference    :  No  1  of        a total of:   289
Combination  :  New list replaces existing list
Access path  :                   By form

Settings for searching for stored reference lists
Ref. list    :                   Save

Search context and conditions

Settings for structured output
Display      :  text
Path         :  :each[];

Activate the search term menu
```

If you have just worked through the last chapter this should be rather familiar.



```
Search for words and terms

    &                                    (  289)
    &c                                   (    4)
    &legacies                            (    1)
    -                                    (    2)
    0                                    (   15)
    10                                   (    4)
    11                                   (    1)
    12                                   (    2)
    14                                   (    2)
    16                                   (    3)
    1615                                 (    1)
    1616                                 (    1)
    ...

Select the terms you want to look at
```

This is not the most fascinating display, but if you scroll down this menu, using the ↓ key or the PgDn key, you should find material that looks more interesting. You can select material from this menu in exactly the same way as you selected material in the last chapter. For example if you wanted to see how the word "trinity" was used in these wills you could select the terms "trinity", "trenitie", "trinite", "trinitie", by pressing the enter key when on those particular terms (thus marking them all with an asterisk), and pressing F1 twice, the first time to return you to the Search menu and the second time so you can see the result of the task.

The result should look something like this:

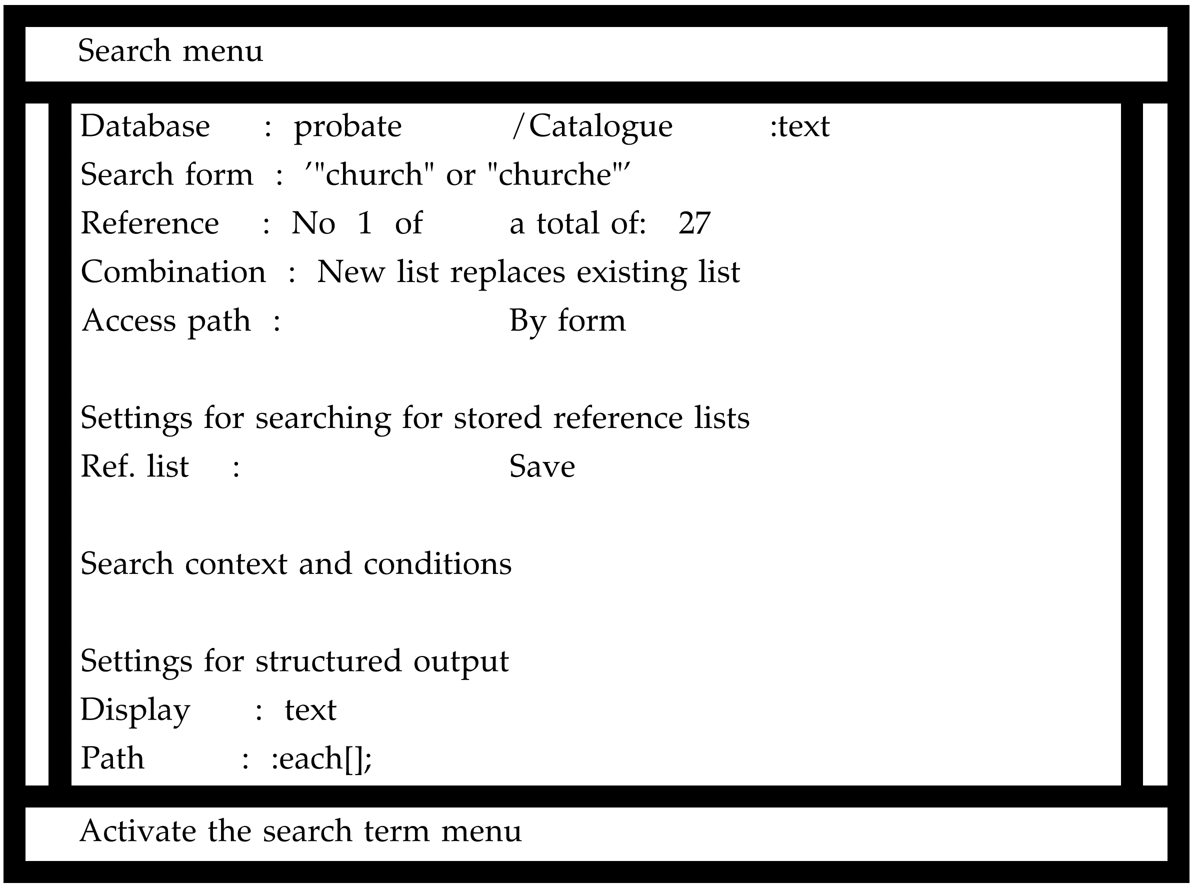
> datasource (4 = "4") :
>
> First I give and bequeath my soule to Allmightie god my maker, & to Jesus christ my redeemer And to god ye holy goste my Sanctifier, & my bodie to be buried in the Trinitie Litten by my first wife.

If you look through the results of this task you would find that "trinity" is used in two ways in these documents, first as the Church of the Holy Trinity and secondly as the name given to the Father, the Son and the Holy Ghost.

The potential for searching full text as made by a catalogue is similar to the method described in the previous chapter. For further details we would refer you there.

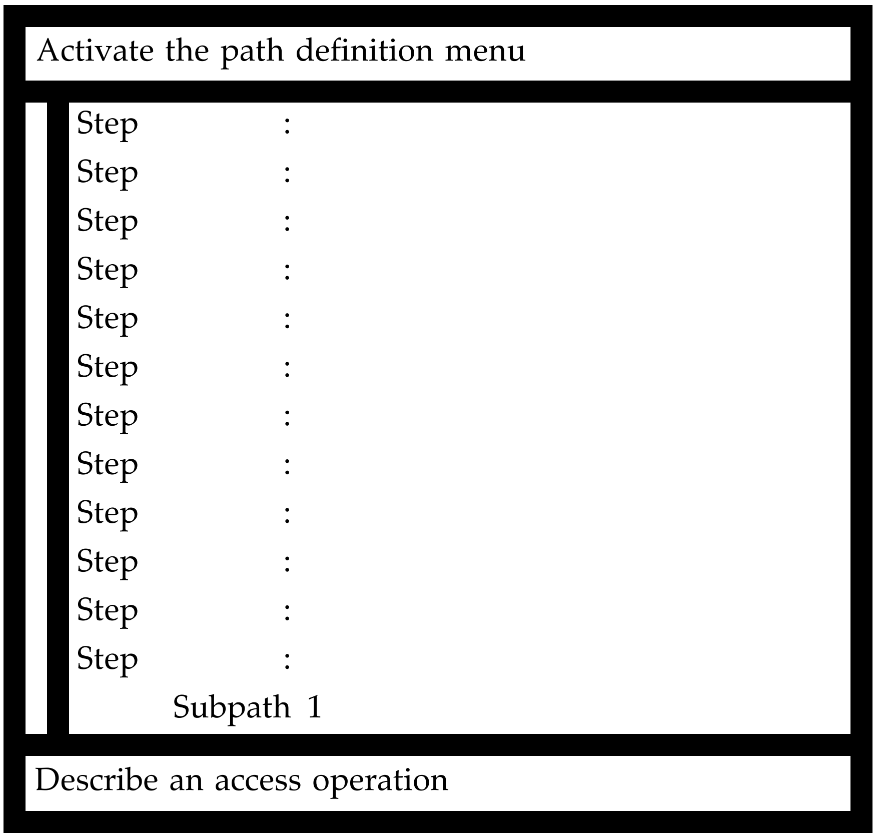
A final example. Let us assume that we are interested in the amount of money left by a person in their will to a church and the total value of goods they left (as given in their inventory). We could formulate a task which would give us this information.

First, using the Search menu, search for all the occurrences of the word "church" and "churche". Your menu should look like this:



```
Search menu

Database     :  probate          /Catalogue        :text
Search form  :  ′"church" or "churche"′
Reference    :  No  1  of        a total of:   27
Combination  :  New list replaces existing list
Access path  :                   By form

Settings for searching for stored reference lists
Ref. list    :                   Save

Search context and conditions

Settings for structured output
Display      :  text
Path         :  :each[];

Activate the search term menu
```

Move down this menu until the highlighted bar is positioned over the path menu. Then press either the right or left arrow keys to activate this menu. This will produce a menu like this:


Activate the path definition menu

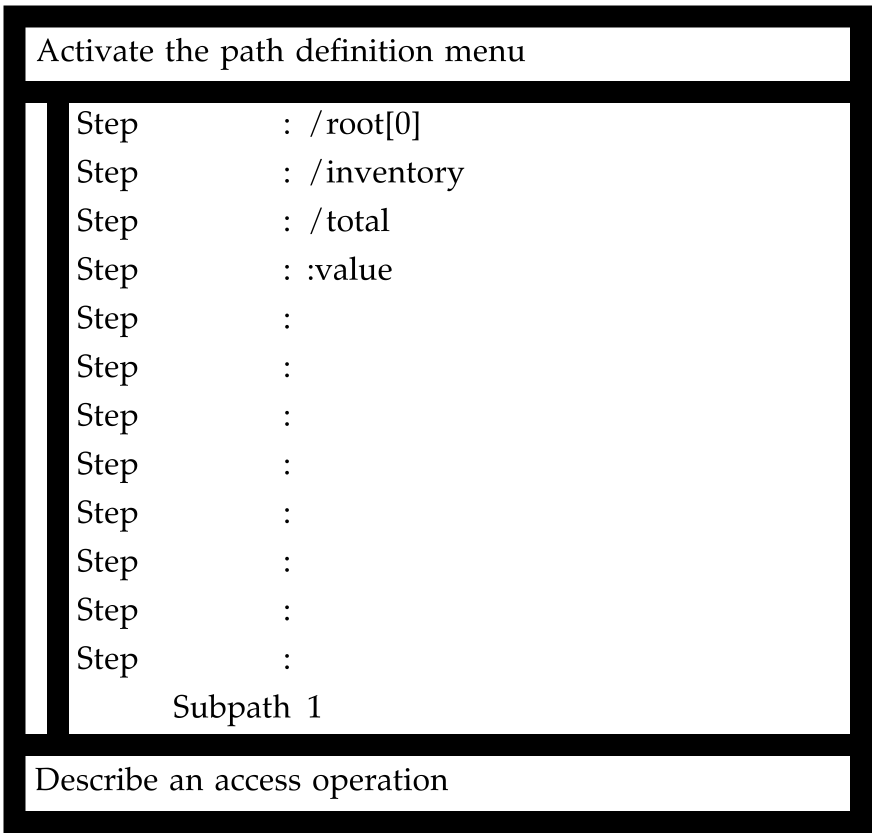
```
Step        :
Step        :
Step        :
Step        :
Step        :
Step        :
Step        :
Step        :
Step        :
Step        :
Step        :
Step        :
        Subpath 1
```

Describe an access operation

This will allow us to access other material from the database probate. In this case we need to describe a path which tells κλειω to go to the highest level of the database, then down two groups to the group total, and then to display the contents of the element value. We can also ask for the first name and surname of the individual involved. In κλειω's command language this query would look like this:

```
write part=root[0]/inventory/total:value,
    back[1]/p:surname,:fname
```

In order to reproduce this in terms that the menu system can understand, first press F4 to select a group function. This will produce a menu asking you to select a group function. Move down this menu until the highlighted bar is on **`higher-ranking groups`**, press F1; another menu will appear. This time move to **`A higher ranking group: counting from document level`**. Press F1 again. The resulting menu will ask you to which level you wish to move down. Here you can press F1 as 0 is the correct value for this example. This should return you to the path definition menu. Then press F2 to select a group. Select inventory from this menu by pressing F1. Then press F2 again and select total. Then press F3 (to select an element) and select value. Press F1 to activate this. This should return you to the path menu. It should now look like this:



```
Activate the path definition menu

Step          : /root[0]
Step          : /inventory
Step          : /total
Step          : :value
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
Step          :
        Subpath 1

Describe an access operation
```

This should complete the definition of the first path that we want to have output. There are two others.

Press F7 to define a second path. There are three stages to the second path:

- Press F4 to select a group function. Then select a higher ranking group from the current level. Going back one level will be sufficient (so you can just press F1 to accept this).
- Press F2 to insert a group. Select p.
- Press F3 to insert an element. Select :surname.

The third path is even easier to define. Press F7 to add a path, then press F3 to insert an element to this path. Select :firstname.

You should now have successfully defined three paths. Press F1 to return to the search menu. The path field should now contain the following entry.

```
/root[0]/inventory/total:value,/back[1]/p:sur...
```

Now move the cursor to the display field and toggle (with the right and left arrow keys) until this field contains the term **`both`**. Then press F1.

This should bring you to the results of your task. The first one is not particularly revealing as it looks as though Thomas Stubbington left no money to the church. However, if you press PgDn to move you to the next result, you should get the following output:

```
datasource (2 = "2") :
Inp[r]imis first I geve to the p[ar]ishe Churche of St Johns xiid. lykwise to the poore of the same
p[ar]ishe xiid.

*******************************************************

datasource (2 = "2") : value            13234.000000
datasource (2 = "2") : surname          Stobbington
datasource (2 = "2") : firstname        Thomas
```

This tells us that Stubbington left twelve pence to the parish church of St John and left a total of over thirteen thousand pence (i.e. a total of over 55 pounds).

You may find it interesting to look through the result to see if there is any correlation between the value of goods left and the amount left to individual churches.

Exercise 9.1

Try to find out if there are any discrepancies between gifts of made out of silver in wills and items made of silver mentioned in the inventories of individuals. For example, in her will Mary Fielder leaves a "silver Cupp" to her nephew William Blunt. See if you can find out whether this item is in the inventory of her goods and chattels.

As this exercise is open-ended, no answer is given! However, it is best solved using both the menu driven system and the command system.

Chapter 10

Codebooks

10.1 Introduction

A codebook in κλειω is a 'translation' table which makes it possible to translate any number of alphanumeric identifiers, e.g. bricklayer, female or plough, into any number of numeric values. The term codebook in κλειω might be defined as a look-up table or a coding table. This form of construction is vitally important in historical computing, and this is just as true when one is using κλειω. Frequent mention has been made in this volume of κλειω as a source-oriented data processor, which we have said implies that the source material is to be input in a form which is as close as possible to the original. The debate in historical computing about the pre- and post- coding of historical material has run for as long as historical computing has been a discipline. The main argument in favour of post-coding is that an individual's interpretation of a category (say, occupation) might change over time, and if material had been coded before data had been entered it would be very difficult to uncode it. (This is not the place to describe this debate; suffice it to say that there is a large range of literature on the subject, for which we refer you to the further reading at the end of this chapter.) However, all historians would probably agree that some form of categorisation of material is necessary to perform effective subsequent analysis. For instance, consider the probate database we have been using. If we wanted to compare the contents of one person's house with that of another, the results would not be satisfactory if we kept strictly to the original terms.

In the case of the probate database there are a large number of different items. If we wanted to learn something about the total value of a class of object owned by each of the individuals in the database, say for example agricultural equipment, we would have to search through the database, probably using the menu-driven system to find all occurrences of agricultural equipment in the inventories and calculate by hand the total value of the items we considered to fall into our category of agricultural equipment for each person. Using a codebook would relieve us of having to do that.

This chapter falls into two parts. The first will describe some of the simpler principles of creating codebooks, while the second will demonstrate the possibilities of outputting material from codebooks into a format that a statistical package can understand. As we have seen, κλειω has not been designed to handle complex statistical material itself, but it does provide an easy way of producing statistical material and outputting it in a format that can be understood by the two most popular social science statistical packages.

10.2 Creating a codebook

There are two methods to create a codebook. The first and simplest is to create one implicitly using the **`create`** command. This method will be described first. The other method is to create a codebook explicitly by defining a logical object which describes the contents of a codebook. The second method is the more usual method especially where there are a large number of items or codes to be entered into the codebook.

Creating a codebook is very similar to creating a catalogue. As they are both logical objects (and are handled separately), it is possible for both a codebook and a catalogue relating to the same database to have the same name. In this case we will create a codebook using the same items from the probate database that we used to make a catalogue. This codebook will be called objects.

When you run this ensure that you send the results to a file as it will be over 6700 lines long. When this task has run successfully, a codebook called objects will have been created.

Example 10.1

```
query name=probate;part=:item
create name=objects;part=:item;repeat=yes;write=yes
stop
```

The **`create`** command is used to generate codebooks. It must be followed by a **`name=`** parameter which takes as a value a user-defined name for the codebook. In this example the name of the codebook is objects. The **`create`** command must also be followed by a **`part=`** parameter which expects a list of paths as a value. Usually this is just a single element, and in this case it takes the element item from the probate database.

For the time being the other two parameters in this command can be ignored; they will both be explained shortly. Neither is needed to create a codebook; they are inserted here to modify the result of this task to make it easier to understand some of the principles of codebooks.

The first part of the result produces information in a similar format to this:

```
andirons                    1 16 (3 times) 18 19 (2 times)
                            22 26 (5 times) 27 33 (3 times) 5 6
(19 times)
```

This means that the term andirons occurs once in document 1, three times in document number 16, once in document number 18 (and in numbers 22, 27, 5 and 6), twice in document number 19, five times in document number 26 and again three times in document number 33. The word andirons occurs a total of 19 times in the database as an item. The layout of this result is caused by the **`write=yes`** parameter which asks the system to produce a report of what is in the codebook. Scroll down the result file (or use the search facility on the word kleio in your screen editor) to see the result of the **`repeat=yes`** parameter, which asks κλειω to produce a description of the codebook's contents with the codes allocated by κλειω. The result should look like this:

```
                                            kleio      system

a parrell                                   10.00        *
a white horse of wood                       20.00        *
alblades                                    30.00        *
all the lumber                              40.00        *
alle halfes                                 50.00        *
```

This demonstrates graphically exactly what a codebook is. It is a table with one column containing a term from the source and a number of columns containing coding variables. This means that in each column there is an independent coding system. In this example there is only one coding system in operation — that defined by κλειω, which relates to the column headed **`kleio`**. The other possible coding variable is called **`system`** (which currently contains no values). κλειω allows the user to define as many different coding systems as necessary for any selection of data. This could be particularly useful when using a coding system to define occupations. When dealing with occupations it might be useful to code both the occupational classification and social class (two different dimensions).

There are further parameters available to use with the **`create`** command. The most important is **`overwrite=`**, which allows you to tell κλειω to overwrite an existing codebook.

10.3 Searching the codebook

Try running the following task which will produce all the information about items starting with the word brewing (i.e. items with a numerical code between 2210 and 2270 inclusive).

Example 10.2

```
query name=probate;part=:codebook[:item,kleio,objects]=
   "2210" greater equal and "2270" less equal
write part=:item,:codebook[:item,kleio,objects],
   :form["next item"],:lines[2]
stop
```

This task introduces the element function **:codebook[]**, which is essentially a function to convert numerical information into textual information. It allows one to access textual information in a database using the numerical codes defined in the codebook.

The function is usually defined as follows:

```
part=:codebook[:item,kleio,objects]
                    /      |       \
              element    name of    name of
          specification  coding     codebook
                         system
```

If this parameter were used in a task it would produce all the items from the database that are in the codebook. Particular items can be specified by adding a condition to this function.

```
part=:codebook[:item,kleio,objects]="340"
```

would return all the items from the database that have the numerical code of 340 in the coding variable called **kleio**.

Returning to the task above this may be translated as follows:

query name=probate;
I am interested in a database called probate.
part=:codebook[:item,kleio,objects]="2210" greater equal and "2270" less equal
I am only interested in that part of the database where elements called item within a codebook called objects have a numerical code of between 2210 and 2270 in the coding variable called **kleio**.

```
write part=:item,:codebook[:item,kleio,objects],
   :form["next item"],:lines[2]
```

I like would the information found in the previous command to be displayed on the screen in the following order. Firstly, the contents of the element, secondly the code that relates to that entry (from the **kleio** coding system) and then the text "next item", which should be followed by two blank lines.

```
stop
```

There is an important piece of syntax in this task, which if used incorrectly will cause many errors. Notice that the numerical codes used in the conditions following the **:codebook[]** function are enclosed within quotation marks. This is because κλειω accepts the use of alphabetical characters to form a number; for example, a coding value of 123b would be acceptable. Without quotation marks κλειω would not have known where the number ended.

However, in cases like the task in Example 10.2, using the group function **keyword[]** with a catalogue is much quicker than using the element function **:codebook[]**. It is advisable only to use **:codebook[]** when analysing the whole database (or at least large parts of it) and the **keyword[]** function on just a few different items. However, the use of a catalogue and the function **keyword[]** with the **index** command is no quicker.

Using the **create** command is the first and simplest method of creating a codebook. However, using this command we are unable to allocate our own codes to different groups of items. This will be achieved in the following section. It is possible, however, to create a codebook using this subsequent material rather than using the **create** command.

For the rest of this chapter, we will be using a new database, 'colebro', which is almost identical to the censsamp database used in earlier chapters. This database contains 'real' data, from a street in Winchester. The source of the data is the 1881 Census. Before you will be able to complete these exercises you will have to compile the database. It can be found on the tutorial disk. This files needed are colebro.mod, colebro.loc and colebro.dat. These three files *must* be compiled in this order or you will find that there will be errors. (The purpose of colebro.loc will be explained in Chapter 14.)

Exercise 10.1

Using the colebro database and any occupational classification scheme you like create a codebook which uses the element occupation. The answers given at the end of this volume are based on a scheme created by Tillott which can be found in the appendix to this chapter. K. Schürer and H. Diederiks (eds.), *The Use of Occupations in Historical Analysis*. Halbgraue Reihe zur historischen Fachinformatik, A19 (St. Katharinen, 1993) contains a useful table of occupational codes for the 1851 Census (and successive censuses to 1911).

10.4 Adding codes to the codebook

Adding codes to a codebook is an easy task. κλειω allows the user to create most of a logical object to create a codebook while the user only needs to add specific coding. The following task 'makes' much of the codebook. All that you have to do afterwards is assign the codes to each of the items in the codebook and add this logical object to the specific environment. This task saves a considerable amount of time in preparing a logical object to define a codebook.

The following task is laid out as below in order to help clarify explanation. Ensure that you send the result of this task to a file for later use.

Example 10.3

```
options lines=0
query name=probate;part=:item
index part=:form['form text="'];
   limit="";
   part=:item;
   limit='";number=999';
   maximum=1;
   signs=39;
   identification=order[];
   write=no
stop
```

The commands in this task can be identified as producing the following results:

`options lines=0`
: Tells κλειω not to put any spaces between each line of output.

`query name=probate;part=:item`
: I am interested in the database probate and in particular to the element item.

`index part=:form['form text="']`
: Create an index which starts with the text `form text="`

`limit="";`
: followed by no spaces (otherwise κλειω would produce a space before writing the entry found in the item element),

`part=:item;`
: followed by the name of the item found,

`limit='";number=999';`
: followed by the text `"number=999`

`maximum=1;`
: ensuring that each element item is only put into the index once,

`signs=39;`
: finally ensuring that there are only 39 characters in the text surrounded by quotation marks. (This is because κλειω allows a maximum of 39 characters for the name of a term in a codebook.)

identification=order[];
Give the ordinal number of the standard identification given by κλειω,
write=no
but do not display it in the result. (The combination of this and the previous parameter is a quick way of telling κλειω not to display the identification number for any particular entry chosen in the **part=** parameter.)
stop

Part of the result of this task should look like this:

```
form text="a parrell                    ";number=999
form text="a white horse of wood        ";number=999
form text="Alblades                     ";number=999
```

All this has done is to produce an index of all the items specified from the database. This has not created a codebook yet!

If one removes all the description given by κλειω at the beginning and end of this result file, and adds the following lines at the top:

```
item name=objects;usage=codebook;source=probate;type=permanent
part name=system;type=insert
```

and the following line at the bottom:

```
exit name=objects
```

a logical object to define the codebook would be virtually complete. Unfortunately, for this example a great deal of work would be necessary to code all these 6700-odd items into categories to exploit the potential of the codebook fully. Before we add a selection of codes to this codebook to demonstrate how to query it, it is worth describing what the lines above and below the index of items mean.

item name=objects;
Using the logical object called objects,
usage=codebook;
which is a codebook,
source=probate;
related to the database probate.
type=permanent
Make this (modified) logical object part of the permanent environment of the database specified in the **source=** parameter.
part name=system;
For the part of the coding variable called ‘system’,
type=insert
prepare to insert new codes into this part of the codebook.

Once you understand what this is going to do, it will be clear that some further changes will be necessary to make this codebook work properly.

Go through the file searching manually for all the terms that relate to brewing. E.g. "beare", "beere", "barlie", "malte" and all terms starting "brewing(e)", "brueing", "bruing(e)" and for each of these terms change the number from 999 to 1000.

As these lines form part of the description of the logical object, we can see that all the items that refer to brewing have been given the code (in the coding variable system) 1000. κλειω interprets the lines as follows:

`form text="beere";`
where the name of the item is "beere"

`number=1000`
add the code 1000 (to that part of the codebook called system: see above).

Once you have made all the changes necessary to this file, save it and run it. If your version does not work, attempt to correct it; if that fails, there is a correctly set up file on the tutorial disk called ex10.4.

In fact it is perfectly possible to create a codebook using just a logical object, i.e. without using the **`create`** command. However if you were to do this, the **`kleio`** variable would be absent from it (unless of course you were to define it yourself). (You would also need to formulate your logical object slightly differently, for example using **`type=create`** rather than **`type=insert`**, as there are no items in the codebook into which you could insert codes.)

10.5 The `describe` command

The result of the previous command was rather unspectacular, so it may be worth checking whether κλειω has performed this task as you specified. κλειω provides a useful command to help you see what lies within the environment. Run the following task:

```
describe name=objects;usage=codebook;source=probate;
  type=permanent
stop
```

Here κλειω is asked by the **`describe`** command to describe a logical object which relates to a particular database. (We have met this before: see pp. 115-17, 138 above.) The rules pertaining to what is described can be found in the Reference Manual, Section 7.3.3; for the time being just this command will be discussed. In English the command could be translated as 'Describe the logical object known as objects. This logical object is a codebook, and it is part of the permanent logical environment of the database called probate'.

In this case what is displayed is the codebook, with all the current codebook variables for the terms in the list displayed in an independent column. Some of the result should look like this:

```
                          kleio       system

bedsteede               1150.00       999.00
bedsteeds               1160.00       999.00
beefe picker            1170.00       999.00
beefe pricker           1180.00       999.00
beere                   1190.00      1000.00
bell salt               1200.00       999.00
bell skellets           1210.00       999.00
```

Notice that the system code for "beere" has changed from 999 to 1000. As mentioned above, it would however be usual to define codes for all the items within the codebook. This is just a demonstration of how κλειω adds codes to a codebook.

10.6 Codebooks and interactive methods of searching

When using a codebook one would not just make codes for one set of items; one would create a coding system that applied to *all* the terms within the database in order to process it fully. In this case the items could be coded to reflect the function of the object, e.g. bedding, kitchenware and cooking apparatus, agricultural equipment, clothing, books, debts, shop goods and so on. All the items in the codebook would be coded and thus prepared for subsequent analysis. If one were only interested in the brewing equipment it would be easier (and more flexible) to interactively process a catalogue made up of all the items in a database using the menu-driven system, in which experimentation might take place.

This interactive processing with the menu-driven system may also be a pre-coding stage in which checks are made to see whether the words mean the same thing.

However, if you know exactly what you want to find out from a database, it may be worthwhile to use a codebook. Creating a codebook would also be the appropriate route to follow if data needs to be prepared for statistical analysis.

The following table shows the main differences between the codebooks and catalogues:

Codebook	**Interactive system**
Rigid	Allows experimentation
Relatively slow	Quick for small analysis
Use with all data in a database	Use with samples of data
Presumes clear-cut knowledge of type of task	Allows for considerable experimentation
Not useful for full-text	Good with full-text
Allows for output for subsequent quantitative analysis	Allows some output (but not processed)

10.7 More querying of codebooks

A further example of querying a codebook is shown below. It is rather similar to the task displayed above, but we hope it will clarify matters. The following task would be used to produce the element **`item`** from the catalogue objects using the code rather than the textual name.

Example 10.5

```
query name=probate;part=:codebook[:item,system,objects]="1000"
write part=:item,:codebook[:item,system,objects],
     :form["next item"],:lines[1]
stop
```

This task does in fact give slightly more information that one would usually want, but asking for the information in this way shows what κλειω is doing in order to retrieve the results. The results of the task look rather like this:

```
datasource (2 = "2") : item          barlie
1000.000000
next item

datasource (2 = "2") : item          beere
1000.000000
next item
```

As the task used to produce this result is a little complicated, it is worth describing what each line means:

`query name=probate;`
: I am interested in a database called probate.

`part=:codebook[:item,system,objects]="1000"`
: I am only interested in that part of the database where the elements called **`item`** within a codebook called objects have a code of 1000 in the coding variable called **`system`**.

```
write part=:item,:codebook[:item,system,objects],
   :form["next item"],:lines[1]
```
: I would like the following information, once retrieved, to be displayed on the screen in the following order. First, the name of the item selected by the previous task, secondly the code that relates to that entry in the database, thirdly the text "next item" and finally please leave one blank line between each item displayed.

`stop`

Codebooks can be very useful as the following queries show, but as will have been seen from the previous task they are very slow even when used in conjunction with a catalogue.

10.8 Adding more codes to a codebook

It would be possible to add codes to other items in the codebook separately. For instance, if we had missed the term "mollt" as a variant spelling of malt, it would be possible to add this code in a similar way:

Example 10.6

```
item name=objects;usage=codebook;source=probate;type=permanent
part name=system;type=substitute
form text="mollt";number=1000
exit name=objects
```

Notice that the parameter value for the **`type=`** parameter has changed from **`insert`** to **`substitute`**. This is because there is already a value for the term "mollt" within the codebook.

Now run example 10.5 again.

10.9 Creating labels for codes

Once a coding scheme has been decided upon it is possible to give labels to the groups of codes. If in this case we wanted all items with the code 1000 in the coding variable called ‘system’ to be related to the group "brewing equipment", this would be achieved through the use of the following logical object:

Example 10.7

```
item name=objects;usage=codebook;source=probate;type=permanent
part name=system;type=insert
write number=1000;text="brewing equipment"
exit name=objects
```

This command assigns the text "brewing equipment" to all those items which have the code 1000. It would of course be possible to make other values relate to the term brewing as well by adding further **write** directives. This is generally used to make codes more understandable and easier to use.

Exercise 10.2

Using the codebook created from the colebro database, add labels for the occupational categories that you have defined.

10.10 Querying a codebook using labels

The following task demonstrates the method of obtaining information from the codebook with a textual code:

Example 10.8

```
query name=probate;part=:codebook[:item,system/text,objects]=
   "brewing equipment"
write part=:item,:codebook[:item,system,objects],
   :form["next item"],:lines[3]
stop
```

Rather than specifying that the code for the items of brewing equipment is 1000, it is possible to search for the name of that code. The only differences between the task in this section and the task in Section 10.5 are, first, that the part of the codebook specified in the

`part=` parameter is equal to the text string "brewing equipment" rather than the code 1000, and, secondly, that the second parameter value for the **`:codebook[]`** function, **`system/text`**, now defines the system code as text (rather than the default which is a number). This is because coding was traditionally done, and is still normally done, with figures rather than text.

Also available on the tutorial disk is a glossary of words found in the element :item. This has been constructed using the following three files. (These three files are respectively called ex10.9, ex10.10 and ex10.11.) The fourth example reproduced here demonstrates how one can obtain a formatted result of the contents of the glossary.

Example 10.9

```
item name=objects;source=probate;type=permanent;usage=codebook
part name=glossary;type=create
form text="a parrell";number=1
form text="Alblades";number=2
form text="Alle halfes";number=0
.
.
.
exit name=objects
```

The task above creates a further variable in the codebook objects. This new variable is called glossary. All the items have been allocated a code number; those items with different spellings have been allocated the same code.

Example 10.10

```
item name=objects;source=probate;type=permanent;usage=codebook
part name=glossary;type=insert
write number=1;text="Apparel. Clothes."
write number=2;text="Awl blades. The pointed ends of awls(?)."
write number=3;text="Awl. A small instrument for piercing
   holes. Probably used for book-binding."
.
.
.
exit name=objects
```

The task above contains all the definitions of all the codes in the task ex10.9. This task tells κλειω to insert into specific variables a label. This label is used in a way different from the one demonstrated above. Here it contains the definitions the items found in the database.

Example 10.11

```
query name=probate;part=:item="aquacomposita"
index part=:item;
   maximum=1;
   limit="\n";
   part=:codebook[:item,glossary/text,objects]
stop
```

The task above demonstrates one way of getting the information out of the codebook. κλειω is asked to find an item in the database which contains the string "aquacomposita"; when it has found it, it is asked to display it (once), and indent the following output. (For an explanation of '**\n**' see the next chapter.) Then display the contents of the label from the codebook which relates to the code for the string aquacomposita. The result should look something like this.

```
aquacomposita
            Aqua Composita. A drink made with a 'composite' or  variety of ingredients. E.g.'Take twelve
            drops of Oyl of Cloves, eight Oyl of Nutmegs, & five of Oyl of Cinamon. Put them into a large
            strong drinking glass, & mingle well with them two ounces of the purest double refined sugar
            in powder' (Eleanor Sinclair Rhode i, p.219 from 'The Closet of Sir Kenneth Digby Opened',
            1699). 8
```

10.11 The statistical interface

κλειω's statistical capabilities are, we have said, rather limited. The built-in statistical command **cumulate** only provides the bare minimum of information about data, while the parameter **type=count** with the **index** command provides even less. However, the combination of these two actions provide most of the statistical information that an ordinary historian might want to use. On the other hand κλειω provides the capability to convert material from a κλειω database into a format that can be understood by a number of different statistical packages. The decision not to create many commands in κλειω for statistical analysis of historical data was made because of the widespread availability of statistical packages. Currently κλειω can easily interface with three main programs: SPSS-X, SPSS-PC and SAS. There is also a useful interface between κλειω and the database management system, CensSys. CensSys was developed at the University of Bergen in Norway and designed to analyse census material but is equally well suited for analysis of other material. At the time of writing CensSys is being translated into English but will only accept material created with the 'original' version of κλειω (i.e. with Latin commands). (This should change shortly.)

For the three main statistical programs κλειω produces two components of a file; a file containing the basic data and a file containing the system files, which tell the particular statistics program how to interpret that data.

κλειω uses two commands to produce these two components. These two commands are **`case`** and **`translation`**. As with most of the other commands that κλειω uses, there are a number of relatively simple parameters which change default rules about the conversion of data. In this volume we shall not be demonstrating all of these parameters but giving an overview of the ways in which κλειω performs this task and suggesting where to look in the Reference Manual for further information.

The two commands which we mentioned above concern two aspects of the conversion procedure. One tells κλειω what do with the material once the other has converted it.

The **`translation`** command tells κλειω where to send the information once κλειω has converted it into a format that can be understood by statistical software, and how any of the default rules that govern this conversion may be changed.

The **`case`** command tells κλειω which information to include in the conversion process, how that material is to be converted, and what variables are to be created.

Before considering how κλειω converts data to a format that statistical software can understand, we should point out again that whenever κλειω is asked to process a task concerning the contents of a database, the **`query`** command is used. This selects part of a database for processing. This is the only command that selects part of a database for processing (except **`confirm`** and **`negate`**). All other commands process the information that the **`query`** command has selected. Just as we use the **`write`** command to print out selected information, or the **`index`** command to create sorted lists of selected information, we use the **`case`** command to tell κλειω how to convert selected information into statistical data records and the **`translation`** command to tell κλειω where to direct the results of the **`case`** command. Thus tasks which are designed to create statistical data records follow the structure of all other basic commands.

In this part of this chapter we are going to demonstrate how to convert some simple material from the probate database into statistical data records. To do this we will be using a similar codebook to the one we created in the first half of this chapter. In this case each of the items has been coded. The file needed to create this codebook is called ex10.12. Look at this file. It will make a codebook called objects2. If you scroll down the file you will see that the coding of these items has not been performed imaginatively; in other words, it will not produce 'real' historical results.

10.12 The `translation` command

Before we discuss the **`case`** command, which actually makes the statistical cases that SPSS or SAS need to run our data, we should first look at the **`translation`** command, which tells κλειω what to do with that data. By default you do not need a **`translation`** command, as all parameters have a default setting. As the **`translation`** command can also define the default settings for the **`case`** command, it is worth looking at a few of these parameters.

First, we should define the task that we hope to solve. We are going to use the probate database to see if there is any correlation between the class of object and its value. (Remember that this is only hypothetical, as the classes of objects have not been coded with accuracy in mind, but only for demonstration purposes.) Thus:

```
query name=probate;part=:item
translation target=pcspss;first="objects2.dat";
  second="objects2.sps"
```

If you omitted the parameters following this **`translation`** command, κλειω would convert the data we will subsequently define into SPSS-X format (which is slightly different to pcspss format), and send the data to a file called probate.num and the definition language to a file called probate.dic, which you may not want.

10.12.1 The `target=` parameter

This parameter accepts the keywords **`spss`**, **`pcspss`**, **`sas`**, **`censsys`** and **`null`**. This keyword takes the name of the statistical program that will eventually be used to analyse the data. By default κλειω assumes that the package to be used is SPSS-X. The keyword **`null`** suppresses the creation of a command file.

10.12.2 The `first=` and `second=` parameters

These two parameters take as values the names of files into which the information generated by the **`case`** command is put. The **`first=`** parameter takes the name of the file where the raw data goes and the **`second=`** parameter takes the name of the file where the definition language goes. By default κλειω gives the **`first=`** parameter the name of the database followed by the suffix .num, and the **`second=`** parameter takes the name of the database followed by the suffix .sps.

Since we are going to run this command more than once in this example it would be wise to add a further parameter, **`overwrite=`**, to this command, thus:

```
query name=probate;part=:item
translation target=pcspss;first="objects2.dat";
  second="objects2.sps";overwrite=yes
```

There are three further parameters which may be useful. One allows to you ask κλειω not to produce a report on what it is doing (**`write=no`**), while two others allow you to override certain errors that may occur in the data.

Six further possible parameters modify the default settings of the **`case`** command and need not detain us here. Details of these parameters can be found in the Reference Manual, Section 8.3.6.2.

10.13 The `case` command

Each **`case`** command creates one type of case and each **`part=`** parameter creates a number of statistical variables. In the example below the task produces one case and two variables; the 'category of object' code for each item, and the value of that item.

```
query name=probate;part=:item
translation target=pcspss;first="objects2.dat";
  second="objects2.sps";overwrite=yes
case part=:codebook[:item,system,objects2];
  part=:value
stop
```

If this task is run part of the result will look like this extract (taken from the file objects.num):

```
868   6.000    16.000
869   1.000   156.000
870   2.000    80.000
871   9.000    28.000
872   1.000    16.000
873   1.000    36.000
874   9.000    60.000
```

The first figure here is the case number (which is automatically generated), the second is the code in the coding variable called system, and the third is the value of that particular item.

By default κλειω gives the name in the **part=** parameter to the name of the variable. In this example the first variable would be given the name **:codebook[........]**. This might be rather cumbersome to work with so we use two parameters to change this default action. We can use the parameter **name=** to define what we would like to call the variable, and we can also use the parameter **write=** to change the description of the variable.

Example 10.13

```
query name=probate;part=:item
translation target=pcspss;first="objects2.dat";
   second="objects2.sps";overwrite=yes
case part=:codebook[:item,system,objects2];name=item;
   write="code";
   part=:value;name=value;write="value"
stop
```

This should make it easier to see what is going on here (especially when using SPSS).

What we have not said here is that when κλειω comes across the first **part=** parameter after a **case** command, it checks to see whether there is valid information for this particular record. (Incidentally the same thing happens with the **index** command.) If the entry in the database is empty, κλειω will not make a statistical case for the rest of the entry. For this example there are no elements which do not contain an item. However in the colebro database there are people without occupations. They thus do not have jobs that can be converted into a coding variable; so κλειω is unable to create a statistical case for that individual.

Exercise 10.3

Using the occupational codebook from the colebro database, create two files called jobs.num and jobs.sps which contain the code for the job in the first column and the age of the individual in the second column.

Once you have successfully created these files, look at the file jobs.num. Notice that the number of cases is considerably less than the number of individuals mentioned in the database. This is because a number of people in this database do not have an occupation, and therefore κλειω ignores them. There are two ways around this problem. First, a **:form[]** function could be added, as follows:

```
case part=:form["0.0",number];
    part=:codebook[:item,system,objects2];
.
.
.
```

which just adds an empty value, and would result in a third variable in the .num file. A possible result might look like this:

```
1        0.0        12.000   345.0000
```

The other solution would be to choose an element which occurs more frequently, as the element in the first **`part=`** parameter following a **`case`** command. In this database every person has an age, while some people do not have an occupation. In this case, therefore, we could reverse the two **`part=`** parameters to solve the problem. When you have succeeded, look at the job.num file and see how many cases have been made.

There are a number of problems inherent in creating statistical material. These do not just occur when one is using a κλειω database, but there are problems which are specifically brought up by the way in which κλειω allows you to input data. In the case of the occupations in the colebro database, some people have more than one. You might assume that κλειω would take both occupations into account when creating stastical cases. However, it does not. You could specify how many entries in an element to select using the **`maximum=`** parameter with the **`case`** command. (N.B. the **`:collect[]`** function (see Section 5.6.4 of this book) might also provide useful results.)

Summary

The last two chapters of this volume have carefully considered two particularly valuable tools which deal with a number of problems inherent within historical data. Both of the tools are particularly able to help in the interpretation of spelling variation.

Catalogues have been used to access and display full-text material and to assign keywords to groups of textual information. Codebooks have been used to demonstrate κλειω's potential in systematically coding items of data and to prepare that data for statistical analysis.

There is a third important tool which deals with spelling variations in historical data and in particular with the names of individuals. Within κλειω there are a number of algorithms that can be used to solve certain problems connected with the identification of people in the past. This is part of a process known as nominal record linkage. Chapter 12 will be devoted to the use of this process for historical data.

Appendix: Occupational and social categories

Based on a mimeographed sheet attached to 'The Censuses of Tickhill in 1851 and 1861' by Peter Tilliott, University of Sheffield, Department of Extramural Studies, 1969.

Occupational groupings

1. AGRICULTURAL SELF-EMPLOYED OR MANAGERS

   Farmer, farmer's son, farm bailiff, market gardener, seedsman (with land).

2a. SKILLED AGRICULTURAL WORKERS

   Drill machine labourer, foreman agricultural labourer, horse breaker, stable man, gardner (self-employed).

2b. AGRICULTURAL LABOURERS

   Farm labourer, farm servant, cottager, field worker.

3. SHOPKEEPERS, TRADERS, PETTY ENTREPRENEURS (not employing more than five people)

   Foodstuffs, retail: baker, butcher, confectioner, fishmonger, flour seller, fruit dealer, grocer, milk dealer or cow keeper.

   Foodstuffs, wholesale: cattle dealer/jobber, corn dealer, cattle drover, miller/maltster, fell-monger, pig jobber, potatoe merchant.

   Clothing: draper, hatter, lace and cap dealer, linen draper, woolen draper, worsted dealer.

   Victuallers, wines, tobacco: licensed victualler, lodging house keeper, innkeeper, wine and spirit merchant, brewer, tobacconist, beer house keeper, publican.

   Carriers: boat proprietor, boatman, waterman (but all in group 11 if not carrying for trade), carrier and general dealer, coal carrier/merchant.

   Miscellaneous: hawker, higgler, huckster, ironmonger, tallow chandler, glass and china dealer.

4. SKILLED CRAFTS, NON-INDUSTRIAL (where such craftsmen seem to be operating in a factory, however small, they should be in group 6)

Metal working and machinery: agricultural implements/machine maker, brazier, nail maker, blacksmith, general smith, whitesmith, tinner.

Building: bricklayer/maker, carpenter/cabinet maker, mason, painter, plasterer, sawyer, stone mason, paviour, plumber.

Clothing: cap maker, clogger, corset maker, framework knitter, hand loom weaver, milliner/dressmaker, tailor, shoemaker.

Leather: currier, harness maker, saddler, tanner, cordwainer.

Miscellaneous: basket maker, blacking maker, boat builder, clock/watchmaker, compositor, printer, cooper, jeweller, roper, sackmaker, upholsterer.

5. MANUFACTURERS, INDUSTRIALISTS, WHOLESALERS OR MANAGERS OF LARGE ENTERPRISES (usually employers of labour on a large scale but varying according to the nature of the industry)

Cotton manufacturer, steel manufacturer, timber merchant, worsted spinner.

6. SKILLED INDUSTRIAL CRAFTSMEN

Cotton: carder, bobbin weaver, overlooker, picker, twister and drawer, piecer, minder, ruler, winder, weaver, rover, stripper, slubber, warp sizer, tatler.

Wool: comber, comb maker, spinner, steam loom weaver, siser.

Iron: moulder, founder, puddler

Miscellaneous: mill hand, operative, engine tender, mechanic.

7a. UPPER PROFESSIONAL

Accountant, banker, architect, lawyer, attorney, clergyman, minister of religion, curate, Methodist minister, doctor, surgeon, physician, judge, land agent, solicitor, surveyor.

7b. PROFESSIONAL

Governess, auctioneer, Inland Revenue officer, musician, artist, Primitive Methodist preacher, school master/mistress, teacher, tutoress, veterinary surgeon.

8. CLERICAL (not supervisory)

Solicitor's clerk, auctioneer's clerk, rail clerk, parish clerk, assistant secretary to savings bank.

9a. UPPER SERVANTS

Butler, companion, cook, gamekeeper, gentleman's servant, gardener, housekeeper, lady's maid, nurse.

9b. GENERAL DOMESTIC SERVANTS

Coachman, footman, gardener's assistant, general servant, groom, housemaid, kitchen maid, nurse girl, nursery maid.

9c. LOWER SERVANTS

Charwoman, laundress, manglewoman, washerwoman, servant boy.

10a. PRIVATE INCOME RECIPIENT

Fund holder, interest of money, gentleman, railway proprietor.

10b. RENTIERS

Proprietor of land/houses.

10c. ANNUITANTS

11. SEMI-SKILLED AND SERVICE WORKERS

Bellman, cab driver, boatman (unless trading see group 3), sexton, mariner, sailor, ferryman, letter carrier, postman, post messenger, toll keeper, canal lock keeper, ostler, hotel waiter, midwife, horse doctor, castrator, farrier, chimney sweep, soldiers and armed force (non-commissioned)

12. UNSKILLED WORKERS AND LABOURERS

Labourer in iron works/railway/brickyard/at factory, porter on railway/for liquor merchant, working on highway, errand boy, factory boy.

13. SUPERVISORY WORKERS

Workhouse master/mistress, postmistress, rail inspector, superintendant constable, canal agent.

14. CHILDREN (of 14 years and under AND scholars of all ages)

15. HOUSEWIVES

16. NO OCCUPATION

17. PAUPERS

    Tramp, poor man, beggar, parish relief/pay, unemployed, almspeople.

18. RETIRED PEOPLE (including superannuated and pensioners)

19. VISITORS.

Social groupings: consolidated classes

I Independent means; upper professional; managerial; manufacturers; large enterprise: GROUPS 5, 7a, 10a, 10b.

II Lower Professional; annuitants; supervisory; large shopkeepers: GROUPS 7b, 10c, 13.

III Agricultural self-employed; farmers; market gardeners; agricultural managers: GROUP 1.

IV Shopkeepers; craftsmen; clerks; skilled agricultural workers; higher servants: GROUPS 2a, 3, 4, 6, 8, 9a.

V Agricultural labourers; semi-skilled and service workers: GROUPS 2b, 11.

VI Domestic servants: GROUP 9b.

VII Labourers and unskilled workers: GROUPS 9c, 12.

VIII Others: GROUPS 14, 15, 16, 17, 18, 19.

Further reading

W. A. Armstrong, 'The Use of Information about Occupation', in *Nineteenth-Century Society. Essays in the Use of Quantitative Methods for the Study of Social Data*, ed. E. A. Wrigley (Cambridge University Press, Cambridge, 1972), pp. 191–310.

G. Bouchard, 'The Saguenay Population Register and the Processing of Occupational Data: an Overview of the Methodology', *Historical Social Research*, 32 (1984), pp. 37–58.

J. H. Goldthorpe *et al.*, *Social Mobility and Class Structure in Modern Britain* (London, 1980).

D. I. Greenstein, 'Standard, Meta-Standard: a Framework for Coding Occupational Data', *Historical Social Research*, 16:1 (1991), pp. 3–22.

W. Hubbard and K. Jarausch, 'Occupation and Social Structure in Modern Central Europe: Some Reflections on Coding Professions', *Quantum Information*, 11 (July 1979), pp. 10–19.

M. B. Katz, 'Occupational Classification in History', *Journal of Interdisciplinary History*, 3 (1973), pp. 63–99.

R. J. Morris, 'Occupational Coding: Principles and Examples', *Historical Social Research*, 15:1 (1990), pp. 3–29.

K. Schürer, 'The Historical Researcher and Codes: Master and Slave or Slave and Master?', in *History and Computing III*, ed. E. Mawdsley *et al.*, (Manchester University Press, Manchester, 1990), pp. 74–82.

K. Schürer and H. Diederiks (eds.), *The Use of Occupations in Historical Analysis*. Halbgraue Reihe zur historischen Fachinformatik, A19 (St. Katharinen, 1993).

Chapter 11

Formatting results

11.1 Introduction

In Chapter 5 we demonstrated a number of functions which could be used to make results look slightly nicer. This chapter aims to demonstrate briefly a number of other methods of formatting the results of a query in a more interesting manner.

In the previous chapter we touched (p. 194) on one of the three available print constants ("**\n**"). We also demonstrated the use of the **:form[]** function. In this chapter, we shall demonstrate the use of all these functions together.

11.2 Print constants

Currently there are three print constants, **\n**, **\t** and **\f**. These three constants tell κλειω how to output your results. These constants are used with the **limit=** parameter. In ex10.11, we demonstrated the use of the **\n** constant. This constant tells κλειω to advance to the beginning of the next line.

```
query name=probate;part=:item="aquacomposita"
index part=:item;
  maximum=1;
  limit="\n";
  part=:codebook[:item,glossary/text,objects]
stop
```

produces a result like this (if sent to a result file):

```
aquacomposita
        Aqua Composita. A drink made with a 'composite' or variety of ingredients. e.g. 'Take
        twelve drops of Oyl of Cloves, eight Oyl of Nutmegs, & five of Oyl of Cinamon. Put them
        into a large strong drinking glass, & mingle well with them two ounces of the purest double
        refined sugar in powder' (Eleanor Sinclair Rhode i, p.219 from 'The Closet of Sir Kenneth
        Digby Opened', 1699). 8
```

Notice that at the end of this result is the number 8. This refers to the ordinal number of the document in which the item was found as a result of the first **`part=`** parameter of the **`index`** command. This could be left out if we were to add two parameters to the end of the task:

Example 11.1

```
query name=probate;part=:item="aquacomposita"
index part=:item;
  maximum=1;
  limit="\n";
  part=:codebook[:item,glossary/text,objects];
  identification=order[];write=no
stop
```

More than one print constant can be strung together within quotation marks. The following example, tells κλειω to first produce the element :item and then move down one line and move three tab positions for the first line of the result.

Example 11.2

```
query name=probate;part=:item="aquacomposita"
index part=:item;
  maximum=1;
  limit="\n \t \t \t";
  part=:codebook[:item,glossary/text,objects];
  identification=order[];write=no
stop
```

The result of this task would look like this if sent to a file:

aquacomposita
: Aqua Composita. A drink made with a 'composite' or variety of ingredients. e.g.'Take twelve drops of Oyl of Cloves, eight Oyl of Nutmegs, & five of Oyl of Cinamon. Put them into a large strong drinking glass, & mingle well with them two ounces of the purest double refined sugar in powder' (Eleanor Sinclair Rhode i, p.219 from 'The Closet of Sir Kenneth Digby Opened', 1699).

The third print constant, `\f`, tells κλειω to advance to the next page. (These three print constants may be remembered with the mnemonic `\t` = tab, `\n` = newline and `\f` = formfeed.)

11.3 The element function `:form[]`

The `:form[]` function has also already been referred to but is more fully described here. This function allows the user to display information in a clearer fashion.

Example 11.3

```
query name=probate;part=piece:value="144" greater and
   :item=("bed" or "Bed")
write second=no;position=no;start=no;self=no;
   part=:form[""],
   :form["*************************************************"],
   :form["The individual called:"],
   root[0]/inventory/p:firstname,:surname,
   :form[""],
   :form["owned a(n)"],:query[],
   :form[""],
   :form["worth"],:value,
   :form["pence"],
   :form[""],
   :form["This item could be found in the "],back[1]:place,
   :form["*************************************************"]
stop
```

Part of the results of the task above are shown below:

```
************************************************
The individual called:
Margarett
Denham

owned a(n)
bed
```

```
worth
160.000000
pence

This item could be found in the
one of the uper garretts
**********************************************


**********************************************
The individual called:
Launcelot
Vibart

owned a(n)
Bedd Cordes

worth
150.000000
pence

This item could be found in the
ware howse
**********************************************
```

It should be fairly clear what this task is doing. Below is an annotated explanation:

`query name=probate;part=piece:value="144" greater and :item=("bed" or "Bed")`
I am interested in the database probate and I am only interested in those items in the group called piece which contain the character string "bed" or "Bed" and have a value of greater than 12 shillings (144 old pence)

`write second=no;position=no;start=no;self=no;`
Display that information to the screen following these rules. Do not display either any comments or any original material which might be found in the groups found above (**`second=no`**); do not display the ordinal number of the group and the group identifier for each of the groups output (**`position=no`**); do not display any of the information about the group (e.g. its name) (**`start=no`**); and do not display the name of the element found in the group (**`self=no`**).

`part=:form[""],`
Display the constant found within the square brackets following the element function **`:form[]`** (in this case an blank line), followed on the next line by

`:form["**************************************************"],`
the constant found within the square brackets following the element function **`:form[]`**, (in this case a line of asterisks), followed on the next line by

`:form["The individual called:"],`
the constant found within the square brackets following the element function **`:form[]`**, followed in the next line by

root[0]/inventory/p:firstname,:surname,
: the first name and the surname of the individual concerned (on separate lines), followed by

:form[""],
: the constant found within the square brackets following the element function **:form[]** (in this case a blank line), followed in the next line by

:form["owned a(n) "],:query[],
: the constant found within the square brackets following the element function **:form[]** (in this case the text **owned a(n)**), followed in the next line by the element found in the last group found within the **query** command, followed by

:form[""],
: the constant found within the square brackets following the element function **:form[]** (in this case a blank line), followed in the next line by

:form["worth"],:value,
: the constant found within the square brackets following the element function **:form[]** (in this the text **worth**), followed in the next line by the value of the item taken from the database, followed in the next line by

:form["pence"],
: the constant found within the square brackets following the element function **:form[]** (in this case the text **pence**), followed in the next line by

:form[""],
: the constant found within the square brackets following the element function **:form[]** (in this case a blank line), followed in the next line by

:form["This item could be found in the "],back[1]:place,
: the constant found within the square brackets following the element function **:form[]** (in this case some text), followed in the next line by place where the inventory showed that item to be kept

:form[""]**
: the constant found within the square brackets following the element function **:form[]**, (in this case a line of asterisks).

stop

This particular task, however, does not produce quite the nicest results possible. Using a number of expression operators and link operators, it is possible to produce something slightly more pleasing to the eye. For instance:

Example 11.4

```
query name=probate;part=piece:value="144" greater and
   :item=("bed" or "Bed")
write second=no;position=no;start=no;self=no;
   part=:form[""],
   :form["**************************************************"],
   :form["The individual called "]&+
   root[0]/inventory/p:firstname&+
   :form[" "]&+root[0]/inventory/p:surname&+
   :form[" owned a(n) "]&+:query[],
   :form[""],
   :form["worth"],:value,
   :form["pence"],
   :form[""],
   :form["This item could be found in the "]&+back[1]:place,
   :form["*************************************************"]
stop
```

which produces results like this:

```
*************************************************
The individual called Margarett Denham owned a(n) bed

worth
160.000000
pence

This item could be found in the one of the uper garretts
*************************************************

*************************************************
The individual called Launcelot Vibart owned a(n) Bedd Cordes

worth
150.000000
pence

This item could be found in the ware howse
*************************************************
```

Comparison of the task with the results should demonstrate what the new syntax in the task means. The expression operator **&** is followed by a link operator (either **+**, **-**, *****, or **:**), which allows the user to join two or more entries togther. The Reference Manual, Sections 8.1.1.5 ff., describes all the possible uses of this function, but we should point out one current failing of this function. At the moment κλειω only permits certain types of data to be joined to each other. The current possibilities are:

```
text - text
date - date
number - number
date - number
number - date
```

This means that we cannot join the contents of the element value to the constants defined in the **:form[]** functions.

The **:form[]** function can also be used with the **index** command. This is demonstrated in the following task, which may look like a rather daunting. *(Note that this example has purposely been omitted from the sample disk, so that you will be able to follow through the task while you type it out.)*

Example 11.5

```
options signs=62
query name=probate;part=piece:quantity="3" greater and
   :item="sheets"
index part=:form["**************************************"];
    limit="\n";
   part=back[2]/p:surname&+:form[", "]&+:firstname&+
      :form[" owned"];
   without=yes;
   part=query[]:quantity;without=yes;
   part=:item;without=yes;
   part=:form["(which was given in the original as"];
   without=yes;
   part=query[]:original[quantity]&+:form[")."];without=yes;
   limit="\n \n";
   part=:form["The total value of these items was"];without=yes;
   part=query[]:value;without=yes;
   part=:form["pence"];without=yes;
   part=:form["which means that each one was valued at"];
   without=yes;
   part=:value&::quantity;
   part=:form["pence."];
   limit="\n \n";
   part=:form["The call mark of the original document is: "]&+
      root[0]/inventory:invref;
    identification=order[];
    write=no
stop
```

The indentation in this example makes it look even worse than it would normally. However, this is by no means the most complicated task that κλειω could perform.

Some of the results of this task look like this:

```
**************************************
   Hayward, Thomas owned 13.000000 sheets (which was given
   in the original as

   The total value of these items was 160.000000 pence
   which means that each one was valued at 12.307693 pence.

   The call mark of the original document is: HRO1623A26/2

**************************************
   Maynard, Thomas owned 8.000000 sheets (which was given
   in the original as 4 payre).

   The total value of these items was 120.000000 pence
   which means that each one was valued at 15.000000 pence.

   The call mark of the original document is: HRO1620B26/2
```

```
************************************
    Denham, Margarett owned 56.000000 sheets (which was
    given in the original as 28 paire).

    The total value of these items was 2400.000000 pence
    which means that each one was valued at 42.857143 pence.

    The call mark of the original document is: HRO1621A20/2
```

Given the nature of your preferred historical source(s), producing output in this type of format could be execptionally useful.

The only possibly perplexing piece of syntax in the above example may be **`part=:value&::quantity`**. This parameter asks κλειω to produce the result of the operation of dividing the contents of the element value by the contents of the element quantity. Both of these elements hold data of the type **`number`** so the expression operator followed by the link operator "**`:`**" can be used to produce a numerical result.

11.4 Some export formats

The techniques outlined above can of course be used for a variety of purposes. Here are two further examples:

Generic markup

Insertion of markup for typesetting or other purposes is a simple matter. The following task:

```
query name=probate;part=will:contents
write part=:form["<reference>"]&+:willref&+
 :form["<\reference><text>"]&+:contents&+:form["<\text>"]
stop
```

produces results like the following:

```
<reference>HRO1620A76/1<\reference><text>Jesus be w[i]th me nowe and ever more Amen In the
name of God Amen the xvith day of June  1615 I George Robins Apparitor dwellinge in the p[ar]ishe
..........................
..........................
..........................
my decesse if they come or send for it.; George Robins apparitor, his seale Thomas Stampe Willia[m]
Budd [Probate 6 October 1620]<\text>
```

Exporting to databases

Exporting data in a format suitable for other database management systems is a matter of finding the appropriate format. Here is just one example:

```
query name=probate;part=total:value
index part=:form["    "];part=:form['value="'];limit="";
  part=:value;limit='"\n';
part=:form['fname="'];limit="";part=back[1]/p:firstname;
  limit='"\n';part=:form['surna="'];limit="";part=:surname;
  limit='"\n';part=:form['occup="'];limit="";part=:occupation;
  limit='"\n';identification=order[];write=no
stop
```

produces the following:

```
value="10460.000000"
fname="Samuell"
surna="Coleman"
occup=" "

value="106412.000000"
fname="Richard"
surna="Parker"
occup=" "

value="10893.000000"
fname="Mary"
surna="Fielder"
occup="Widdow"
```

If you are contemplating work involving complex output (e.g. for an edition), these tools may be cumbersome. κλειω's sister program StanFEP (Standard Format Exchange Program) was designed for such purposes. It is included with your release of κλειω, and is written in English (although unfortunately the manual has not yet been translated). For further information on StanFEP, see the introduction to this volume, p. xviii and n. 7.

Exercise

Produce in beautiful output (or as near to that as possible) the following information from the probate database, the amount in *pounds* of the goods that each person left at their death. You should include their name, their occupation (if they have one) and the amount in pound that their goods were valued at. You will need to consult the reference manual to find out how the element function **`:form[]`** allows you to do this. (By the way there were 240 old pence in a pound).

PART III

SPECIALISED FEATURES

This section introduces more complicated features of κλειω, including nominal record linkage and automated cartography. It also introduces techniques for family reconstitution. Also included in this part are more advanced concepts of database design and an alternative method of constructing databases.

Chapter 12

Nominal record linkage

12.1 Introduction

The linkage of nominal records is one of the underlying procedures of all historical research. All historians try to bring together different source material relating to the same person in order to understand more about that individual. When historians deal with a single person it is usually easy to identify that particular individual within the source. One assumes that there would be only one Wilkie Collins mentioned in the letters of Charles Dickens, but if one looked at the Census of England and Wales for 1871 more than one Wilkie Collins may be found. An historian may use other knowledge to distinguish between the many different Wilkie Collinses in that census. However, when an historian is interested in the dynamics of a group of people, rather than between individuals, it may not always be enough to rely on the historian's ability to discern between different individuals when they have similar names. This chapter demonstrates how, when using large bodies of source material, the computer can assist the researcher in deciding which may refer to the same people. The chapter also contains a reading list on some material which may be of interest.

κλειω performs the operation of nominal record linkage in a relatively simple way; however the method presented here is based on a step-by-step process which makes the procedure look rather cumbersome. This is not really the case, as will be demonstrated at the end of the chapter. We suggest that you work through this chapter completely in one session, in order to understand the process fully.

Nominal record linkage, when performed on the computer, is usually performed between two (or more) very large files of data. In this chapter we will be discussing two very small data sets which have been designed to show most of the problems encountered when performing real nominal record linkage. The chapter also contains details of other important areas of κλειω which should not be ignored even if you feel you will never use the record linkage facilities.

12.2 The data sets

The data part of the file births.mod is shown below:

```
a$p1-1/Francis/Owens/m/13 Apr 1622
a$p1-2/Francis/Owen/f/24 Aug 1643
a$p1-3/William/Owens/m/15 June 1625
a$p1-4/Ann/Nicholls/f/1 Mar 1672
a$p1-5/Anne/Nichols/f/23 Jun 1641
a$p1-6/An./Nichalls/m/5 Aug 1632
a$p1-7/Stephen/Riddingdon/m/10 Mar 1637
a$p1-8/Steven/Ridlington/m/23 Jan 1657
a$p1-9/George/ap Gryffith/m/27 Jan 1609
```

This is an imaginary source made up of first name, surname, sex and date of birth of seven individuals found in an imaginary parish register. It should help to demonstrate some of the orthographic problems encountered in historical sources.

Though none of these people are supposed to be the same as any of the others, it should be clear that only different surnames are supposed to be represented here. Obviously it would be possible to use a codebook, giving those people who looked as if they had the same surname the same code. Even though we normally feel happy about allocating codes to items of furniture or occupations, it has not typically been considered sensible to 'code' surnames by hand (though in some circumstances it is the only method), as a number of other considerations need to be taken into account when dealing with the names of individuals. The codebook deals with the human truth. Decisions can be made arbitrarily to put categories of items together, but with names it has been shown to be desirable to apply an algorithm to the original form deciding on whether people had the same name.

```
b$p2-1/Frances/Owen/f/31y/link=p1-2
b$p2-2/George/Griffiths/m/65y/link=p1-9
b$p2-3/Ann/Nichols/f/2y/link=p1-4
b$p2-4/Stephen/Riddington/m/37y/link=p1-7
b$p2-5/Frank/Owen/m/52y/link=p1-1
b$p2-6/Stephen/Ridington/m/17y/link=p1-8
```

This data is taken from a database called deaths.mod. This is also fictitious data which is supposed to represent a list of people buried during a single week in January 1674. The only oddity in this data is the element link. The contents of this element represent the unique identifier of the matching pair in the births database. Obviously this information would not be found in an original source, but is provided here to help show whether links have been correctly made or not.

Inspect the two files by looking at births.mod and deaths.mod.

12.3 An introduction to soundex

The most common algorithm used in coding names in preparation for record linkage is known as Soundex. Within κλειω the term Soundex is used broadly. The 'real' Soundex is a pre-defined coding system which changes the letters of a surname into a code based on the similarity of sounds of letters within a name (by default this code is entirely numerical but κλειω provides a parameter to use traditional Soundex codes with the initial letter of the string being soundexed remaining a letter). κλειω however uses the term soundex just to describe a Soundex-type algorithm. This has a particular advantage over some data processing systems which come with a built-in but rigid soundex system that can not be altered to suit the needs of the user. κλειω allows users to define different soundex algorithms to deal with the particular type (or language) of name that they are using.

Ensure that the files births.mod and ex12.2 have both been compiled, then run the following task:

Example 12.1

```
query name=births;part=:soundex[:surname,code]=
   :soundex[:form["Nichols"],code]
index part=:soundex[:surname,code];part=:surname
stop
```

Before considering the syntax of the task, look at the result.

```
6353   Nichalls   p1-6

6353   Nicholls   p1-4

6353   Nichols    p1-5
```

The result gives the soundex code for the surname, followed by the surname. The soundex algorithm has assigned a numerical code to each surname in the database. As such the three different surnames are considered to be the same, even though they are spelled differently.

Now to consider the task shown above:

`query name=births;`
I am interested in a database called births.
`part=:soundex[:surname,code]=:soundex[:form["Nichols"],code]`
I am interested in that part of the database where the numerical code for a surname which has had a soundex algorithm (called code) applied to it is exactly the same as the numerical code for the character string "Nichols". (The element function **`:soundex[]`** is described below.)
`index part=:soundex[:surname,code];`
Produce a sorted list of all the soundex codes that have been found after the above task.
`part=:surname`
Display the surname from which the immediately preceding part of this task has been derived, in the last column of this index.
`stop`

12.3.1 A logical object for a soundex algorithm

What exactly is a κλειω soundex algorithm and what does it do?

Soundex is a phonetic coding system usually used to bring together variant spellings of what are essentially the same name. There are many different coding systems which normally suppress any information relating to vowels in a name and replace groups of consonants by a single character representing their phoneme. The 'real' soundex is less used today as it has been found to lack sufficient discrimination on the position of the vowels within a name. However, it (or a slight variant of it) is still the phonetic coding system most commonly used by historians who use data in English.

Even though soundex is basically a phonetic coding system, its uses are not confined to phonetic processing. There may occasionally be letters in a language that have orthographic similarities other than phonetic ones. If sets of these characters are treated in the same way as groups of characters easily mistaken for each other for phonetic reasons, κλειω's soundex code could not be called 'phonetic'.

The following task contains some of the information found in the file ex12.2. The whole of that file will be considered later, but for the time being let us just consider this part of it. It contains the standard English soundex code converted into a format that κλειω can understand. The file represents a logical object called code which defines the soundex algorithm. Translated into English, the first line says create a logical object called code, which is of type soundex, and is related to the database called births. (It is permanently associated with this database and it can be overwritten.) The first line tells κλειω to ignore the letters a e i o u and y unless they are the first letter of a name, in which case the letter is to be given the code 1. The letters b p f and v should all be given the code 2, the letters c g j k q s x and z should all be given the code 3, the letters d and t should be given the code 4, the letter l should be given the code 5, the letters m and n should be given the code 6, the letter r should be represented by the code 7. The code number is determined by the command **conversion**; those characters found on the same line as this command are coded as 1; the characters occuring in the first **part** directive are given the code 2; those in the second **part** directive the code 3; those on the third **part** directive the code 4, and so on.

Example 12.2

```
item name=code;usage=soundex;type=permanent;source=births;
  overwrite=yes
conversion without="aeiouyhw"
part signs="bpfv"
part signs="cgjkqsxz"
part signs="dt"
part signs="l"
part signs="mn"
part signs="r"
exit name=code
```

A number of other rules are taken into account when applying the κλειω soundex algorithm.

- Characters not defined in the algorithm are ignored. For example the names O'Connor and Oconnor would be considered as the same unless the apostrophe has been defined as a character for coding in the algorithm. Thus all significant characters used in the database must be defined in the algorithm.

- Any group of characters following a character that is coded with the same numerical value as the code of the characters of this group is ignored. A separator, that is, one of the characters found in the **conversion** directive, does not contribute to the code being generated if it occurs within a group. As described above, it would 'separate' it so that the same code is used twice in succession.

- By default, when the first four characters of the code have been generated the remainder of the name is ignored.

- In contrast to the original Soundex code, in κλειω the initial character of a name is treated in the same way as the rest of it (i.e. it has a numeric code.) This can be changed by the user.

Thus by applying a soundex algorithm κλειω reduces the name from an alphabetical string to a numerical code based mainly on phonetic similarities.

Consider the four names below, with their codes.

```
N I C A L L S

6 x 3 x 5 x 3
```

With the soundex code as above, the name Nicalls is given the code 6353. The initial letter N is coded 6, the letter I is a separator and thus ignored, the letter C is coded 3, the letter A is also a separator and not coded. The first letter L receives a code of 5, while the second does not, because the second of two consecutive letters receiving the same code not separated by a separator is also ignored. The letter S receives the code 3. The x's represent characters which are not given a code.

```
N I C H O L L S

6 x 3 x x 5 x 3
```

Likewise the name Nicholls is given the code 6353. N is coded 6, I is ignored, C is coded 5, H would also be coded 5 but is ignored because it is preceded by a letter receiving the same code, O is ignored, L is coded 5, the second L is ignored and S is coded 3.

```
N I C H O L S

6 x 3 x x 5 3
```

The name Nichols is also given the code 6353 for almost the same reasons as the name Nicholls.

However, if we were faced with the (for this source) rather implausible name Nikekeles, we would hope that it had a different code.

```
N I K E K E L E S

6 x 3 x 3 x 5 x|3
```

The name Nikekeles receives the code 6335. The N is coded 6, the I is ignored, the first K is given the code 3, the E is ignored, the second K receives the code 3 even though the last coded letter also had a code of 3 because they are separated from each other by a separator. The next E is also ignored and L is coded 5. The next E is ignored and the final S would be given a code of 3. However, as we have seen, soundex codes are normally only four characters long and thus the last numerical code in this case would also be ignored.

Exercise 12.1

Without using κλειω, but using the soundex code described above, find out the codes for the following names:

Anderson, Andersen
Bergman, Brigham
Fischer, Fisher, Fisshire
Oldroyd, Holroyd

12.3.2 A few tips on constructing a soundex algorithm

- First, it is important to get a good sense of the data. Examine the source for easily mistaken characters, remembering that κλειω's soundex code can take into account orthographic differences as well as phonetic ones.
- Write down the alphabet. Cross off each letter as you put it into groups in the algorithm. Remember that all significant letters within the database should be included in the algorithm. (In the 'real' English soundex algorithm, the letters w and h are ignored; it may be worth considering similar possibilities for other algorithms).
- Usually odd letters like q, z and x are put into a group by themselves.
- Consider the possibility of pairing letters with no obvious phonetic similarities. Some letters have no phonetical similarities but nonetheless may have orthographic similarities (eg d and l). These letters would not normally be put in the same group, but for certain data it may be useful to do so.
- Letters which are only in the alphabet because they are found in names or words that are 'foreign' relative to the remainder of your source might form a group.

12.3.3 Using the soundex algorithm

Run the following task:

Example 12.3

```
query name=births;part=:surname
write part=:surname,:soundex[:surname,code]
stop
```

This will produce a list of all the people in the database called births with their surnames and the soundex codes for those surnames.

The `:soundex[]` element function

This has been seen above, but some further explanation is necessary here.

There are two parts to a **`:soundex[]`** element function. The first is the element specification, the second is the name of the algorithm used to process the element specified in the first parameter. The whole produces the soundex form of an element after being converted by a particular soundex definition.

```
:soundex[:surname,code]
                 /      \
            element    name of
        specification  algorithm
```

Using two element functions together

In the following example the element function **`:form[]`** is invoked to describe the element specification for the **`:soundex[]`** function. Here κλειω produces all the information about those people whose surnames, once coded with the soundex algorithm, have the same code as the name Nikols once it too has been coded with the soundex algorithm.

Example 12.4

```
query name=births;
  part=:soundex[:surname,code]=
     :soundex[:form["Nikols"],code]
write
stop
```

The **`:soundex[]`** function takes the following form:

```
:soundex[:form["Nikols"],code]
                       /           \
                   element        name of
               specification     algorithm
```

This task will be important when completing the following exercise:

Exercise 12.2

Below is an exercise which should check that you have understood what has already been said about record linkage. It is important to solve this problem to the best of your ability before continuing with this chapter.

Using the following piece of text, perform the operations which are listed below it.

> The reader is also to consider that the names of persons very often vary; and although every one thinks he spells and pronounces his own name right, yet it is not in the power of a clerk at the time of polling, to spell ever man's name exactly as he would do it himself: He must therefore be ruled more by the sound than the letters, in looking for the person's name wanted. For what writer could tell the difference in our northern manner of pronouncing Cay and Kay; Carr and Kerr; Leighton, Leaton and Layton; Turnbull and Trumball; Hindmarsh and Hynmers; Atkinson and Atchison; Kirsop and Crisop; Hawdon, Haydon, Aydon and Haddon; Irwin and Urwin; Eccles and Heccles, &c, &c yet these names are often corruptions of, or confounded and understood one for another.
>
> (From the introduction to *The Poll, &c.*, [Newcastle-upon-Tyne poll book, 1774], nd)

- Using the names in the above quotation, create a database with a structure which consists of one group and one element and a data file of 23 elements.
- Using a separate file, create a soundex algorithm which groups characters for this. (The original soundex algorithm will not solve this problem, though it was devised for English names. This should demonstrate why it might be useful to devise one's own soundex codes.)
- Create a list of ten query commands which link these names.

The solutions to these tasks will be found at the end of the book, but there are comments over the page.

Comments on exercise 12.2

(which should be read after completing the exercises)

Here is the 'standard' English Soundex code made into a logical object for use with κλειω.

```
item name=code10;usage=soundex;type=permanent;source=newcast
conversion without="aeiouywh"
part signs="bpfv"
part signs="cgjkqsxz"
part signs="dt"
part signs="l"
part signs="mn"
part signs="r"
exit name=code10
```

If this algorithm is used in conjunction with the following task on the names in the database constructed with the names from the Newcastle Poll Book, some names that are supposed to refer to the same person are given different soundex codes.

```
query name=newcast
index part=:soundex[:name,code10];part=:name
stop
```

The results of this task are shown below.

```
1353   Eccles      22
1353   Heccles     23

1433   Atchison    13
1436   Atkinson    12

1460   Aydon       18
1460   Haddon      19
1460   Hawdon      16
1460   Haydon      17

1646   Hindmarsh   10
1673   Hynmers     11

1760   Irwin       20
1760   Urwin       21


3000   Cay         1
3000   Kay         2

3700   Carr        3
3700   Kerr        4
```

3732	Crisop	15
3732	Kirsop	14
4762	Trumball	9
4762	Turnbull	8
5346	Leighton	5
5460	Layton	7
5460	Leaton	6

It will immediately be noticeable that there are three problems here. First, the names Atchison and Atkinson are respectively given the codes 1433 and 1436; second, the names Hindmarsh and Hynmers are respectively given the codes 1646 and 1673; third, the name Leighton is given a different code (5346) to the names Layton and Leaton (5460).

It would be possible to make this algorithm work (for these names) by leaving out the letters g and h, and moving d and s into the group with m and n. However, though this would work, it would not be the most effective way of getting κλειω to consider that the groups of names were in fact groups. This suggests that a more systematic way of coping with phonetic or orthographic differences between names is needed. The soundex algorithm works with single characters, but the problems encountered in the names above refer to groups of characters; for instance in the name Leighton the string "gh" is not sounded. Therefore it may be useful to remove the string "gh" from the middle of that name. This is achieved by means of a different logical object. The first part of the following file contains the standard English soundex algorithm, exactly the same logical object as described immediately above except that it asks for a logical object called **`simplify`** to be used preparatory to applying the soundex algorithm. The parameter **`preparation=`** has as a parameter value a user-defined name for a logical object which must also be described.

```
item name=code11;usage=soundex;type=permanent;source=newcast
conversion without="aeiouywh";preparation=simplify
part signs="bpfv"
part signs="cgjkqsxz"
part signs="dt"
part signs="l"
part signs="mn"
part signs="r"
exit name=code11

item name=simplify;usage=conversion;type=permanent;
   source=newcast
substitution current="ght";result="t"
substitution current="ndm";result="nm"
substitution current="nson";result="son"
exit name=simplify
```

The second part of this file contains another logical object which is used to convert groups of letters within a name. The first line should be reasonably clear. It defines a logical object called **`simplify`**, which makes systematic changes to a character string before it is processed further. The second, third and fourth lines should also be reasonably self-explanatory. They say that whenever a character string is found in a word which is the same as the character string in the **`current=`** parameter value, substitute for it the character string in the **`result=`** parameter value.

In this case the three **`substitution`** directives prepare the names in the database of Newcastle names, so that when the soundex algorithm is applied to them, each group of names is given the same soundex code and is thus considered to be the same.

It is important to recognise certain features of this logical object. For the third problem described above the string "gh" needed to be removed. However, in order to make these names correspond, the string "gh" only needs to be removed when it occurs in front of the letter t. So the **`substitution`** directive tells κλειω only to remove the string "gh" when it occurs in front of the letter t. According to the same reasoning the letter n only needs to be removed when it occurs before the letters son.

It is important to remember that this database, which is made up from a mere paragraph in the Newcastle Poll Book, is very small compared with a normal working database. In this example the **`substitution`** directive is only used three times. In a normal database on which record linkage was being performed this directive might be invoked between 80 or 90 times. However this number would of course depend on the source(s) being used. As with all applications to compensate for phonetic and orthographic problems these should be devised in conjunction with reference to the source, rather than just taking any published code system. This directive has played a prominent part in a research database linking one source in German to another in Czech.

Experience with κλειω and other record linkage projects has suggested that for Germanic languages vowels should always be separators, and there should be between six and eight **`part signs=`** groups. On the other hand, Romance languages use vowels differently, so it would be wise to remove a couple of vowels from the list of separators (depending on the language and the source), and past experience has also suggested that these languages need around ten to twelve **`part signs=`** groups. There has not been a great deal of experience with Slavic names but it is suspected that they act in a not dissimilar way to the Germanic languages. We would suggest you to refer to items in the further reading for more information about using soundex-type codes with languages other than English.

For certain projects using κλειω the soundex algorithm has proved exceptionally effective in linking records that potentially relate to the same people, but as with all record linkage projects the greater difficulty is not that of combining two groups of people but that of *not* linking unmatching records. With refinements of the algorithm, which will be demonstrated below, it is possible to recreate human linkage and perform the same task on a much larger scale. However, if one has two lists of people, both of which contain two or more people with exactly the same name and other similar attributes, it becomes much harder for the computer to decide the correct links between the files. The problems surrounding list uniqueness should be referred to in the standard record linkage texts referred to in the further reading (below, p. 264).

12.4 Linking two databases

For the rest of this chapter we will return to the examples of the births and deaths databases. In the next couple of sections we will be discussing the use of a catalogue to help link records from two databases.

In this example the problems in these two sources will be tackled one at a time. Surnames will be considered first, as they are usually the most reliable part of a name to link. (Newcome has stated that for twentieth-century Canadian names, the most common surname, Smith, has a frequency of 0.720%, while the most common first name from the same sample (John) has a frequency of 5.304%.) Thus it is usually much more reliable to find potentially matching surnames before finding potentially matching first names.

The two sample databases that we are attempting to link are both very small; as a result of this the advantages of creating a catalogue of names is not immediately apparent. However, if one were attempting to link two lists of 10,000 names each, speed would undoubtedly become an issue. As κλειω would attempt to link every name in the first list with every name in the second list there would be 100 million potential links. Most of these 100 million comparisons would be completely unnecessary for a human being to perform. One would not bother to try and link a 'Smith' with a 'Thompson'. It is clear that they are not the same person. In order for us to try to get the computer to emulate a human being (i.e. not try and link those surnames that are quite obviously different), we tell it to try to compare all those people with the same surname (e.g. Smith) in one database *only* to those who appear in a catalogue made from the second database. So the 120 Smiths in database 1 would just be compared with the 100 Smiths in database 2, instead of the 10,000 people mentioned there, which reduces the 1.2 million comparisons to 12,000. This result would be even more spectacular with less common names. The average frequency of surnames in Northern Europe tends to be about 0.1%; together with the other characteristics of the distribution of names, the above process would usually reduce the number of necessary comparisons to, at the most, 1%, of the ones that would otherwise be needed. (As there are fairly few names like Smith, the effect is, in reality, more spectacular.)

While this procedure is attractive, it would of course prevent any chance of the computer recognising that 'Smith' and 'Smythe' are identical, which it would do if these names had had a soundex algorithm applied to them. This problem is solved by applying the process described above *after* applying the soundex algorithm to those surnames. This would mean that the catalogue from database 1 would contain soundex codes for the surnames from that database. In order to qualify for a comparison, a person mentioned in database 2 would have to have the same soundex code for their surname as an entry in that catalogue.

When linking two databases, it is usual to create the catalogue out of the file with the greater number of entries. In record linkage for the purpose of family reconstitution (from baptismal and burial records), the larger file is usually that of the baptisms. For this example we have followed the assumption that this is the case, and will create a catalogue out of the surnames in the births database.

12.4.1 Creating a catalogue

First, we need to construct a catalogue of surnames from the births database. Do this as before:

Example 12.5

```
query name=births
catalogue name=bircat1;part=:surname;type=terms
stop
```

This will create a catalogue of all the surnames in the births database.

12.4.2 Querying a catalogue

Once this catalogue has been created, it would be possible run a task similar to this to display all the names like Nichols (N.B. this does not use the soundex algorithm):

Example 12.6

```
query name=births;
   part=catalogue[bircat1,complete,"Nichols"]
write part=:each[]
stop
```

As we have seen in Section 8.2, the keyword **`complete`** in the **`catalogue[]`** group function specifies that the whole of the term should be searched for.

This would produce a result like the following

Birth list (5 = "p1-5")

sex	female
dob	23.6.1641
id	p1-5
fname	Anne
surname	Nichols

Consider Figure 12.1, which shows in diagrammatic form what κλειω does when it creates a catalogue. It takes each of the names from the database, and creates an item in a catalogue which relates to it. Notice that the name Owens only occurs once in the catalogue. This is because κλειω has created a link to show that the entry in the catalogue refers to multiple entries in the database.

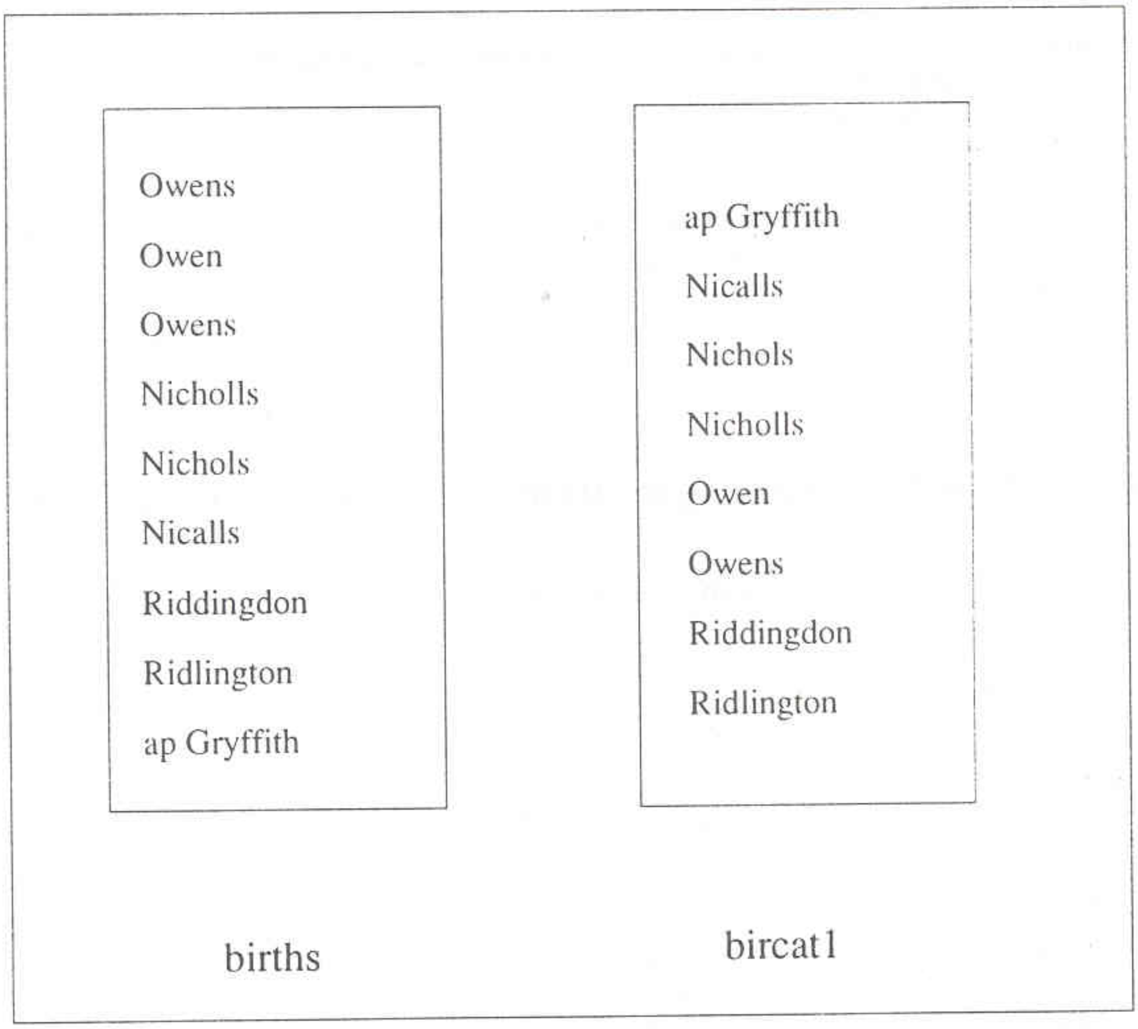


Figure 12.1

However, the soundex algorithm has not yet been applied to the surnames within the database, so that should be considered next. In this case the standard English soundex code will be applied.

12.4.3 The soundex algorithm

Below is the whole of the logical object contained in file ex12.2:

```
item name=code;usage=soundex;type=permanent;source=births
conversion without="aeiouywh";preparation=simplify
part signs="bpfv"
part signs="cgjkqsxz"
part signs="dt"
part signs="l"
part signs="mn"
part signs="r"
exit name=code

item name=simplify;usage=conversion;type=permanent;
  source=births
substitution current="dli";result="di"
exit name=simplify
```

This is the soundex algorithm that will be used with the births and deaths databases. Not only does it include the 'standard' English soundex, it includes a single **substitution** directive to prepare the names Ridlington and Ridlingdon. (Further details concerning this directive can be found in Section 12.4.7.)

12.4.4 Creating a catalogue using a soundex algorithm

Once this algorithm has been run it is necessary to create a new catalogue of all the soundexed surnames.

Example 12.7

```
query name=births
catalogue name=bircat2;part=:surname;type=terms;soundex=code
stop
```

This task is slightly different from the task used above to get κλειω to produce a catalogue. We have introduced a new parameter, **soundex=**. This takes as its parameter value the name of the logical object which holds the applicable rules for the soundex algorithm. As the parameter value in this case is **code**, κλειω insists that that word occurs in a **name=** parameter of an **item** command. (In this case it can be found in the file ex12.2.)

This new catalogue consists of all the surnames once they have had the soundex algorithm called code applied to it.

12.4.5 Querying a catalogue using the soundex algorithm

The previous task that was used to query the catalogue was ex12.6. This needs to be changed slightly in order to query that part of the catalogue which contains the soundexed surnames. Since we have made a different catalogue that will need to be changed, and the keyword **`complete`** will have to be changed to **`algorithm`**. This tells κλειω that the term being searched for should first be converted into a "derived format". In this case, because the catalogue has a soundex entry, soundex is the derived format. (κλειω has two different coding systems.)

Example 12.8

```
query name=births;
   part=catalogue[bircat2,algorithm,"Nichols"]
write part=:soundex[:surname,code],:surname
continue
query name=births;
   part=catalogue[bircat2,algorithm,"Owens"]
write part=:soundex[:surname,code],:surname
stop
```

This gives the result:

```
6353
Birth list (4 = "p1-4") : surname      Nicholls
6353
Birth list (5 = "p1-5") : surname      Nichols
6353
Birth list (6 = "p1-6") : surname      Nicalls

1630
Birth list (1 = "p1-1") : surname      Owens
1630
Birth list (3 = "p1-3") : surname      Owens
```

In order to describe fully what is happening here, it is worth considering more precisely how κλειω catalogues function. When κλειω creates a catalogue of an element within a database, each relevant entry in the database is given an entry in a catalogue, but each entry in a catalogue can be made up of more than one piece of information. In this case there are two; the surnames and their soundexed equivalent. The importance of this will not be lost when it is considered that the keyword **`algorithm`** relates to that part of a catalogue which deals with entries that have had an algorithm applied to them.

Figure 12.2

12.4.6 Joining two databases

The following task shows how two databases are joined together. Notice that, although the second catalogue that we have created is being used, the soundex part of it is not!

Example 12.9

```
query name=deaths;also=births;
   part=catalogue[<births>bircat2,complete,<deaths>:surname]
write part=<deaths>:each[],<births>:each[]
stop
```

An explanation follows:

`query name=deaths;also=births;`
: I am interested in a database called deaths, and I am also interested in a database called births.

`part=catalogue[<births>bircat2,complete,<deaths>:surname]`
: I am interested in that part of the deaths database where there is a surname. Once a surname has been found in this database, look it up in the catalogue called bircat2 (which relates to the database births). If any pairs of surnames are found to be identical,

`write part=<deaths>:each[];<births>:each[]`
: display all the information about both groups on the screen.

`stop`

Notice that when two databases are being interrogated with the same task, the **`query`** command must have both a **`name=`** parameter and an **`also=`** parameter. Each takes as a value the name of one of the two databases being interrogated. The rules relating to this are described in the Reference Manual, Section 9.2.1.

Note also another convention:

> When more than one database is invoked in a κλειω task the names of those databases must be enclosed in angled brackets '<', '>' when used in a task.

Compare the second line of this task with the second line of the task in Section 12.4.2.

The result of this task looks like this.

```
Death list (1 = "p2-1")
        sex             female
        age             31y
        id              p2-1
        fname           Frances
        surname         Owen
        link            p1-2

Birth list (2 = "p1-2")
        sex             female
        dob             24.8.1643
        id              p1-2
        fname           Francis
        surname         Owen
```

```
Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (5 = "p1-5")
        sex             female
        dob             23.6.1641
        id              p1-5
        fname           Anne
        surname         Nichols

Death list (5 = "p2-5")
        sex             male
        age             52y
        id              p2-5
        fname           Frank
        surname         Owen
        link            p1-1

Birth list (2 = "p1-2")
        sex             female
        dob             24.8.1643
        id              p1-2
        fname           Francis
        surname         Owen
```

In this result κλειω has suggested three potential links between the two databases, displaying information about three people who have exactly the same surnames. In order to find more potential links it would be wise to perform the same sort of task on the soundexed names. However, if one checks the contents of the element link from the Death list and the id from the Birth list that follow them, it can be seen that only one of these three potential links is correct (the first pair). Eventually we will modify the task as put to κλειω in order to remove these incorrect links.

To see if κλειω can produce more potential links, it would be wise to perform the same sort of task on the soundexed names.

We mentioned above that the larger database of a pair was used to make the catalogue. For the same reason, the smaller database is usually made to be the ‘first’ database in a task that contains more than one database name.

12.4.7 Joining two databases using a soundex algorithm

The changes to be made to the previous task are slight. The name of the catalogue has to be changed, and the access path has to be changed from **`complete`** to **`algorithm`**. This signifies that κλειω should search for a term after it has been converted into a derived format.

Example 12.10

```
query name=deaths;also=births;
   part=catalogue[<births>bircat2,algorithm,<deaths>:surname]
write part=<deaths>:each[],<births>:each[]
stop
```

This query produces a much longer list of potential linked people:

```
Death list (1 = "p2-1")
        sex             female
        age             31y
        id              p2-1
        fname           Frances
        surname         Owen
        link            p1-2

Birth list (2 = "p1-2")
        sex             female
        dob             24.8.1643
        id              p1-2
        fname           Francis
        surname         Owen

Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (4 = "p1-4")
        sex             female
        dob             1.3.1672
        id              p1-4
        fname           Ann
        surname         Nicholls
```

```
Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (5 = "p1-5")
        sex             female
        dob             23.6.1641
        id              p1-5
        fname           Anne
        surname         Nichols

Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (6 = "p1-6")
        sex             male
        dob             5.8.1632
        id              p1-6
        fname           An.
        surname         Nicalls

Death list (4 = "p2-4")
        sex             male
        age             37y
        id              p2-4
        fname           Stephen
        surname         Riddington
        link            p1-7

Birth list (7 = "p1-7")
        sex             male
        dob             10.3.1637
        id              p1-7
        fname           Stephen
        surname         Ridlingdon

Death list (4 = "p2-4")
        sex             male
        age             37y
        id              p2-4
        fname           Stephen
        surname         Riddington
        link            p1-7
```

```
Birth list (8 = "p1-8")
          sex             male
          dob             23.1.1657
          id              p1-8
          fname           Steven
          surname         Ridlington


Death list (5 = "p2-5")
          sex             male
          age             52y
          id              p2-5
          fname           Frank
          surname         Owen
          link            p1-1

Birth list (2 = "p1-2")
          sex             female
          dob             24.8.1643
          id              p1-2
          fname           Francis
          surname         Owen

Death list (6 = "p2-6")
          sex             male
          age             17y
          id              p2-6
          fname           Stephen
          surname         Ridington
          link            p1-8

Birth list (7 = "p1-7")
          sex             male
          dob             10.3.1637
          id              p1-7
          fname           Stephen
          surname         Ridlingdon

Death list (6 = "p2-6")
          sex             male
          age             17y
          id              p2-6
          fname           Stephen
          surname         Ridington
          link            p1-8

Birth list (8 = "p1-8")
          sex             male
          dob             23.1.1657
          id              p1-8
          fname           Steven
          surname         Ridlington
```

Even though this list is much longer, it does not bring up all the possible links. Refer back to the births and deaths databases. Two important problems remain; linking the surnames Owen and Griffiths to Owens and ap Gryffith. To cope with this, two further **subst-itution** directives have been added to the **conversion** algorithm.

Example 12.11

```
item name=code2;usage=soundex;type=permanent;source=births
conversion without="aeiouywh";preparation=simplify2
part signs="bpfv"
part signs="cgjkqsxz"
part signs="dt"
part signs="l"
part signs="mn"
part signs="r"
exit name=code2

item name=simplify2;usage=conversion;type=permanent;
   source=births
substitution current="dli";result="di"
type limit=yes
substitution result="";current="s"
type start=yes
substitution result="";current="ap"
exit name=simplify2
```

In the logical object simplify2, the first **substitution** directives tell κλειω to substitute the character string "di" wherever it finds the string "dli". These two strings are parameter values for the **result=** and **current=** parameters respectively. The second **subst-itution** directive tells κλειω to replace the character string "s" with the string "" (i.e. nothing). However, as it would be unwise to remove all occurrences of the letter s in a word when it was only necessary to replace terminal s's, the **type** directive followed by the parameter **limit=yes** ensures that these substitutions only take place when the letter "s" is at the end of a word. Likewise, for the third **substitution** directive, the preceding **type start=yes** command ensures that the string "ap" is only removed when it occurs at the beginning of an entry.

As a new soundex algorithm has been defined, a new catalogue needs to be defined. (It would be possible to overwrite the existing algorithm and the catalogue, but for clarity here we will create new ones.)

Example 12.12

```
query name=births
catalogue name=bircat3;part=:surname;type=terms;soundex=code2
stop
```

A new task has also to be written to perform the linkages:

Example 12.13

```
query name=deaths;also=births;
   part=catalogue[<births>bircat3,algorithm,<deaths>:surname]
write part=<deaths>:each[],<births>:each[]
stop
```

The result is longer again:

```
Death list (1 = "p2-1")
        sex             female
        age             31y
        id              p2-1
        fname           Frances
        surname         Owen
        link            p1-2

Birth list (1 = "p1-1")
        sex             male
        dob             13.4.1622
        id              p1-1
        fname           Francis
        surname         Owens

Death list (1 = "p2-1")
        sex             female
        age             31y
        id              p2-1
        fname           Frances
        surname         Owen
        link            p1-2

Birth list (2 = "p1-2")
        sex             female
        dob             24.8.1643
        id              p1-2
        fname           Francis
        surname         Owen

Death list (1 = "p2-1")
        sex             female
        age             31y
        id              p2-1
        fname           Frances
        surname         Owen
        link            p1-2
```

```
Birth list (3 = "p1-3")
        sex             male
        dob             15.6.1625
        id              p1-3
        fname           William
        surname         Owens

Death list (2 = "p2-2")
        sex             male
        age             65y
        id              p2-2
        fname           George
        surname         Griffiths
        link            p1-9

Birth list (9 = "p1-9")
        sex             male
        dob             27.1.1609
        id              p1-9
        fname           George
        surname         ap Gryffith

Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (4 = "p1-4")
        sex             female
        dob             1.3.1672
        id              p1-4
        fname           Ann
        surname         Nicholls

Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (5 = "p1-5")
        sex             female
        dob             23.6.1641
        id              p1-5
        fname           Anne
        surname         Nichols
```

```
Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (6 = "p1-6")
        sex             male
        dob             5.8.1632
        id              p1-6
        fname           An.
        surname         Nicalls

Death list (4 = "p2-4")
        sex             male
        age             37y
        id              p2-4
        fname           Stephen
        surname         Riddington
        link            p1-7

Birth list (7 = "p1-7")
        sex             male
        dob             10.3.1637
        id              p1-7
        fname           Stephen
        surname         Ridlingdon

Death list (4 = "p2-4")
        sex             male
        age             37y
        id              p2-4
        fname           Stephen
        surname         Riddington
        link            p1-7

Birth list (8 = "p1-8")
        sex             male
        dob             23.1.1657
        id              p1-8
        fname           Steven
        surname         Ridlington

Death list (5 = "p2-5")
        sex             male
        age             52y
        id              p2-5
        fname           Frank
        surname         Owen
        link            p1-1
```

```
Birth list (1 = "p1-1")
          sex           male
          dob           13.4.1622
          id            p1-1
          fname         Francis
          surname       Owens

Death list (5 = "p2-5")
          sex           male
          age           52y
          id            p2-5
          fname         Frank
          surname       Owen
          link          p1-1

Birth list (2 = "p1-2")
          sex           female
          dob           24.8.1643
          id            p1-2
          fname         Francis
          surname       Owen

Death list (5 = "p2-5")
          sex           male
          age           52y
          id            p2-5
          fname         Frank
          surname       Owen
          link          p1-1

Birth list (3 = "p1-3")
          sex           male
          dob           15.6.1625
          id            p1-3
          fname         William
          surname       Owens

Death list (6 = "p2-6")
          sex           male
          age           17y
          id            p2-6
          fname         Stephen
          surname       Ridington
          link          p1-8

Birth list (7 = "p1-7")
          sex           male
          dob           10.3.1637
          id            p1-7
          fname         Stephen
          surname       Ridlingdon
```

```
Death list (6 = "p2-6")
        sex           male
        age           17y
        id            p2-6
        fname         Stephen
        surname       Ridington
        link          p1-8

Birth list (8 = "p1-8")
        sex           male
        dob           23.1.1657
        id            p1-8
        fname         Steven
        surname       Ridlington
```

This list now gives all the potential links between people in both databases where the surname from the deaths database (once soundexed) are the same as the soundexed surnames in the catalogue (bircat3 — which relate to the database births). But there are still problems, for example people who are obviously different, such as Frances Owen and William Owens who are still linked. There is also the problem that many different people are suggested as linked. Therefore we should consider using a condition to sort out the problem caused by the first names.

An example of a task that would be able to consider this is ex12.14:

Example 12.14

```
query name=deaths;also=births;
  part=catalogue[<births>bircat3,algorithm,<deaths>:surname]
  :fname=<deaths>:fname
write part=<deaths>:each[],<births>:each[]
stop
```

This task is only slightly different from the previous one. It adds a condition; to select only those pairs of surnames from the previous selection where the first names are also the same. Previously we have seen that one can write **part name=**group:element="constant". The condition in the above task works in exactly the same way, where the **catalogue[]** group function refers to a group and **:fname=<deaths>:fname** refers to an element followed by a condition.

```
Death list (2 = "p2-2")
        sex           male
        age           65y
        id            p2-2
        fname         George
        surname       Griffiths
        link          p1-9
```

```
Birth list (9 = "p1-9")
        sex             male
        dob             27.1.1609
        id              p1-9
        fname           George
        surname         ap Gryffith

Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (4 = "p1-4")
        sex             female
        dob             1.3.1672
        id              p1-4
        fname           Ann
        surname         Nicholls

Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (5 = "p1-5")
        sex             female
        dob             23.6.1641
        id              p1-5
        fname           Anne
        surname         Nichols

Death list (4 = "p2-4")
        sex             male
        age             37y
        id              p2-4
        fname           Stephen
        surname         Riddington
        link            p1-7

Birth list (7 = "p1-7")
        sex             male
        dob             10.3.1637
        id              p1-7
        fname           Stephen
        surname         Ridlingdon
```

Death list (6 = "p2-6")

sex	male
age	17y
id	p2-6
fname	Stephen
surname	Ridington
link	p1-8

Birth list (7 = "p1-7")

sex	male
dob	10.3.1637
id	p1-7
fname	Stephen
surname	Ridlingdon

Note that the parameter value of a **`part=`** parameter always identifies a group, whether it is just an element within a group or a group itself. Thus a usable parameter value must relate to a group, so that:

```
catalogue[<births>bircat3,algorithm,<deaths>:surname]
```

refers to a group, while adding **`:fname`** after it (as in ex12.14) indicates that it is an element which refers to that group. In simpler κλειω queries we have seen conditions after groups; this would be perfectly possible here.

However this task produces a much shorter list, as it misses out the alternative spellings of the first names. The simplest way of ensuring that this does not happen is to soundex those names as well.

Though it is not always necessary to produce separate soundex algorithms for first names and surnames, it is considered good practice to have a different code for each. First names and surnames have different properties. There are standard abbreviations for Christian names; for example, Thos is (almost!) always an abbreviation for Thomas, but if the string Thos occurred as the beginning of a surname it would not be appropriate to convert it into the string Thomas before processing it. Certain registers in the eighteenth and nineteenth centuries used pseudo-Latin for first names, which would also require a different coding system from that for surnames.

The file ex12.15 contains the standard English soundex codes in a logical object called xtcode. This soundex algorithm will be used with first names.

Example 12.15

```
item name=xtcode;usage=soundex;type=permanent;source=births
conversion without="aeiouywh"
part signs="bpfv"
part signs="cgjkqsxz"
part signs="dt"
part signs="l"
part signs="mn"
part signs="r"
exit name=xtcode
```

Once this has be made, the following task will take care of the problem. Though this looks much more complicated, compare it with the task in the file ex12.14.

Example 12.16

```
query name=deaths;also=births;
part=catalogue[<births>bircat3,algorithm,<deaths>:surname]
   :soundex[:fname,<births>xtcode]=<deaths>
   :soundex[:fname,<births>xtcode]
write part=<deaths>:each[],<births>:each[]
stop
```

This translates as:

query name=deaths;also=births;
I am interested in a database called births; I am also interested in a database called deaths.

part=catalogue[<births>bircat3,algorithm,<deaths>:surname]:
soundex[:fname,<births>xtcode]=<deaths>
:soundex[:fname,<births>xtcode]
Firstly assign codes to the first names in the database called births with the soundex code called xtcode, then go to the database called deaths, treat the first names in a similar way and then compare them. If they are the same, continue by checking whether the surnames in the deaths database are the same as those in the catalogue bircat3 which relates to the database births.

write part=<deaths>:each[],<births>:each[]
Display on the screen all the information about the two groups which fulfil the criteria mentioned above.

stop

The result of this task will now bring up pairs of people whose surnames have the same soundex codes after they have been treated by the logical object pre-treating those surnames, and who have the same first names once they too have been treated by a different soundex algorithm.

```
Death list (1 = "p2-1")
          sex             female
          age             31y
          id              p2-1
          fname           Frances
          surname         Owen
          link            p1-2

Birth list (1 = "p1-1")
          sex             male
          dob             13.4.1622
          id              p1-1
          fname           Francis
          surname         Owens

Death list (1 = "p2-1")
          sex             female
          age             31y
          id              p2-1
          fname           Frances
          surname         Owen
          link            p1-2

Birth list (2 = "p1-2")
          sex             female
          dob             24.8.1643
          id              p1-2
          fname           Francis
          surname         Owen

Death list (2 = "p2-2")
          sex             male
          age             65y
          id              p2-2
          fname           George
          surname         Griffiths
          link            p1-9

Birth list (9 = "p1-9")
          sex             male
          dob             27.1.1609
          id              p1-9
          fname           George
          surname         ap Gryffith

Death list (3 = "p2-3")
          sex             female
          age             2y
          id              p2-3
          fname           Ann
          surname         Nichols
          link            p1-4
```

```
Birth list (4 = "p1-4")
        sex             female
         dob            1.3.1672
        id              p1-4
        fname           Ann
        surname         Nicholls

Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (5 = "p1-5")
        sex             female
        dob             23.6.1641
        id              p1-5
        fname           Anne
        surname         Nichols

Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (6 = "p1-6")
        sex             male
        dob             5.8.1632
        id              p1-6
        fname           An.
        surname         Nicalls

Death list (4 = "p2-4")
        sex             male
        age             37y
        id              p2-4
        fname           Stephen
        surname         Riddington
        link            p1-7

Birth list (7 = "p1-7")
        sex             male
        dob             10.3.1637
        id              p1-7
        fname           Stephen
        surname         Ridlingdon
```

```
Death list (4 = "p2-4")
        sex             male
        age             37y
        id              p2-4
        fname           Stephen
        surname         Riddington
        link            p1-7

Birth list (8 = "p1-8")
        sex             male
        dob             23.1.1657
        id              p1-8
        fname           Steven
        surname         Ridlington

Death list (5 = "p2-5")
        sex             male
        age             52y
        id              p2-5
        fname           Frank
        surname         Owen
        link            p1-1

Birth list (1 = "p1-1")
        sex             male
        dob             13.4.1622
        id              p1-1
        fname           Francis
        surname         Owens

Death list (5 = "p2-5")
        sex             male
        age             52y
        id              p2-5
        fname           Frank
        surname         Owen
        link            p1-1

Birth list (2 = "p1-2")
        sex             female
        dob             24.8.1643
        id              p1-2
        fname           Francis
        surname         Owen

Death list (6 = "p2-6")
        sex             male
        age             17y
        id              p2-6
        fname           Stephen
        surname         Ridington
        link            p1-8
```

```
Birth list (7 = "p1-7")
        sex             male
        dob             10.3.1637
        id              p1-7
        fname           Stephen
        surname         Ridlingdon

Death list (6 = "p2-6")
        sex             male
        age             17y
        id              p2-6
        fname           Stephen
        surname         Ridington
        link            p1-8

Birth list (8 = "p1-8")
        sex             male
        dob             23.1.1657
        id              p1-8
        fname           Steven
        surname         Ridlington
```

This has effectively taken care of all the similar first names, but a cursory glance at the possible solutions that κλειω has suggested shows that there are still some incorrect links. Most notably, Frances Owen from the death list is a woman, while the potential link Francis Owen is a man. Though gender-ambiguous first names are not frequent in English, this problem would be much greater in Romance languages where gender is denoted by a different vowel ending.

A possible solution to this problem would be to add a further condition to the **`part=`** parameter. The whole of the **`part=`** parameter might now read:

```
part=catalogue[<births>bircat3,algorithm,<deaths>:surname]:
   soundex[:fname,<births>xtcode]=
   <deaths>:soundex[:fname,<births>xtcode]
      and <births>:sex=<deaths>:sex
```

This extra condition (and **`<births>:sex=<deaths>:sex`**) tells the system to continue to the analysis command only where the sex of the individual in the births database is the same as the sex of the person in the deaths database.

However, this does not make the whole condition any clearer, and if there were other conditions to be added to this it would only add to the confusion.

12.4.8 Block-structured tasks

A much better way of adding a condition is to use it within a block. A block-structured task performs a similar function to a condition. When a block is used κλειω decides whether part of a database fulfils a condition set by a **`query`** command. If it does then κλειω will continue to process the task. In the case below, κλειω first selects all the potential links within the database and then 'checks' those results against the query found in the block. The task in this block could be translated as saying "if the result of the above task is true AND the sex of the two people chosen is the same then display the results to the screen".

Blocks must consist of a block initialisation command (in this case **`confirm`**), any number of κλειω instructions, and an **`exit`** command. Each block initialisation command must be followed by a **`name=`** parameter, with a user-defined name as the parameter value. Likewise the **`exit`** command. The block initialisation commands **`confirm`** and **`negate`** are only valid after a **`query`** command which is followed by a condition.

A block will only be entered if a condition in a previous **`query`** command is fulfilled.

Example 12.17

```
query name=deaths;also=births;
  part=catalogue[<births>bircat3,algorithm,<deaths>:surname]
  :soundex[:fname,<births>xtcode]=
  <deaths>:soundex[:fname,<births>xtcode]
confirm name=gender
query part=<births>:sex=<deaths>:sex
write part=<deaths>:each[],<births>:each[]
exit name=gender
stop
```

In this case the task checks the first names and the surnames as previously, and then, if the two people chosen are of the same gender, the results are written to the screen.

```
Death list (1 = "p2-1")
        sex         female
        age         31y
        id          p2-1
        fname       Frances
        surname     Owen
        link        p1-2

Birth list (2 = "p1-2")
        sex         female
        dob         24.8.1643
        id          p1-2
        fname       Francis
        surname     Owen
```

```
Death list (2 = "p2-2")
        sex             male
        age             65y
        id              p2-2
        fname           George
        surname         Griffiths
        link            p1-9

Birth list (9 = "p1-9")
        sex             male
        dob             27.1.1609
        id              p1-9
        fname           George
        surname         ap Gryffith

Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (4 = "p1-4")
        sex             female
        dob             1.3.1672
        id              p1-4
        fname           Ann
        surname         Nicholls

Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (5 = "p1-5")
        sex             female
        dob             23.6.1641
        id              p1-5
        fname           Anne
        surname         Nichols

Death list (4 = "p2-4")
        sex             male
        age             37y
        id              p2-4
        fname           Stephen
        surname         Riddington
        link            p1-7
```

```
Birth list (7 = "p1-7")
        sex             male
        dob             10.3.1637
        id              p1-7
        fname           Stephen
        surname         Ridlingdon

Death list (4 = "p2-4")
        sex             male
        age             37y
        id              p2-4
        fname           Stephen
        surname         Riddington
        link            p1-7

Birth list (8 = "p1-8")
        sex             male
        dob             23.1.1657
        id              p1-8
        fname           Steven
        surname         Ridlington

Death list (5 = "p2-5")
        sex             male
        age             52y
        id              p2-5
        fname           Frank
        surname         Owen
        link            p1-1

Birth list (1 = "p1-1")
        sex             male
        dob             13.4.1622
        id              p1-1
        fname           Francis
        surname         Owens

Death list (6 = "p2-6")
        sex             male
        age             17y
        id              p2-6
        fname           Stephen
        surname         Ridington
        link            p1-8

Birth list (7 = "p1-7")
        sex             male
        dob             10.3.1637
        id              p1-7
        fname           Stephen
        surname         Ridlingdon
```

```
Death list (6 = "p2-6")
        sex          male
        age          17y
        id           p2-6
        fname        Stephen
        surname      Ridington
        link         p1-8

Birth list (8 = "p1-8")
        sex          male
        dob          23.1.1657
        id           p1-8
        fname        Steven
        surname      Ridlington
```

There is an interesting side effect relating to the **confirm** command. Usually one would want the condition to be satisfied before continuing with the task, but there may be cases where we would like output for data not satisfying the condition as well as output of that data that does satisfy it.

```
query name=deaths;also=births;
  part=catalogue[<births>bircat3,algorithm,<deaths>:surname]
  :soundex[:fname,<births>xtcode]=
  <deaths>:soundex[:fname,<births>xtcode]
confirm name=gender
query part=<births>:sex=<deaths>:sex
write part=<deaths>:each[],<births>:each[]
exit name=gender
negate name=no
index part=<births>:fname;
  part=<deaths>:fname
exit name=no
stop
```

In this example the contents of the **negate** block would cause a list to be written of those first names that were the same but were of a different gender. This list would be displayed in the output after the other data.

Returning to the previous list, it is still possible to discover some errors in κλειω's links. If we remember that the list of burials were all supposed to take place during the same week, and that we have the age of people at burial, we can ascertain the approximate date of birth of those people who were buried. For some of the potential links that κλειω has suggested, the date of burial less the age at burial is nowhere near the date of birth. (In a real source this could not be used as a discriminating element as people were not always baptised within the first few weeks of life.)

This problem can be solved by using another block. Just as any retrieval command can be used within a block, a block can be contained within another block. In this case a further block containing a condition specifying that the date of birth when added to the age at death must be between 1 January 1674 and 1 January 1676. κλειω, will, for example let you 'add' dates and numbers. By default, κλειω adds a number (which should represent a number of year — for example an age), to the year 'element' of a calendar date. Thus a date of birth added to an age would give as a result a year. For example, if κλειω were asked to add the age 25 to the year 1.1.1900 the result would be 1.1.1925.

Example 12.18

```
query name=deaths;also=births;
  part=catalogue[<births>bircat3,algorithm,<deaths>:surname]
  :soundex[:fname,<births>xtcode]=
  <deaths>:soundex[:fname,<births>xtcode]
confirm name=gender
query part=<births>:sex=<deaths>:sex
confirm name=agecount
query part=<births>:dob&+<deaths>:age=
  after "1.1.1674" and before "1.1.1676"
write part=<deaths>:each[],<births>:each[]
exit name=agecount
exit name=gender
stop
```

The command **`confirm name=agecount`** means that once the conditions preceding this have been fulfilled, check whether the following **`query`** command is correct, before proceeding to perform any of the further commands. The specification

```
part=<births>:dob&+<deaths>:age= after "1.1.1674" and
  before "1.1.1676
```

means that if the date of birth (found in the births database) added to the age given in the deaths database (for any potential pair of linked people) is between 2 January 1674 and 31 December 1675, continue processing this task.

This should look familiar if you remember the last chapter where *link operators* were described as a facility to assist in more beautiful output. The ampersand is known in κλειω as the *expression operator* and must precede any arithmetical operator that is being used. When used with dates or text these arithmetical operators are known as link operators but perform the same function. κλειω allows one to perform link operations on dates and numbers and on text. (i.e. it would be possible to say **`:surname&+fname`**).

In κλειω the numerical operators are "+" (for addition), "-" (for subtraction), "*" (for multiplication) and ":" (for division).

This final task displays only the correct links.

```
Death list (1 = "p2-1")
        sex             female
        age             31y
        id              p2-1
        fname           Frances
        surname         Owen
        link            p1-2

Birth list (2 = "p1-2")
        sex             female
        dob             24.8.1643
        id              p1-2
        fname           Francis
        surname         Owen

Death list (2 = "p2-2")
        sex             male
        age             65y
        id              p2-2
        fname           George
        surname         Griffiths
        link            p1-9

Birth list (9 = "p1-9")
        sex             male
        dob             27.1.1609
        id              p1-9
        fname           George
        surname         ap Gryffith

Death list (3 = "p2-3")
        sex             female
        age             2y
        id              p2-3
        fname           Ann
        surname         Nichols
        link            p1-4

Birth list (4 = "p1-4")
        sex             female
        dob             1.3.1672
        id              p1-4
        fname           Ann
        surname         Nicholls

Death list (4 = "p2-4")
        sex             male
        age             37y
        id              p2-4
        fname           Stephen
        surname         Riddington
        link            p1-7
```

```
Birth list (7 = "p1-7")
        sex             male
        dob             10.3.1637
        id              p1-7
        fname           Stephen
        surname         Ridlingdon

Death list (5 = "p2-5")
        sex             male
        age             52y
        id              p2-5
        fname           Frank
        surname         Owen
        link            p1-1

Birth list (1 = "p1-1")
        sex             male
        dob             13.4.1622
        id              p1-1
        fname           Francis
        surname         Owens

Death list (6 = "p2-6")
        sex             male
        age             17y
        id              p2-6
        fname           Stephen
        surname         Ridington
        link            p1-8

Birth list (8 = "p1-8")
        sex             male
        dob             23.1.1657
        id              p1-8
        fname           Steven
        surname         Ridlington
```

This is the final task needed to link these two databases correctly. All of the correct links have been found.

12.4.9 Joining two databases permanently

Now that we have all this information about the links between two databases, it is possible to ask κλειω to use this information in a constructive way. And we can ask κλειω to do it without having to perform this task over and over again. (Remember that running this task with two lists of 10,000 names may take quite some time.)

To get κλειω to use this information in a useful way, it would be possible to run the following task in order to assist the processing of this information. It fulfils the same basic purpose as the previous task, but it formats the results in a considerably different way.

Example 12.19

```
options lines=0
query name=deaths;also=births;
   part=catalogue[<births>bircat3,algorithm,<deaths>:surname]
      :soundex[:fname,<births>xtcode]=
     <deaths>:soundex[:fname,<births>xtcode]
confirm name=gender
query part=<births>:sex=<deaths>:sex
confirm name=agecount
query part=<births>:dob&+<deaths>:age=
     after "1.1.1674" and before "1.1.1676"
index part=:form['bridge first="'];limit="";
   part=<deaths>/root[0]:sign[];limit='";second="';
   part=<births>/root[0]:sign[];limit='"';
   identification=root[0]:sign[];write=no
exit name=agecount
exit name=gender
stop target="ex12.20"
```

The result looks like this:

```
bridge first="p2-1";second="p1-2"
bridge first="p2-2";second="p1-9"
bridge first="p2-3";second="p1-4"
bridge first="p2-4";second="p1-7"
bridge first="p2-5";second="p1-1"
bridge first="p2-6";second="p1-8"
```

For the moment only consider the items contained within quotation marks. In the first line the two strings are p2-1 and p1-2. These refer to the two id numbers from the deaths and the births databases respectively of a linked pair. Each subsequent line contains another pair of id numbers for linked pairs. Since we know that these refer to the same people, a new declaration can be used to 'join' these two databases together. This declaration is known as the **bridge** declaration, as it bridges the gap between two different databases.

Every **bridge** declaration begins with a **bridge** command, which must specify in two parameters the names of the two databases being linked. These two parameters are **first=** and **second=**. In this case a **type=** parameter is needed with the parameter value **definitions** to denote that the linkages between the two databases are made by way of an explicit definition.

Example 12.21

```
bridge first=deaths;second=births;type=definitions
bridge first="p2-1";second="p1-2"
bridge first="p2-2";second="p1-9"
bridge first="p2-3";second="p1-4"
bridge first="p2-4";second="p1-7"
bridge first="p2-5";second="p1-1"
bridge first="p2-6";second="p1-8"
exit name=definitions
```

Figure 12.3 shows what κλειω does when it performs the **bridge** command, demonstrating graphically the links between the elements in two databases.

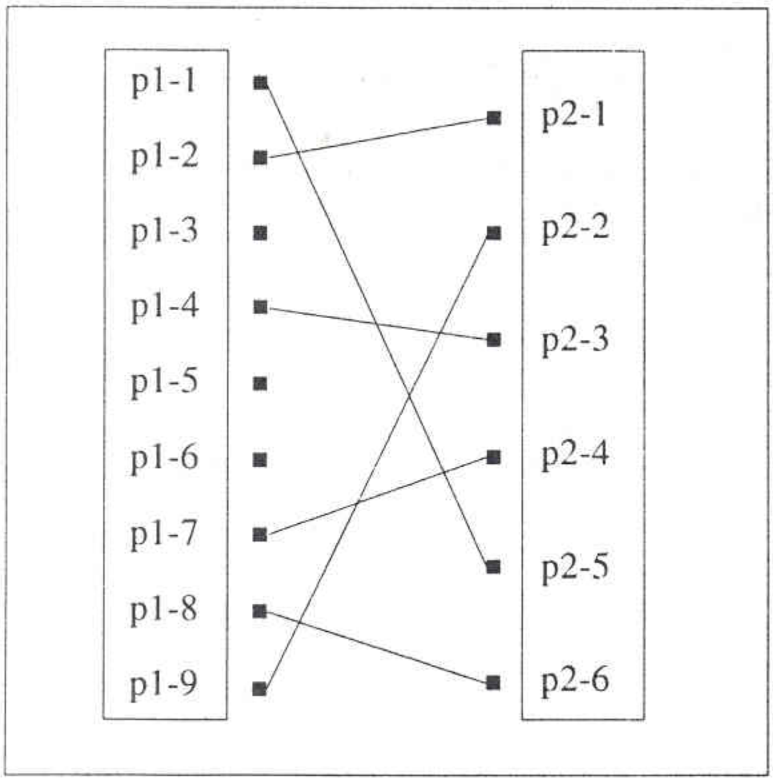


Figure 12.3

Running this task permanently integrates the information from it into the database where it will be available for all future tasks. Run the file ex12.22 to see what occurs.

Example 12.22

```
query name=births
write
stop
```

This produces the following:

```
Birth list (1 = "p1-1")
        sex             male
        dob             13.4.1622
        id              p1-1
        fname           Francis
        surname         Owens
        deaths          <deaths>Death list (5 = "p2-5") : Death list (5 = "p2-5") -> births

Birth list (2 = "p1-2")
        sex             female
        dob             24.8.1643
        id              p1-2
        fname           Francis
        surname         Owen
        deaths          <deaths>Death list (1 = "p2-1") : Death list (1 = "p2-1") -> births

Birth list (3 = "p1-3")
        sex             male
        dob             15.6.1625
        id              p1-3
        fname           William
        surname         Owens

Birth list (4 = "p1-4")
        sex             female
        dob             1.3.1672
        id              p1-4
        fname           Ann
        surname         Nicholls
        deaths          <deaths>Death list (3 = "p2-3") : Death list (3 = "p2-3") -> births

Birth list (5 = "p1-5")
        sex             female
        dob             23.6.1641
        id              p1-5
        fname           Anne
        surname         Nichols

Birth list (6 = "p1-6")
        sex             male
        dob             5.8.1632
        id              p1-6
        fname           An.
        surname         Nicalls

Birth list (7 = "p1-7")
        sex             male
        dob             10.3.1637
        id              p1-7
        fname           Stephen
        surname         Ridlingdon
        deaths          <deaths>Death list (4 = "p2-4") : Death list (4 = "p2-4") -> births
```

```
Birth list (8 = "p1-8")
        sex             male
        dob             23.1.1657
        id              p1-8
        fname           Steven
        surname         Ridlington
        deaths          <deaths>Death list (6 = "p2-6") : Death list (6 = "p2-6") -> births

Birth list (9 = "p1-9")
        sex             male
        dob             27.1.1609
        id              p1-9
        fname           George
        surname         ap Gryffith
        deaths          <deaths>Death list (2 = "p2-2") : Death list (2 = "p2-2") -> births
```

Notice that the additional element displayed here takes the name of the database first mentioned in the **`bridge`** declaration.

This element is an example of the data type **`relation`**. It denotes the logical connection between one arbitrary point in one database to one arbitrary point in another. The use of this new element will be fully described in the next chapter.

Further reading

S. W. Baskerville, P. Hudson & R. J. Morris (eds.), 'Record Linkage', special issue of *History and Computing*, 4:1 (1992).

G. Bouchard and C. Pouyez, 'Name Variations and Computerized Record Linkage', *Historical Methods*, 13 (1980), pp.119–125.

Charles Harvey, 'Record Linkage and Local History' in *Computers and Local History*, ed. A. Brown, P. Wakelin and K. Schürer (Leicester, Leicester UP & Association for History and Computing, 1993, forthcoming).

Theodore Hershberg, Alan Burnstein and Robert Dockhorn, 'Record Linkage', in *Historical Methods Newsletter*, IX (1975/76), pp.137–163.

A. Kitts, D. Doulton and E. Reis, *The Reconstruction of Viana do Castelo* (London, Association for History and Computing, 1990).

Howard B. Newcombe, *Handbook of Record Linkage* (Oxford, OUP, 1988) (not historical record linkage).

Ian Winchester, 'The Linkage of Historical Records by Man and Computer: Techniques and Problems', *Journal of Interdisciplinary History*, 1 (1970), pp. 107–124.

Ian Winchester, 'What Every Historian Needs to Know about Record Linkage for the Microcomputer Era', *Historical Methods*, 25 (1992), pp.149–165.

E. A. Wrigley (ed.), *Identifying People in the Past* (London, Edward Arnold, 1973).

Chapter 13

Relational capabilities

13.1 Introduction

This chapter is designed to help you to understand some of the relational capabilities of κλειω. κλειω is not known for its simplicity and this is where κλειω can start to become very difficult indeed. For this reason, this chapter has been made as simple as possible, with very simple made-up data. One of the main uses for this feature of κλειω was almost introduced in the previous chapter where the **`bridge`** command was used to 'join' two databases together. This, as we will demonstrate, is an example of the use of the relational capability of κλειω.

κλειω is neither a hierarchical nor a relational database. It has been described as "a semantic network tempered by practical considerations", which is all very well for computer scientists but less than helpful for historians. An explanation of sorts is in order. We have previously said that κλειω handles hierarchically structured data, which might have implied that κλειω handles data in a hierarchical fashion. It does not. To confuse matters further, it could be said that κλειω handles both hierarchically and relationally *structured* data. However the way in which κλειω handles data cannot be described as either. To the user it looks as though κλειω processes data in a fashion similar to the hierarchical model, but internally κλειω handles everything in a network structure. This allows it to possess the usual features of a relational DBMS while allowing the user to link entirely independent databases, rather than just join a number of tables of a database (as with a relational database). However, one could also join together disparate parts of a database, in a fashion similar to a relational database.

κλειω achieves its relational properties using a data type called **`relation`**. Data of this type can be used to represent references in any number of databases. As has been shown κλειω can create an 'artificial' element of this data type using the **`bridge`** command, which performs a similar task as a join between two tables in a standard relational database, except that κλειω does it between two databases that remain independent. κλειω can also use an element of **`relation`** data type to connect pieces of data within a single database.

κλειω has been designed as a source-oriented data processor. No matter how κλειω represents the underlying structure of a database for its own use, it aims to be able to process all the types of source likely to be considered by an historian. So whatever the structure of the primary material, κλειω should be able to process it. A word of warning, however. Use of the **`relation`** data type could become very unwieldy if it were used to construct a fully fledged relational database. It would be recommended to use the **`relation`** data type only at a low level or at least only with data that is inherently structured in such a fashion.

Before considering how κλειω can be used in such a way, we will reconsider some of the points we made about the **`bridge`** declaration at the end of the last chapter.

13.2 The `bridge` declaration

Ensure that ex12.21 has been run. The task contains the **`bridge`** declaration which links the two databases births and deaths.

The following task gives a result which demonstrates how κλειω processes information within a database once a bridge with another database has been made.

Example 13.1

```
query name=births
write
stop
```

Part of the result of this task is shown below:

```
Birth list (1 = "p1-1")
        sex             male
        dob             13.4.1622
        id              p1-1
        fname           Francis
        surname         Owens
        deaths          <deaths>Death list (5 = "p2-5") : Death list (5 = "p2-5") -> births
```

Here κλειω has made a new element called deaths. When two databases are joined together with a **bridge** declaration, a new element relating to the link between two elements is created. The name of this element is taken from the name of the database from which the information has come. The contents of this element will be explained shortly.

If you receive a rather different result to this, it is possible that you have created the bridge between the two databases twice. If this happens you should destroy it and create a new one. The command necessary to delete the bridge between two databases is **delete**.

```
delete name=deaths;usage=bridge;source=births;type=permanent
stop
```

This asks κλειω to delete a bridge called bur which belongs to the database bap, and is permanent. As the bridge links two databases, it would be perfectly possible to destroy this bridge with the following command:

```
delete name=births;usage=bridge;source=deaths;type=permanent
stop
```

This would be possible because the **bridge** declaration creates double-sided links and both are destroyed by one command. Using the metaphor of the bridge crossing a river, it would be impossible to destroy only half a bridge (no letters about Avignon please).

The following example has been extracted from a task similar to ex13.1, but using the database deaths as the target database rather than the database births.

```
Death list (5 = "p2-5")
        sex          male
        age          52years
        id           p2-5
        fname        Frank
        surname      Owen
        link         p1-1
        births       <births>Birth list (1 = "p1-1") : Birth list (1 = "p1-1") -> deaths
```

Both of these results contain an element that was not previously in the databases.

Bridges between two databases are maintained by this element. Note, moreover, that these elements contain a different type of data than we have encountered before. They contain data of the type **relation**.

In order to describe what this element holds it is simplest to break down its contents into parts:

`births`	the name of the element
`<births>`	the name of the 'other' database
`Birth list`	the name of the document from which we are going
`(1 = "p1-1")`	the identification of that document
`Birth list`	the name of group from which we are going
`(1 = "p1-1")`	the identification of the group from which we are going
`-> deaths`	the name of the database to which we are going

This example is not entirely clear, as the identifications of the groups that we are using have the same style as the identifications of the documents we are using. In order to distinguish this we have created a simple database which should make the difference between the identification of the groups and documents more clear.

13.2.1 Another example of a `bridge` declaration

Look at trial.lnk and trial2.lnk. Both of these databases are supposed to contain the same imaginary people. To make this database slightly more realistic for more complicated (or ordinary) projects, we have allotted the information relating to the people in the first of these databases to the second document. We have also asked κλειω to use the contents of the element id as part of the identification for that group. Before compiling both of these databases, we suggest that you read through them.

```
note trial.lnk
database name=trial;first=ref;overwrite=yes
part name=ref;
   part=a;
   position=refnum
part name=a;
   position=id,name,link
element name=id;identification=yes
exit name=trial

read name=trial
ref$First_document
ref$Second_document
a$1/Matthew
a$2/Mark
a$3/Luke
a$4/John
```

```
note trial2.lnk
database name=trial2;first=ref;overwrite=yes
part name=ref;
   part=b;
   position=refnum
part name=b;
   position=id,name
element name=id;identification=yes
exit name=trial2

read name=trial2
ref$xxx
b$5/John
b$6/Luke
b$7/Mark
b$8/Matthew
```

The links between these two databases are not at the document level (as they are in the births and deaths databases), and we have changed the identifications of the element id. The **bridge** declaration is thus slightly more complicated. A full explanation of what is happening here can be found in the Reference Manual in Section 10.1.1.2.1.2. For the time being, though, just remember that people with the same names are supposed to be linked.

```
note bridge.lnk
bridge first=trial;second=trial2;type=definitions
bridge first="Second_document/a=1";second="xxx/b=8"
bridge first="Second_document/a=2";second="xxx/b=7"
bridge first="Second_document/a=3";second="xxx/b=6"
bridge first="Second_document/a=4";second="xxx/b=5"
exit name=definitions
```

Once you have compiled both the databases and the bridge definition, run the following task:

Example 13.2

```
query name=trial
write
stop
```

This should produce the following result:

```
ref (1 = "First_docume")
        refnum          First_document

ref (2 = "Second_docum")
        refnum          Second_document
```

```
a (1 = "1")
        id          1
        name        Matthew
        trial2      <trial2>ref (1 = "xxx") : b (4 = "8") -> trial

a (2 = "2")
        id          2
        name        Mark
        trial2      <trial2>ref (1 = "xxx") : b (3 = "7") -> trial

a (3 = "3")
        id          3
        name        Luke
        trial2      <trial2>ref (1 = "xxx") : b (2 = "6") -> trial

a (4 = "4")
        id          4
        name        John
        trial2      <trial2>ref (1 = "xxx") : b (1 = "5") -> trial
```

If we run through the contents of the element trial2 as we did for the births element it should make it clearer what they refer to.

```
trial2    <trial2>ref (1="xxx") : b: (1 = "5") -> trial
  |          |     |      |        |      |         |
  |          |     |      |        |      |        name of the element
  |          |     |      |        |      |        to which we are
  |          |     |      |        |      |        going
  |          |     |      |        |      |
  |          |     |      |        |      |
  |          |     |      |        |     identification of the group from
  |          |     |      |        |     which we are going
  |          |     |      |        |
  |          |     |      |       name of the group from which we are
  |          |     |      |       going
  |          |     |      |
  |          |     |     identification of that document
  |          |     |
  |          |    name of the document from which we are going
  |          |
  |         name of the other database
  |
 the name of the newly created element
```

While you look at the above diagram, look at the structure declarations and the data files for the two databases. What κλειω is doing here must be understood before we continue. Not only does this have to be understood, we must demonstrate that this newly created element can be used in exactly the same way as other elements.

13.3 The `relation` data type

For the new sample database, it might be interesting to see the results of this task. Notice that the new element (trial2) is included as a value of the **`part=`** parameter of the **`write`** directive. The task below:

Example 13.3

```
query name=trial;part=:name
write part=:name,:trial2
stop
```

produces a result like this:

```
ref (2 = "Second_docum") : name      Matthew
ref (2 = "Second_docum") : trial2    <trial2>ref (1 = "xxx") : b (4 = "8") -> trial
ref (2 = "Second_docum") : name      Mark
ref (2 = "Second_docum") : trial2    <trial2>ref (1 = "xxx") : b (3 = "7") -> trial
ref (2 = "Second_docum") : name      Luke
ref (2 = "Second_docum") : trial2    <trial2>ref (1 = "xxx") : b (2 = "6") -> trial
ref (2 = "Second_docum") : name      John
ref (2 = "Second_docum") : trial2    <trial2>ref (1 = "xxx") : b (1 = "5") -> trial
```

The same task can also be applied to the database we used in the last chapter. Look at the result of the following task:

```
query name=bap;part=:surname
write part=:surname,:bur
stop
```

This demonstrates that this newly made element can be treated in the same way as any other element. However, it also has a property which other elements in κλειω do not have.

Access paths in κλειω usually take the following form:

```
group1/group2/group3....groupn:element
```

However if the element is of the **`relation`** data type, then another element can be added to the end of this; allowing the user to specify the element within the group that the bridge points to. For example:

```
group1/group2/group3....groupn:relation:element
```

Thus in the following task κλειω is asked to use the database births, go to each group where there is an element **`:surname`**, then to write that surname, and then write the surname found by following the path from the same group via the element **:deaths** to the database <deaths>.

Example 13.4

```
query name=births;part=:surname
write part=:surname,:deaths<deaths>:surname
stop
```

Notice the name of the second database must also be specified and it must be specified within angled brackets. (Unlike the task of joining two databases, though, here we do not need to specify an **also=** parameter.)

A single entry from the result is given here:

```
Birth list (1 = "p1-1") : surname       Owens
> Birth list (1 = "p1-1") :  ---> <deaths>Death list (5 = "p2-5") : > >
Death list (5 = "p2-5") : surname       Owen
<
```

This result tells us that a person with the surname Owens in the births database is linked (via the bridge) to a person called Owen in the deaths database.

Notice that at the end of this entry there is a **<** which signifies the end of that result.

Use of this 'artificial' element can be made wherever an element is normally used, for example in a task using the **index** command. See also the example ex12.6.

Example 13.5

```
query name=births;part=:surname
index part=:surname;
   part=:deaths<deaths>:surname
stop
```

The result is:

```
ap Gryffith Griffiths p1-9
Nicalls   p1-6
Nicholls Nichols p1-4
Nichols   p1-5
Owen Owen p1-2
Owens   p1-3
Owens Owen p1-1
Ridlingdon Riddington p1-7
Ridlington Ridington p1-8
```

Example 13.6

```
options lines=0
query name=births;part=:surname
index part=:surname;signs=12;
   part=:fname;signs=12;
   part=:deaths<deaths>:surname;signs=12;
   part=:deaths<deaths>:fname;signs=12;
   identification=order[];write=order[]
stop
```

Exercise 13.1

Create an index which contains entries looking like this, where the elements come from the databases indicated below:

Griffiths George was born on 1609.01.27

<*bur*>   <*bur*> was born on <*bap*>

In all the examples shown in this chapter so far, two databases have been connected using an element of the data type **`relation`**. In these examples the connecting points between the two databases had to be specified explicitly using the **`bridge`** command, but it is possible to link groups within a single database without explicitly specifying where the links are to be made. Obviously the use of this will depend on the type of historical source material that is being used, but beware; while most historical data fits into a hierarchical structure, almost all historical data could also be forced into a relational structure. κλειω has exceptionally powerful relational capabilities which should be used whenever necessary, but they introduce an additional level of complexity and trying to use κλειω to implement the relational model will be very cumbersome. Therefore the relation data type should only be used if the data really requires it.

13.4 The `relation` data type in a single database

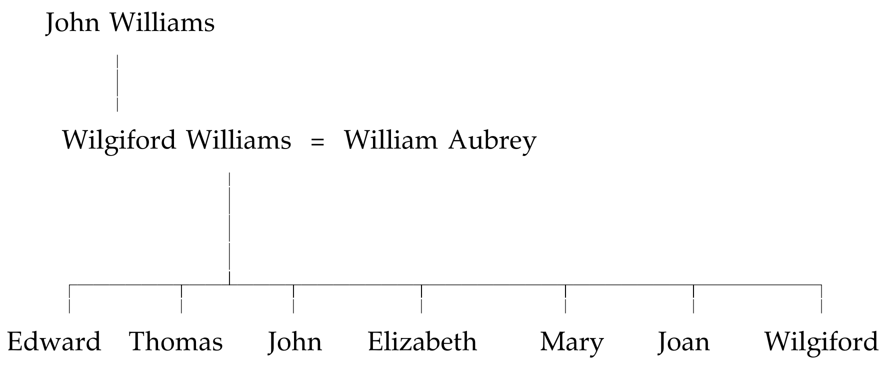
The example database that will be used to demonstrate further uses of the **`relation`** data type is made from a family tree and other written sources pertaining to the life of the antiquarian and biographer John Aubrey.

The data and the structure declaration of the database can be found in aubrey.dat.

The following lines of data represent two families. The first family (001) is made up of a man (William Aubrey), his wife (Wilgiford Aubrey) and their children. The second is made up of a man and his child. If you find it hard to understand what is supposed to be represented here, refer back to Chapter 3 where there is a family tree of John Aubrey.

```
fam$001
m$William/Aubrey/#Oxon/0 0 1529/0 0 1595
w$Wilgiford/Aubrey/Williams///002
c$Edward/Aubrey///003
c$Thomas/Aubrey///004
c$John/Aubrey/0.0.1578/0.0.1616/005
c$Elizabeth/Aubrey
c$Mary/Aubrey
c$Joan/Aubrey
c$Wilgiford/Aubrey

fam$002
m$John/Williams/Teynton#Oxon
c$Wilgiford/Williams///001
```

In family tree form the relationships shown above could be displayed thus:



The second family in the datafile has a group called fam which contains one element familynum. In this case the element contains the identification 002.

Each group relating to a co-founder of a family (husband or wife, denoted by the groups m and w) contains, amongst other elements, one element called parfnum. The element corresponds to the familynum of the family in which they were born. In this case Wilgiford Williams (as mentioned in family 001) has the element parfnum with contents of 002. This refers us to family 002, where she can be found as a child.

Similarly each group relating to a child (denoted by the group c) contains an element famnum, the contents of which is always the same as the familynum of the family that they go on to found. In this case, Wilgiford Aubrey (as mentioned in family 002) is the same woman who later co-founded family 001.

In plain English, these relationships could be described as follows:

```
In family 002 Wilgiford Williams (a child) later co-founded family 001 and
in family 001 Wilgiford Aubrey (a wife) was born in family 002.
```

From the information given in that part of the file displayed above we can also say that Edward Aubrey (a child in family 001) later co-founded family 003; and from the same family Thomas Aubrey later co-founded family 004 and John Aubrey later founded family 005.

If this still seems confusing, please look at the whole family tree in Chapter 3 and look back to the file aubrey.dat.

The structure declaration for this database can also be found in aubrey.dat.

There is a logical object which will look unfamiliar.

```
item name=family;usage=relation
part type=permanent;part=no
exit name=family
```

This defines a logical object called family which deals with **`relation`** type data within a database. This type of declaration tells κλειω how to treat non-hierarchical relationships within a database. The **`type=permanent`** parameter tells κλειω to keep a file of raw data to create the database incrementally. The **`part=no`** parameter also relates to this; it tells κλειω not to integrate any of the network identities until it is specifically told to. This line is not necessary here, but is included to demonstrate two potential problems regarding the **`relation`** data type. It would be wise for any user who was not very familiar with this mechanism to follow this procedure.

The following **`element`** command within the structure declaration of this database tells κλειω that the elements familynum, famnum and parfnum are all of data type **`relation`** and are operated upon by a logical object called family.

```
element name=familynum,famnum,parfnum;type=relation;
  relation=family
```

Compile the database now, and then run the following task:

Example 13.7

```
query name=aubrey;part=m
write part=:each[]
stop
```

Part of the result looks like this:

```
fam (3 = "003") : m (1 = "m-1")
        parfnum         *  (Unknown)
        fname           Edward
        surname         Aubrey
        place           Brecon
```

Compare this with the information in the data file relating to Edward Aubrey in family 003. Notice that within the data file there *is* information referring to the parfnum of Edward Aubrey. The reason why κλειω has suggested that this information is unknown is because κλειω was specifically told in the logical object family not to integrate any of the network identities until it was told to do so. Therefore it will be necessary to integrate these network identities. This can be done with the following command:

Example 13.8

```
relation name=family;source=aubrey;cumulate=yes
```

which in English means "within the database aubrey, put the logical object family into operation". This will create the appropriate links between the elements familynum, famnum and parfnum.

If this task is run a further time, the result will be quite different.

Example 13.9

```
query name=aubrey;part=m
write part=:each[]
stop
```

```
fam (3 = "003") : m (1 = "m-1")
        parfnum         fam (1 = "001") : fam (1 = "001") -> familynum
                          fam (2 = "002") : c (1 = "001") -> famnum
                          fam (4 = "004") : m (1 = "m-1") -> parfnum
                          fam (5 = "005") : m (1 = "m-1") -> parfnum
        fname           Edward
        surname         Aubrey
        place           Brecon
```

It is absolutely crucial to understand exactly what this means. Some methods of clarifying this information will be demonstrated below.

The information in the first line of this result shows that we are dealing with the first m (man) in the third family (known as 003). The beginning of the next line tells us that there is some information relating to the parfnum of the man in this family. The remainder of that line and the subsequent three show that there are links between the parfnum of the man in family 003 and other families.

The first of these, **`fam (1 = "001") : fam (1 = "001") -> familynum`**, says that the parfnum of the man in family 003 is the same as the familynum of family 001.

The second, **`fam (2 = "002") : c (1 = "001") -> famnum`**, says that the parfnum of the man in family 003 is the same as the famnum of a child in family 002.

The third, **`fam (4 = "004") : m (1 = "m-1") -> parfnum`**, says that the parfnum of the man in family 003 is the same as the parfnum of the man in family 004.

Finally, the fourth line **`fam (5 = "005") : m (1 = "m-1") -> parfnum`**, says that the parfnum of the man in family 003 is the same as the parfnum of the man in family 005.

This can be shown in the following diagram:

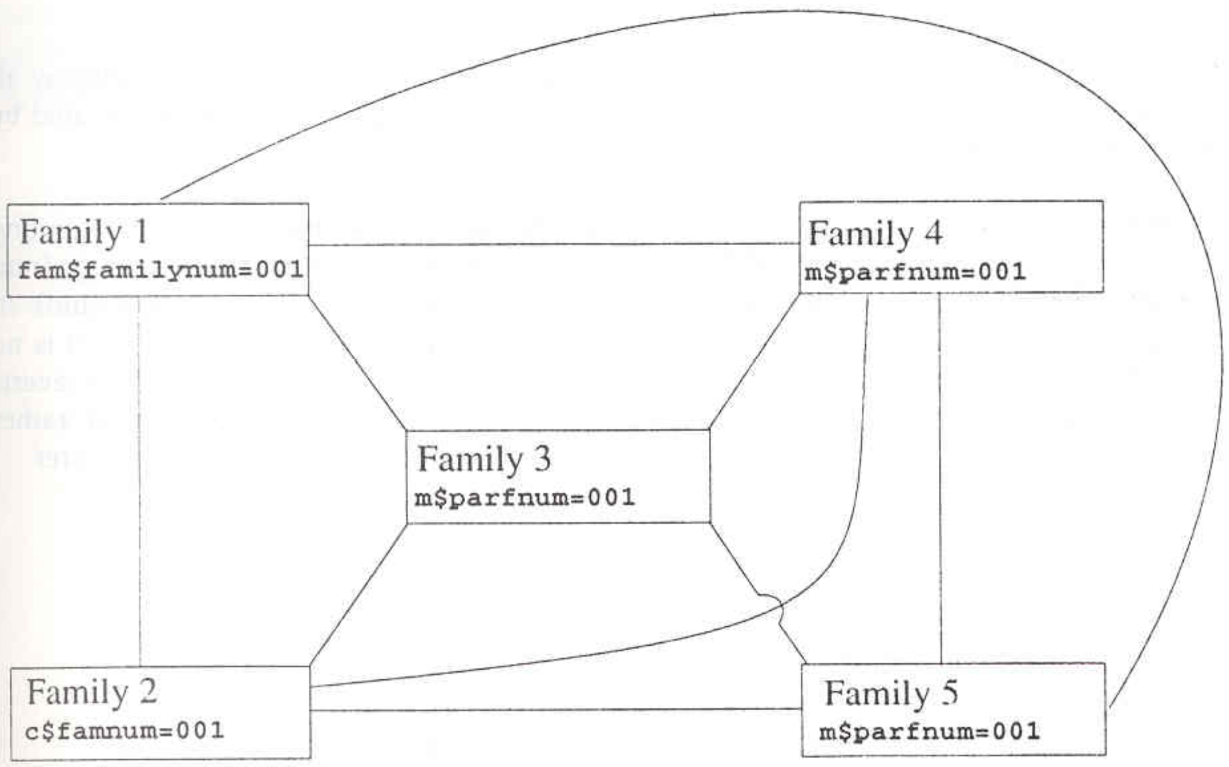


Figure 13.1

If you look at the original data you will see that that number occurs four times in the database (not including where it is given as the parfnum of the third family).

Put into the terminology used above, these four lines represent the following information.

1) Family 001 has the same familynum as the parfnum of the man we are looking at, therefore he was born in this family.

2) In family 002 a child later co-founded the family where the man in family 003 was a child, therefore this child is a parent of the man we are looking at.

3) The man in family 004 has the same parfnum as the man we are looking at, therefore they are brothers.

4) The man in family 005 has the same parfnum as the man we are looking at, therefore they are brothers.

13.4.1 Displaying more information

The following task asks κλειω to perform the same task as above, and also to display all the information relating to each of the groups found moving along the paths created by the **`relation`** element.

Normally, when κλειω executes a task, it moves through the database from the beginning to the end in a linear fashion. In the task reproduced below the system is asked to go from one element to another (i.e. wherever a particular parfunum element can be found). In doing this, κλειω is no longer processing in a linear fashion, and indicates that it is no longer doing so by displaying an arrow which 'points' to its target. When κλειω reverts to linear processing the arrows are reversed (and shorter). This may sound rather confusing at the moment but as you read the following section it will become clearer.

Example 13.10

```
query name=aubrey;part=m
write part=:each[],:parfnum:each[]
stop
```

Part of the result is annotated below to show what each part means:

```
fam (3 = "003") : m (1 = "m-1")
        parfnum        fam (1 = "001") : fam (1 = "001") -> familynum
                         fam (2 = "002") : c (1 = "001") -> famnum
                         fam (4 = "004") : m (1 = "m-1") -> parfnum
                         fam (5 = "005") : m (1 = "m-1") -> parfnum
        fname          Edward
        surname        Aubrey
        place          Brecon
```

This represents the first execution of the **`:each[]`** element function.

```
> fam (3 = "003") :  ---> fam (1 = "001") : > >
```

This represents 'moving' from family 003 to family 001.

```
fam (1 = "001")
        familynum         fam (2 = "002") : c (1 = "001") -> famnum
                        fam (3 = "003") : m (1 = "m-1") -> parfnum
                        fam (4 = "004") : m (1 = "m-1") -> parfnum
                        fam (5 = "005") : m (1 = "m-1") -> parfnum
```

The information about all the elements in that group are now displayed, fulfilling the command **`:parfnum:each[]`**

```
>  -- " --  ---> fam (2 = "002") : > >
```

indicates that κλειω has returned to family 003 (shown by the ditto marks) and is now moving to family 002. This process continues until there is no further relevant information to display.

```
fam (2 = "002") : c (1 = "001")
        famnum          fam (1 = "001") : fam (1 = "001") -> familynum
                          fam (3 = "003") : m (1 = "m-1") -> parfnum
                          fam (4 = "004") : m (1 = "m-1") -> parfnum
                          fam (5 = "005") : m (1 = "m-1") -> parfnum
        fname           Wilgiford
        surname         Williams

>  -- " --  ---> fam (4 = "004") : > >

fam (4 = "004") : m (1 = "m-1")
        parfnum         fam (1 = "001") : fam (1 = "001") -> familynum
                          fam (2 = "002") : c (1 = "001") -> famnum
                          fam (3 = "003") : m (1 = "m-1") -> parfnum
                          fam (5 = "005") : m (1 = "m-1") -> parfnum
        fname           Thomas
        surname         Aubrey
        place           Glamorgan

>  -- " --  ---> fam (5 = "005") : > >
```

```
fam (5 = "005") : m (1 = "m-1")
        dob             0.0.1578
        dod             11.6.1616
        parfnum         fam (1 = "001") : fam (1 = "001") -> familynum
                          fam (2 = "002") : c (1 = "001") -> famnum
                          fam (3 = "003") : m (1 = "m-1") -> parfnum
                          fam (4 = "004") : m (1 = "m-1") -> parfnum
        fname           John
        surname         Aubrey
        place           Burleton
         comment        Hereford
<
```

and as there are no further places to go, no further information can be displayed. κλειω represents this with a <.

Thus κλειω has been asked to display all of the information about all of the groups that are linked with a group by a **`relation`** type element. (Remember that here we are only looking at a single individual within the database, rather than all of the people contained in the result.) Try following through the above with a different person within the database – looking at the results of the previous two queries using John Aubrey (the head of family 014) as an example.

How useful is all this information to us? The previous task displayed information relating to all the people immediately related to all the men within the database. If we only wanted information about the fathers of men in the database we could specify this using a different task. Remember that in the case of the man in family 003 there were four links to other groups in the database:

```
fam:familynum      m:parfnum
          \            /
           fam$003
          /            \
   c:famnum        m:parfnum
```

13.4.2 The element function **`:target[]`**

If we could find a way to specify only some of these links or connection lines, it would be possible to display only certain information about people related to an individual. This is achieved using the element function **`:target[]`**. This function tells κλειω to go to all the elements in the database that are the same as those specified in the group function. In the example below, κλειω is asked to follow the connections found only if they relate to **`fam:familynum`**.

Example 13.11

```
query name=aubrey;part=m
write part=:each[],:target[:parfnum,"fam:familynum"]:each[]
stop
```

For the example we have been using above, the following information is given as a result:

```
fam (3 = "003") : m (1 = "m-1")
        parfnum       fam (1 = "001") : fam (1 = "001") -> familynum
                        fam (2 = "002") : c (1 = "001") -> famnum
                        fam (4 = "004") : m (1 = "m-1") -> parfnum
                        fam (5 = "005") : m (1 = "m-1") -> parfnum
        fname         Edward
        surname       Aubrey
        place         Brecon
> fam (3 = "003") :  ---> fam (1 = "001") : > >

fam (1 = "001")
        familynum       fam (2 = "002") : c (1 = "001") -> famnum
                      fam (3 = "003") : m (1 = "m-1") -> parfnum
                      fam (4 = "004") : m (1 = "m-1") -> parfnum
                      fam (5 = "005") : m (1 = "m-1") -> parfnum
<
```

For another member of the family the result looks like this:

```
fam (14 = "014") : m (1 = "m-1")
        dob           12.3.1626
        dod           1.1.1500 - 7.6.1697
        parfnum       fam (5 = "005") : c (1 = "011") -> famnum
                        fam (11 = "011") : fam (11 = "011") -> familynum
                        fam (13 = "013") : c (1 = "011") -> famnum
        fname         John
        surname       Aubrey
        place         Easton Pierce
> fam (14 = "014") :  ---> fam (11 = "011") : > >

fam (11 = "011")
        familynum      fam (5 = "005") : c (1 = "011") -> famnum
                      fam (13 = "013") : c (1 = "011") -> famnum
                      fam (14 = "014") : m (1 = "m-1") -> parfnum
<
```

This refers to *the* John Aubrey. The first part of this result shows that he has two siblings, one the head of family 011 and one the head of family 013, and that he was born in family 011. The second part of the result shows the links from family 011, which include the link back to John Aubrey in family 014.

This second part of this result gives information about the links from the family. The amended task, shown below, gives information about the links from the man in that family.

If we want information about the father of John Aubrey, we have to specify that request. As the group m (which contains the data about the husband in the family) is dependent on the group fam, the access path to that group must be specified.

Example 13.12

```
query name=aubrey;part=m
write part=:each[],:target[:parfnum,"fam:familynum"]/m:each[]
stop
```

Line by line, this task means:

query name=aubrey;part=m
I am interested in the database aubrey and I am only interested in that part where there is information about the group m.

write part=:each[],
display on the screen all the information about the group specified in the line above.

:target[:parfnum,"fam:familynum"]
move through the database along any link where the element :parfnum (for the individual displayed previously) is the same as a familynum in a group fam.

/m:each[]
display all the information about the subordinate group m for all those families which fulfil the conditions above.

The result of this task for John Aubrey of family 014 is shown below.

```
fam (14 = "014") : m (1 = "m-1")
        dob             12.3.1626
        dod             1.1.1500 - 7.6.1697
        parfnum         fam (5 = "005") : c (1 = "011") -> famnum
                          fam (11 = "011") : fam (11 = "011") -> familynum
                          fam (13 = "013") : c (1 = "011") -> famnum
        fname           John
        surname         Aubrey
        place           Easton Pierce
> fam (14 = "014") :  ---> fam (11 = "011") : > >
```

```
fam (11 = "011") : m (1 = "m-1")
        dob             0.0.1603
        dod             0.0.1652
        parfnum         fam (1 = "001") : c (3 = "005") -> famnum
                          fam (5 = "005") : fam (5 = "005") -> familynum
                          fam (12 = "012") : c (1 = "005") -> famnum
        fname           Richard
        surname         Aubrey
        place           Burleton
         comment        Heref
<
```

What is displayed here is the information about the father of John Aubrey. In the same way we can ask κλειω to give us information about the father of Richard Aubrey:

Example 13.13

```
query name=aubrey;part=m
write part=:each[],
        :target[:parfnum,"fam:familynum"]/m
        :target[:parfnum,"fam:familynum"]/m:each[]
stop
```

The result of this task for John Aubrey of family 014 looks like this:

```
fam (14 = "014") : m (1 = "m-1")
        dob             12.3.1626
        dod             1.1.1500 - 7.6.1697
        parfnum         fam (5 = "005") : c (1 = "011") -> famnum
                          fam (11 = "011") : fam (11 = "011") -> familynum
                          fam (13 = "013") : c (1 = "011") -> famnum
        fname           John
        surname         Aubrey
        place           Easton Pierce
> fam (14 = "014") :  ---> fam (11 = "011") : > >
> > fam (11 = "011") :  ---> fam (5 = "005") : > > >

fam (5 = "005") : m (1 = "m-1")
        dob             0.0.1578
        dod             11.6.1616
        parfnum         fam (1 = "001") : fam (1 = "001") -> familynum
                          fam (2 = "002") : c (1 = "001") -> famnum
                          fam (3 = "003") : m (1 = "m-1") -> parfnum
                          fam (4 = "004") : m (1 = "m-1") -> parfnum
        fname           John
        surname         Aubrey
        place           Burleton
         comment        Hereford
< <
<
```

As the task did not specify displaying the results of the first move along the links, only all the information about John Aubrey's grandfather is displayed here.

The curious lines shown after the first part of the result show that κλειω is moving from family 014 to family 011 and then from family 011 to family 005.

13.4.3 The group function **`continue[]`**

Obviously if there is enough information about ancestors it would be possible to continue specifying the element function **`:target[]`** as many times are there are potential ancestors. In fact κλειω has a built-in function which will continue to perform this over and over again. It works in a similar way to the **`:target[]`** element function. The task below shows how one would have to use the **`:target[]`** function to display information about four generations

Example 13.14

```
query name=aubrey;part=m
write part=:each[],
   :target[:parfnum,"fam:familynum"]/m:each[],
   :target[:parfnum,"fam:familynum"]
      /m:target[:parfnum,"fam:familynum"]/m:each[],
   :target[:parfnum,"fam:familynum"]
      /m:target[:parfnum,"fam:familynum"]
      /m:target[:parfnum,"fam:familynum"]/m:each[]
stop
```

The reason why a task like that below will not work is because all relational 'jumps' are cancelled by the end of a subpath (i.e. the comma):

```
query name=aubrey;part=m
write part=each[],
         :target[:parfnum,"fam:familynum"]/m:each[],
         :target[:parfnum,"fam:familynum"]/m
         :target[:parfnum,"fam:familynum"]/m:each[]
stop
```

The group function **`continue[]`** is a fully recursive function, which in effect 'cruises' through all the networks within a database. It is especially useful if we do not know how far these networks stretch. In the task above we start at the group m, and each time we reach this group κλειω displays each element it finds there. After that has been displayed κλειω goes to the parental family number, checks that it relates to a family number in the group fam. If it does, κλειω moves down one level of the hierarchy and displays each of the elements contained in the group m. This will be repeated as often as possible using **`continue[]`**.

Example 13.15

```
query name=aubrey;part=fam
write part=continue[m,:parfnum,"fam:familynum"]:each[]
stop
```

The result shows John Aubrey, his father Richard, his grand-father John and his great grand father William.

```
fam (14 = "014") : m (1 = "m-1")
        dob             12.3.1626
        dod             1.1.1500 - 7.6.1697
        parfnum         fam (5 = "005") : c (1 = "011") -> famnum
                          fam (11 = "011") : fam (11 = "011") -> familynum
                          fam (13 = "013") : c (1 = "011") -> famnum
        fname           John
        surname         Aubrey
        place           Easton Pierce
> fam (14 = "014") :  ---> fam (11 = "011") : > >

fam (11 = "011") : m (1 = "m-1")
        dob             0.0.1603
        dod             0.0.1652
        parfnum         fam (1 = "001") : c (3 = "005") -> famnum
                          fam (5 = "005") : fam (5 = "005") -> familynum
                          fam (12 = "012") : c (1 = "005") -> famnum
        fname           Richard
        surname         Aubrey
        place           Burleton
         comment        Heref
> > fam (11 = "011") :  ---> fam (5 = "005") : > > >

fam (5 = "005") : m (1 = "m-1")
        dob             0.0.1578
        dod             11.6.1616
        parfnum         fam (1 = "001") : fam (1 = "001") -> familynum
                          fam (2 = "002") : c (1 = "001") -> famnum
                          fam (3 = "003") : m (1 = "m-1") -> parfnum
                          fam (4 = "004") : m (1 = "m-1") -> parfnum
        fname           John
        surname         Aubrey
        place           Burleton
         comment        Hereford
> > > fam (5 = "005") :  ---> fam (1 = "001") : > > > >
```

```
fam (1 = "001") : m (1 = "m-1")
        dob             0.0.1529
        dod             0.0.1595
        fname           William
        surname         Aubrey
        place
         comment        Oxon
< < <
< <
<
```

Summary

This chapter has introduced:

- The basic relational facilities of κλειω
- Further information on the **`bridge`** declaration
- The **`relation`** datatype
- The functions **`:target[]`** and **`:collect`**.

Chapter 14

Mapping

14.1 Introduction

κλειω has the ability to produce graphical representations of topographical objects. However, in order to produce maps κλειω needs to have topographical information presented to it in a particular fashion. Some of the examples presented in this chapter have been produced using digitizing software to which you may not have access, or indeed you may have access to digitizing equipment that κλειω does not yet support. Once you have worked through this chapter and you find that you would like to produce your own maps using κλειω it would be of value to consult the Reference Manual (Section 7.3.1.13.1.4.1 ff.) to see more clearly the different methods of supplying κλειω with topographical objects.

Producing a map using κλειω is intrinsically no different to producing a result from any other task in κλειω. When κλειω produces the result of a task it normally displays the information on the screen in a format which the user has specified. Alphanumeric information of this kind is displayed with alphanumeric characters on the screen, while data defined as topographical objects are displayed spatially. Thus databases containing both 'historical' and 'geographical' data can produce information in the form of a map.

14.1.1 The `location` data type

The file ex14.1 contains the structure declaration for a database called dummy1.

Example 14.1

```
database name=dummy1;first=x
element name=grid-reference;type=location;location=map

part name=x;
     position=id,name,grid-reference
exit name=dummy1
```

The document for this database is x, and it contains the elements id, name and grid-reference. The **`element`** declaration states that the element grid-reference is of data type **`location`** and that these elements should be operated upon by rules contained in an (as yet unseen) logical object called map.

This introduces a new data type, **`location`**. The data type **`location`** refers to data which κλειω must be able to convert into topographical objects.

So far the data types introduced are **`text`**, **`number`**, **`date`**, **`category`** and **`relation`**. The first four of these are relatively common in κλειω databases, the fifth is much rarer. There are two other types of data, **`location`** and **`image`**. The first five data types are machine independent; whatever hardware is being used, κλειω will treat them in the same way — but the other two data types are machine dependent and κλειω will treat the results of any task involving either of them slightly differently depending on the type of computer being used.

The file ex14.3 contains the data for this simple database. There is one group, x, which has an id of a1, the name John Smith and the grid-reference house1.

Example 14.3

```
read name=dummy1
x$a1/John Smith/house1
```

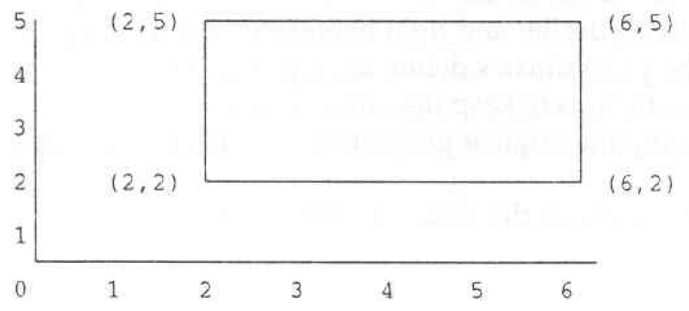
If you have already run ex14.1 and attempt to run ex14.3, an error message will appear which states that an implicitly addressed logical object is missing. This implicitly addressed logical object is called map, and is of type **`location`**. It is introduced in the file ex14.1. Therefore κλειω is telling you that no logical object called map exists, but it has already been implicitly addressed in the structure declaration of the database.

Look at the file ex14.2:

Example 14.2

```
item name=map;usage=location;source=dummy1;type=permanent
type form=arcinfo
location first=house1
2 2
6 2
6 5
2 5
end
exit name=map
```

This file contains the logical object called map. It understands data of type **`location`**, is related to the database dummy1 and is permanent. (Please ignore the **`type`** directive and the **`form=`** parameter for the time being. They will be described in Section 14.2.2.) The **`location`** directive is used to define one (and exactly one) topographical object. The parameter **`first=`** with a **`location`** directive takes the user-defined name of the topographical object. In this case its value is house1 (refer to ex14.3). On the following four lines are four pairs of coordinates which relate to the four points of the object house1. The keyword **`end`** is used to tell κλειω that these are the only four coordinates used to define the topographical object house1. The following diagram represents house1.



Each of the four 'corners' of house1 are represented in the logical object map. The points in the logical object are simple *(x,y)* coordinates, where *x* represents the horizontal axis and *y* the vertical axis.

κλειω understands these four coordinates as an object, consisting of four lines, one linking the point (2,2) with (6,2), another linking (6,2) with (6,5), a third joining (6,5) with (2,5) and finally one joining (2,5) with (2,2). In the **`location`** directive all four points are distinguished, but the lines are not. The sequence of coordinates tells κλειω where the joining lines should be. In this case, the first line links the first and second coordinates, the second line joins the second and third coordinates, the third line joins the third and fourth, and, importantly, the fourth line is represented as being between the fourth and the first coordinate.

14.1.2 The **`mapping`** command

The task ex14.4 displays the information contained in the element grid-reference, where the element name in the database dummy1 contains the character string Smith.

Exercise 14.4

```
query name=dummy1;part=:name="Smith"
mapping part=:grid-reference
stop
```

The new command here, **`mapping`**, tells κλειω what to map. It is rather like the **`write`** command, except that only data of **`location`** data type can be used in a **`mapping`** command. Thus the **`part=`** parameter of a **`mapping`** command *must* have the name of an element which refers to the **`location`** data type. (This does not mean that the name of a group can not appear in part of a path).

Run this file and wait for the map to be displayed on the screen.

κλειω produces a 'map', containing one object, which fills the screen. By default κλειω, whenever producing a map, makes maximum use of the screen. In effect the smallest and largest values for *x* coordinates define the left and right boundaries for the map, while the smallest and largest values for the *y* coordinates define the top and bottom boundaries of the map. However, κλειω always attempts to keep the units of measurement for both axes exactly the same, thereby preserving the original proportions of the objects displayed.

By default κλειω also displays the name of the object shown on the screen, and displays the object itself in white.

The map will remain on your screen until you press a key (almost any key will do). As on some national keyboards certain national characters may not have this effect, we would propose that you use the Enter key to clear the map from the screen.

14.1.3 The **`delete`** command

The use of an **`item`** command which is explicitly referred to in a structure declaration, but which is in a different file to the database, is unusual. So far we have only once seen this type of logical object in a file separate from the structure declaration. It is usually convenient to have them together because this type of logical object must be in existence whenever a database is used. However, in the case of logical objects dealing with the **`location`** data type it is usually not convenient to have them together, as the definitions found in a logical object referring to **`location`** data are much more complicated than those dealing with other data types. If this logical object is kept separately, it can be altered without the need to recompile the whole database. (Incidentally this is also true for the data of **image** data type.) Consider the following task:

Example 14.5

```
delete name=map;usage=location;source=dummy1;type=permanent
item name=map;usage=location;source=dummy1;type=permanent
type form=arcinfo
location first=house1
2 2
16 2
16 5
2 5
end
exit name=map
```

There is a slight change in coordinates, but rather than recompiling the structure file, the data file and the location file, it is possible, using the **delete** command, just to remove the old definition of the logical object and replace it with a new one. Even with this very small database, it is quicker to add this line and run this task than to re-run all three files. Imagine the time saved if one had a map with a thousand objects.

Run task ex14.5, and then task ex14.4, again. This will show what changes have been made to the map. You may wonder why in this case you have to delete the defintion explicitly rather than relying on the usual **overwrite=yes** parameter. If a logical object already exists, the **item** command would 'modify' the contents of the logical object, which in the case of large topographical objects could speed up single changes considerably. (Try to remember how we modified codebooks (p. 191 above) to understand this concept.) For smaller applications the way described above is, however, much easier to learn.

14.1.4 Parameters used to control graphical output

There are a number of other parameters used in conjunction with the **mapping** command to control the graphical output from a task. Three of these parameters are shown in task ex14.6:

Example 14.6

```
query name=dummy1;part=:name="Smith"
mapping part=:grid-reference;colour=red;line=triple;
  usage=tenthtone
stop
```

The **colour=** parameter

The **colour=** parameter (which could also be spelled **color=**. Note that κλειω only looks at the first four characters of any command) takes four possible parameter values: **contrast**, **red**, **blue**, and **green**. **contrast** is the default, which κλειω understands as producing a line in white (on the screen) or black (on paper output). The parameter values **red**, **blue**, and **green** produce output (either to screen or on paper via a plotter or colour printer) in the particular colour specified.

The **line=** parameter

The **line=** parameter accepts three possible keywords: **simple**, **double** and **triple**. Each of these parameter values refers to the thickness of the lines joining the coordinates. The value **simple** produces a line of single thickness. (The actual thickness is equal to one pixel on the screen.) The keywords **double** and **triple** produce lines of double and treble thickness respectively.

The **usage=** parameter

The **usage=** parameter accepts four keywords: **solid**, **halftone** and **tenthtone**. The value **solid** is the default and produces a continuous line. **halftone** creates a dashed line, and the value **tenthtone** produces a dotted line.

These three parameters can also be contained in the **location** directive of an **item** declaration. For instance, if the parameter **usage=tenthtone** is added to ex14.7 as follows:

Example 14.7

```
delete name=map;usage=location;source=dummy1;type=permanent
item name=map;usage=location;source=dummy1;type=permanent
type form=arcinfo
location first=house1;usage=tenthtone
2 2
16 2
16 5
2 5
end
exit name=map
```

and this and ex14.4 are run again, the result will show the object with a dotted line. However, if the **query** command in ex14.4 were changed to:

Example 14.8

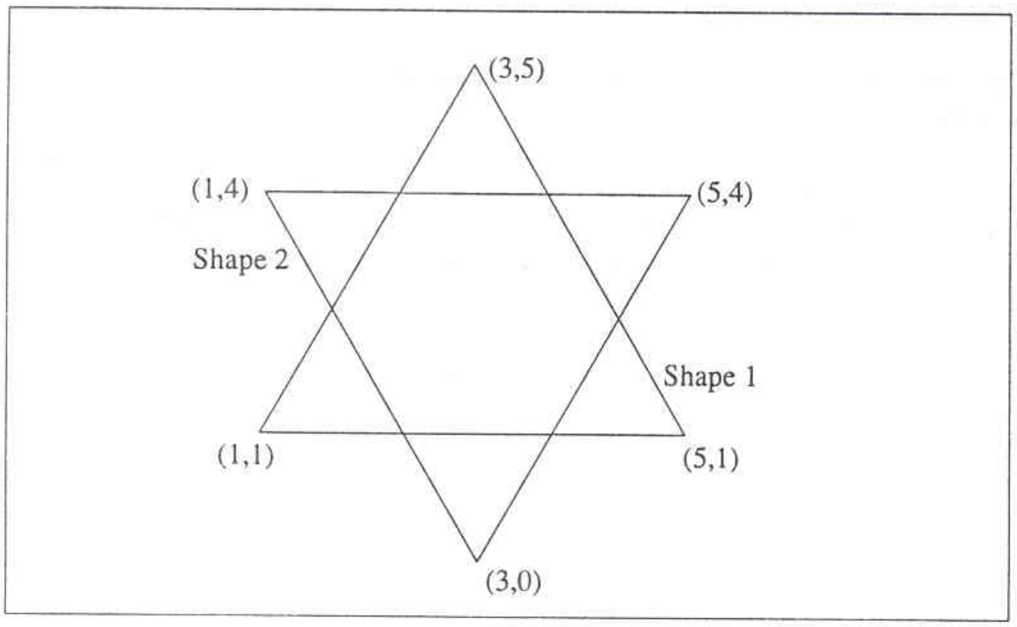
```
query name=dummy1;part=:name="Smith"
mapping part=:grid-reference;usage=halftone
stop
```

the object would be displayed with a dashed line. This is because κλειω only checks the **`location`** directive of a logical object if a characteristic of a line has not been specified in a **`mapping`** command.

Exercises

Exercise 14.1

Produce a database which contains 'Shape 1' and 'Shape 2' in two different groups, and display them on the screen.



Exercise 14.2

Change the database so that 'Shape 1' is displayed in red and 'Shape 2' is displayed in blue.

14.2 More complex mapping

14.2.1 Introduction to Winchester Census material

Look at the file colebro.dat. This contains information from one street in Winchester taken from the 1881 English Census. A small example is shown below.

```
house$/37 Colebrook Street/site=n83
schedule$139
head$mz/William/Ventham/51/General Laboror/Winchester/Hants
relp$fs/Emma/Ventham/daur/12/Scholar/Winchester/Hants
relp$fs/Thomas/Ventham/son/6/Scholar/Winchester/Hants
```

You should be familiar with this database as it was used in the exercises on codebooks. Look at the datafile. There is an unusual element in the group house. This element is called site and contains data of type `location`.

This can be seen in the file colebro.mod. Here is one line of it:

```
element name=site;type=location;location=houses
```

This shows that the element site is of `location` data type and refers to a logical object called houses.

The next file to consider is colebro.loc. This file contains the logical object houses. The fact that this file is over 800 lines long and refers to just over 100 objects suggests that it would be wiser to follow the instructions above about keeping this type of logical object in a separate file in case it needs correcting.

This logical object is called houses, relates to `location` type data and has the database colebro as a source.

The coordinates described in this command refer in the same way as do those above to a particular house. (In this case, however, the coordinates are given to six decimal places.)

```
item name=houses;usage=location;source=colebro;type=permanent
type form=arcinfo;
location first=n1
26.326000      11.083000
26.878000      11.052000
26.898001      10.785000
26.337000      10.789000
END
location first=n2
26.337000      10.789000
26.900000      10.776000
26.945000      10.140000
26.365999      10.101000
END
...
...
exit name=houses
```

The coordinates in this example were produced using a software application called ARC/INFO which is used for producing, controlling, analysing and presenting geographical information. Most data processed by geographical information systems like ARC/INFO are input using a digitizer. There are three components to a digitizer, a computer, a digitizing table and a pointer (rather like a mouse). Coordinates are collected by placing a basemap on the digitizing table and clicking on the pointer. Each click of the pointer represents a coordinate. Once all the objects have been digitized the GIS (Geographical Information System) allows the user to produce output in a format very similar to that shown above. Some slight editing was needed to produce this format.

As we have seen above, it is possible to input coordinates 'by hand', but for a map of any complexity a digitizer is recommended. Notice that in this example the coordinates are given to six decimal places. This is the default format of ARC/INFO. κλειω will accept these coordinates, but, as we have shown, you do not require them.

If you have not already created this database, do so now, first compiling the .mod file, then the .loc file and finally the .dat file. Please note that it is essential to compile the files in this order.

14.2.2 The `type` directive and the `form=` parameter

In the colebro database the **`type`** directive takes a **`form=`** parameter which has the value **`arcinfo`**. This particular directive indicates to κλειω the format of the coordinates that follow it. The parameter value **`arcinfo`** relates to the software used to create the coordinates. Currently κλειω understands data of this type in two formats; as above (called **`arcinfo`**) and as below, called **`digipad`**. (See the Reference Manual, Section 7.3.1.13.1.4.1 ff).

```
type form=digipad
location first=n1
3 2632, 1108 2687, 1105 2689, 1078
1 2633, 1078
```

The coordinates in this example have been made up to look as though they have been output from a digitizer using the software application digipad.

14.2.3 The structure of the colebro database

The task contained in the file ex14.9 displays the occupations of the heads of the household within this database:

Example 14.9

```
query name=colebro;part=head
write part=:occupation
stop
```

There are rather a lot of bricklayers in this sample, and perhaps we might consider it useful to see where they all lived — whether they all lived close together or were regularly distributed over the whole of the street. Look at the results of the following task:

Example 14.10

```
query name=colebro;part=head:occupation="brick" or "Brick"
write part=back[2]:site
stop
```

This task will give the ‘site’ of each person whose occupation contained the character string "brick" or "Brick", but the manner in which the information was retrieved is not very useful.

Also note that the function **`back[]`** is needed as the site is on a different level in the hierarchy to the group head.

Compare the following task with ex14.10 before executing it:

Example 14.11

```
query name=colebro;part=head:occupation="brick" or "Brick"
mapping part=back[2]:site
stop
```

This task will display the element site (i.e. a graphical representation of the house) where the head of household has an occupation which has the character string "brick" or "Brick" in it. This will include all bricklayers' labourers.

The task below slightly alters the information to be shown on the screen. The only change in the display is that the outline of the houses is produced with a triple line. The other added parameters need not be included, as they only tell κλειω to do what it would by default.

Run this task now:

Example 14.12

```
query name=colebro;part=head:occupation="brick" or "Brick"
mapping part=back[2]:site;
        colour=contrast;usage=solid;lines=triple
stop
```

What is displayed? Only the houses of the bricklayers and the bricklayers' labourers. The line at the bottom left of the map is part of the castle wall. (We will explain this shortly.) All the reference labels relating to those houses are also displayed.

14.2.4 The `sign=` and `total=` parameters

The following example introduces the **`sign=`** parameter:

Example 14.13

```
query name=colebro;part=head:occupation="brick" or "Brick"
mapping part=back[2]:site;
        colour=contrast;usage=solid;lines=triple;sign=no
mapping total=yes;sign=no;colour=red
stop
```

In this task κλειω is again asked to produce a map displaying the houses where all the bricklayers and their labourers live. κλειω is asked to produce them in continuous lines of treble thickness, but without their reference labels.

The parameter **`sign=`**, followed by the parameter value **`no`**, tells κλειω not to display the reference labels of the objects that it is displaying. In effect **`sign=no`** suppresses the labelling of an object. This is of considerable value when displaying results on the monitor of a PC, as these labels are not always shown in the most convenient place. However, when you are using a plotter or a printer, these labels are displayed in a much more useful position and could be worth retaining.

The second **`mapping`** command tells κλειω to display all the other topographical objects in the database on the screen without their labels and in red.

A full explanation of how the two **`mapping`** commands in this task interact will be described in detail in Section 14.2.6.

14.2.5 Reference objects

In the map just produced there is still one label, and the castle wall is still produced in white even though the task specified that no labelling should be specified and all the remaining items in the database should be displayed in red. This is because this particular object is defined as a reference object within the logical object houses. The following lines are taken from the logical object map.

```
location first=n111;connected=no;always=yes
 4.806000     13.729000
 5.399000     13.634000
 4.570000     11.163000
 5.413000     10.714000
 7.512000      7.945000
 8.707000      6.778000
10.787000      5.509000
12.569000      4.439000
14.910000      3.213000
end
```

The following two sections describe the two new parameters.

The **`connected=`** parameter

The parameter **`connected=`** followed by the keyword **`no`** tells κλειω not to join the last coordinate to the first coordinate. This allows the user to define a line, rather than a polygon. In the case of this map it defines the route of the castle wall.

The **`always=`** parameter

The other new specification, **`always=yes`**, tells κλειω always to display an object defined in the logical object as a reference object, i.e. defined in the logical object with the specification **`always=yes`** in a **`location`** declaration as in the above example.

The following task shows that it is possible to modify the way in which κλειω interprets this. As shown earlier, κλειω first checks the **`mapping`** command to see if any rules apply to an object, and then looks at the logical object containing such data to see if any rules are shown there. In the following task there is an **`always=`** parameter, so κλειω produces the results in accordance with this task rather than the instructions in the logical object.

Example 14.14

```
query name=colebro;part=head:occupation="brick" or "Brick"
mapping part=back[2]:site;
        colour=contrast;usage=solid;lines=triple;sign=no
mapping total=yes;sign=no;colour=red
mapping always=yes;usage=halftone;sign=no;colour=green
stop
```

The **`always=`** parameter can also have the parameter value **`no`** if reference objects were not to be displayed.

In this case the boundary of the castle wall is displayed with a green dashed line without its label.

Notice that this parameter can be used both in the definition of a logical object and within a **`query`** command.

A full translation of that task would be:

> I am interested in a database called colebro, and I am particularly interested in those people whose occupation contains the character string "brick". Display the outlines of the houses of those people, in the default colour, in solid lines of treble thickness and without their labels. Display all the other houses in the database as well, omitting their labels and showing them in red. Also display any reference objects with dashed lines and in green, also without their reference labels.

14.2.6 How `mapping` commands cooperate

κλειω always uses information from all the **`mapping`** commands in one task to produce a single map. How the individual parameters are interpreted depends to some extent on the other commands specified within the task. In the example above, the houses of bricklayers are displayed with the attributes described by the first **`mapping`** command while 'everything else' is displayed with the attributes specified in the **`mapping`** command followed by **`total=yes`**. In this case **`total=yes`** means "apply the following set of attributes to all the topographical objects in our **`location`** definition which have not already been displayed by another **`mapping`** command".

For this mechanism to work, the order in which the **`mapping`** commands occur with in a task is immaterial. The order of precedence is as follows:

- Each topographical object which is selected by criteria specified in the **`query`** command is included in the map by the **`part=`** parameter of a **`mapping`** command. These objects are displayed with the attributes specified in the same command.

- All topographical objects which have not been displayed are scanned for those which have a **`location`** declaration containing the specification **`always=yes`**. If you have specified a **`mapping`** command which has the same parameter set, the attributes specified there will be used to display those objects.

- Finally, if you have specified a **`mapping`** command which contains the parameter **`total=yes`**, every topographical object not selected in any of the steps already mentioned will be displayed with the attributes specified by this **`mapping`** command.

For advanced applications there is the possibility to combine a set of topographical objects in a group, which can be addressed by a common name. Such a group can be addressed by the **`name=`** parameter of the **`mapping`** command. This mechanism will not be discussed further in this volume. For the sake of completeness, however, any topographical object which is not included because of a **`part=`** specification but because it belongs to one of these groups specified by a **`name=`** parameter will be displayed with the attributes specified in a **`mapping`** command with the **`name=`** parameter. This option would fall second in the order of precedence mentioned above.

Exercises

The following exercises will take some time. They have been designed to exercise a number of techniques, which have been demonstrated both in this chapter and in earlier ones.

Exercise 14.3

Using the database colebro, create a codebook which classifies the occupations of the heads of household into three categories (manual labourer, artisan and others). (It is not vitally important to be accurate in deciding which of the occupations refer to which of the categories for this exercise, But it *is* important to have three categories of occupations.)

Exercise 14.4

Using this codebook, write a task which displays the houses of the artisans, using the codebook, in green with double strokes. Display the remaining houses in red and the castle wall in blue.

Exercise 14.5

Write a task which leaves the castle wall white, displays the houses of the artisans in green, the houses of the manual labourers in red and the other houses in blue. HINT: use the **`query`** ... **`confirm`** ... **`negate`** mechanism. This needs to be expressed as follows; if the occupation is artisan, paint green, if occupation is labourer, paint red; else if occupation is other, paint blue.

14.3 Printing a map using a PostScript printer

The following task demonstrates how a map is output to a PostScript printer rather than to the screen.

Example 14.15

```
query name=colebro;part=head:occupation="brick" or "Brick"
mapping part=back[2]:site;colour=contrast;
  usage=solid;lines=triple;sign=no;target="plot.15";
  write="Solution for mapping example 15";overwrite=yes
mapping total=yes;sign=no;colour=red
mapping always=yes;usage=halftone;sign=no;colour=green
location usage=Postscript
stop
```

There are four additional parameters, but otherwise it is relatively similar to ex14.14.

These differences can be found in the first **`mapping`** command and after a new command **`location`**.

14.3.1 The `target=` parameter

When κλειω produces a map to be output on a PostScript printer, it needs to convert the data into a special PostScript format. This must be placed in a particular file before it can be sent to the printer. The **`target=`** parameter in a **`mapping`** command takes the value of a user-defined file where this information should be stored. In the case of the task ex14.15 the name of this file is plot.15. If this parameter is absent and there is also a **`location`** command (see below), κλειω will send the information to a file called kleio.plt if you are using a DOS system. (If you are using another operating system you should refer to the Reference Manual, Section 8.3.7.2.4.1).

14.3.2 The `write=` parameter

The **`write=`** parameter expects a parameter value consisting of a character string. The contents of this string are displayed below the map, centred on the page.

14.3.3 The **`overwrite=`** parameter

This parameter is used in the same way as it usually is within κλειω, expecting the value **`yes`** or **`no`**. It relates, in this case, to the file plot.15.

14.3.4 The **`location`** command

In the task shown above there is also a new command, **`location`**. This command tells κλειω what format the result of the task is to be displayed in. If this command is absent, κλειω will direct the result to the screen. This can be explicitly requested with **`location usage=doscreen`**. If this command is followed by a **`usage=postscript`** parameter, κλειω will produce the result in PostScript format and send it to the file specified in the **`target=`** parameter. If you are using a colour PostScript printer you can use the parameter **`usage=colourpostscript`**. If you are not using a DOS based system there are different defaults. Refer to the Reference Manual, Section 7.3.4.6.2.

Once the task ex14.15 has been run, the results are stored in a file called plot.15. This file can then be copied to a PostScript printer. This will not work if any other sort of printer is being used.

If you are using a network, some special command may have been implemented to access a multi-purpose laser printer as a PostScript device. If you are unsure about how to access a particular printer you should consult an advisor, otherwise you may end up sending the printer hundreds of pages of PostScript code rather than a map. In simple set-ups the file plot.15 can be sent to a printer using the following command at the DOS prompt:

```
copy plot.15 prn:
```

14.4 Shading objects

The following task demonstrates how to control the shading of selected objects:

Example 14.17

```
query name=colebro;part=head:occupation="brick" or "Brick"
mapping part=back[2]:site;sign=no;
    colour=contrast;colour=contrast;
    usage=solid;usage=solid;
    lines=triple;
    write="Solution for mapping example 17";overwrite=yes
mapping total=yes;sign=no;
        colour=red;colour=red;
        usage=solid;usage=halftone
mapping always=yes;sign=no;
        colour=green
stop
```

The use in this task of two `colour=` parameters and two `usage=` parameters is the main difference between this task and the previous one. κλειω interprets the first of each of these parameters to relate to the outline of the object being displayed on the screen and the second to the 'contents' of the object. These second parameters take the same keywords as the first, but these keywords do not have to have the same value as each other. (ex14.16 is the same as ex14.17, except that it sends the output to a PostScript file.)

14.5 Using variable names with the printer

The following task demonstrates how κλειω uses variable names with the printer. In this case all the labels defined for the houses and the reference objects will be displayed on the printed output. Print this map out to see how κλειω positions these labels on the map. These are not particularly beautiful but may help you to interpret the map.

Example 14.18

```
query name=colebro;part=head:occupation="brick" or "Brick"
mapping part=back[2]:site;
   colour=contrast;
   usage=solid;
   lines=triple;overwrite=yes;
   write="Solution for mapping example 18";
   target="plot.18"
mapping total=yes;
   colour=red;
   usage=solid;
   mapping always=yes;
   usage=solid;
   colour=green
location usage=postscript
stop
```

This final task shows one of the limitations of producing the labels on the printed output. Look carefully at object n62 on the map produced from task ex14.17, and then compare it with the map produced from this task. In this task the label has been overwritten by the shading of the 'contents' of the object.

Example 14.19

```
query name=colebro;part=head:occupation="brick" or "Brick"
mapping part=back[2]:site;
        colour=contrast;colour=contrast;
        usage=solid;usage=tenthtone;
        lines=triple;overwrite=yes;
        write="Solution for mapping example 19";
          target="plot.19"
mapping total=yes;
        colour=red;
        usage=solid;
mapping always=yes;
        usage=solid;
        colour=green
location usage=postscript
stop
```

14.6 Other features

The mapping features of κλειω have been under-used for some time, as only the recent accessibility of PostScript printers has made hardcopy output available to most users. These features are now under active development. If you would like to experiment, we would briefly like to point to three groups of parameters, one of which has not even made it into the Reference Manual yet.

14.6.1 Scale of maps

Normally κλειω will try to produce a map in such a scale as to use the plotting or display area as effectively as possible. You can influence this by specifying the parameters **`first=`** and **`second=`** in a **`mapping`** command. Both parameters accept a pair of numbers separated by a comma. For example:

```
mapping....;first=10,250;second=30,300
```

This tells κλειω to restrict the display area to a box extending from 10 units to 250 units of the area in the horizontal, and from 30 to 300 units in the vertical. The units that κλειω uses depends on the granularity of your output device (that is, the size of the smallest distinguishable dot that can be drawn). Rather than going into a lengthy technical explanation, we would suggest that you experiment with these parameters.

14.6.2 Distribution maps

All the maps we have described so far are topographical; that is, they show the layout of a restricted area, where individual objects cover a wide area. Recently support for another type of map has been implemented, where individual objects are displayed by symbols rather than by lines surrounding them – a good method to display phenomena like the geographical place of origin. This method of display will be chosen automatically when topographical objects consist of a single coordinate (which by default will be represented by a circle centred around that point). To force κλειω to represent a topographical object by a symbol, another attribute can be specified with the **`symbol=`** parameter which accepts the keywords **`circle`**, **`square`** and **`triangle`** – which unsurprisingly produces those symbols for the topographical object to be displayed. Sections 8.3.7.2.2.6 ff. in the Reference Manual describe this mechanism in slightly more detail.

14.6.3 Directional parameters

Four parameters not described in the Reference Manual may be useful with distribution maps which relate to larger areas. Assume you have a database which contains information about the geographical origin of a group of individuals. Assume also that these people are only from places within the British Isles. However, when producing a map of all those individuals, you find that a number of points cluster around London – in fact the symbols are too dense to discriminate between them. It would be more convenient to see a map which only contains those people who came from south-eastern England. The display area of the map can be modified with these parameters:

```
mapping....;north=Buckingham;east=Southend;south=Brighton;
 west=Windsor
```

This means that places further north than Buckingham, further east than Southend, further south than Brighton and further west than Windsor will not be displayed. Obviously these names must be valid names in the **`location`** definition. For example:

```
location first=Buckingham
320, 475
end
```

Exercise 14.6

Using the data from the colebrook database, produce a map which shows the households for which the occupation of the wife is unknown. Make these houses solid. HINT: conditions in κλειω can have a syntax like this:

```
part=:element1=xxx and :element2=yyy
```

Summary

There are three ways to select objects for display:

- Specifying it in a **`part=`** parameter of a **`query`** command
- Specifying it with a **`total=`** parameter after a **`mapping`** command
- Specifying it in an **`always=`** parameter after a **`mapping`** command.

Further reading

Kenneth E. Foote, 'Mapping the Past. A Survey of Microcomputer Cartography', *Historical Methods*, 25:3 (1992), pp.121–131.

A. King and J. R. Moeschl, 'Mapping 19th Century Transport: The Application of Computer Cartography to Historical Data', *Histoire et Informatique V*, ed. J. Smets, (Montpellier, 1992), pp. 325–31.

Paul M. Mather, *Computer Applications in Geography* (Wiley, Chichester, 1991)

H. Southall and E. Oliver, 'Drawing Maps with a Computer... and Without', *History and Computing*, 2:2 (1990), pp. 146–54.

Chapter 15

Database design; some advanced features

15.1 Introduction

This chapter introduces some further information on the production of databases and some tips and tricks for dealing with κλειω databases.

15.1.1 Using the `:query[]` function

Probably the most common error (apart from missing out semi-colons) encountered using κλειω involves not using the built-in function `:query[]`. This problem relates to the use of more than one entry within a particular information element.

The following task:

```
query name=colebro;part=:occupation="Bricklayers Laborer" equal
index part=:occupation
stop
```

will produce the following result:

```
Bricklayers Laborer 1
Bricklayers Laborer 1
Bricklayers Laborer 1
Bricklayers Laborer 1
Bricklayers Laborer 1
Chelsea Pensioner 1
```

The task asked κλειω to produce an index of all the occupations of all those people whose occupation was exactly the same as the character string "Bricklayers Laborer" and κλειω has responded by displaying a list of five Bricklayers Laborers and one Chelsea Pensioner. This is because κλειω has been asked to go to the database colebro, and that part of the database where the element occupation contains the string "Bricklayers Laborer". And then display all the occupations found where that string occurs. The following is a line from the data file:

```
head$mz/Henry/Taplin/50/Chelsea Pensioner;Bricklayers Laborer
   /London/Middlesex
```

κλειω would select this group while processing the first line of the task, but when asked to produce an index of all the occupations in all of the groups selected it would also select Chelsea Pensioner as an occupation. This problem can be solved by using the element function **:query[]**. This function is interpreted by κλειω as the last entry reached in a **query** command. In this case the group contains only those occupations that are exactly equal to the character string "Bricklayers Laborer".

```
query name=colebro;part=:occupation="Bricklayers Laborer" equal
index part=:query[]
stop
```

A similar problem may occur on the level of groups. Remember the probate database. There we had a structure in which the group p (which can occur only once) occurs at the same level as the group relp (which can occur with varying frequency). So a specific document might contain the structure:

```
          inventory
           |         |
p ——   |         |—— relp 1
                     |—— relp 2
                     |—— relp 3
```

Now assume that we want to create an index in which we want to have information about related people on one line with some information about those people. We might, for example, use a query like the following:

```
query name=probate;part=relp:occupation="widow"
```

If we use the **index** command:

```
index part=back[1]/p:surname;
  part=:fname;
  part=back[1]/relp:surname;
  part=:fname
```

we would be rather disappointed with the result. κλειω, as always interpreting everything at the most general level, would be directed to go back from the specific relp which we selected to the group containing it. When it is ordered to go back to relp (by the third **`part=`** parameter), the system would interpret this as "go to *all* relps which you can reach from here". So while our widow might be relp "2" in the diagram above, we would actually produce three index lines, one for each of the relps above.

The problem would be solved if instead we specified the following:

```
index part=back[1]/p:surname;
  part=:fname;
  part=query[]:surname;
  part=:fname
```

Now **`query[]`** would bring us back to exactly that group which we were in when the **`query`** command selected a group for further processing.

15.1.2 Using the comparison operator `null`

Probably the next most common problem in κλειω relates to the use of the comparison operator `null`.

Try running the following task:

```
query name=colebro;part=:birthto=null
index part=:surname;part=:fname;part=:birthco;part=:country
stop
```

κλειω will produce this:

```
System did not find any information items for further processing.
```

The following task will work, however:

```
options lines=0
query name=colebr;part=relp:birthto=null
index part=:surname;part=:fname;part=:birthco;part=:country
stop
```

and the result should be as follows:

```
Boyle Mary  Ireland 1
Harvey Ann  Cape of Good Hope 1
Warren Kate  Ireland 1
```

In the first task, κλειω is directed to all groups where there is an element birthto but where that element is missing. However, once κλειω is in the group(s) where the element birthco does not exist, it is unable to access any other information, so it says that there is no further information for processing. In the second task the group from which we want κλειω to extract data is explicitly mentioned, so it is able to produce the required result. κλειω must be able to select some information before processing it.

If we wanted to look at all those people in this list (both heads of household and related people) who were distinguished by not having an entry for their town of birth, we would have to formulate the task in a rather different way:

```
options lines=0
query name=colebro;part=father[:surname]:birthto=null
write part=:each[]
stop
```

This introduces a new built-in function, **`father[]`**, which always takes as an element specification as an argument. κλειω interprets this built-in function as all the groups which have that particular element. As all people in this database have a surname, it is possible to call up both of the groups which have this element with this single function.

15.1.3 A further note on catalogues

As we said in Chapter 8, creating a catalogue is easy. Run this:

```
query name=colebro;part=:surname
catalogue part=:surname;name=colecat
stop
```

This produces a catalogue of all the surnames from the database colebro.

```
query name=colebro;part=:surname="Holloway"
write part=:each[]
stop
```

This task produces all the information about all the people in the database colebro whose surname is Holloway. Try to time the task, i.e. see how long you have to wait between the display of the task on the screen and the start of the display of the result.

Compare that time with the time necessary for κλειω to start producing results from this task:

```
query name=colebro;part=catalogue[colecat,complete,"Holloway"]
write part=:each[]
stop
```

The following task produces an alphabetical list (by surname) of all the people in the database colebro:

```
query name=colebro;part=catalogue[colecat]
write part=:each[]
stop
```

whereas the usual task set up for this purpose would produce the names in the order that they appeared in the database:

```
query name=colebro;part=:surname
write part=:each[]
stop
```

If this catalogue also included the soundexed surnames, the result of the third task here would result in a list of all the people in soundex order.

```
item name=code;usage=soundex;type=permanent;source=colebro
conversion without="aeiouy"
part signs="bpfv"
part signs="cgjkqsxz"
part signs="dt"
part signs="l"
part signs="mn"
part signs="r"
exit name=code


query name=colebro;part=:surname
catalogue name=colecat;part=:surname;type=terms;soundex=code;
  overwrite=yes
stop


query name=colebro;part=catalogue[colecat,algorithm]
write part=:each[],soundex[:surname,code]
stop
```

15.1.4 Using catalogues with the **index** command

Using a catalogue with the **index** command? Considering what has been said above, which of the following two tasks would you think were quicker? Why? Try running them both.

```
query name=colebro;part=:surname
index part=:surname
stop

query name=colebro;part=catalogue[colecat]
index part=:surname
stop
```

Actually in this case there is almost no difference in the amount of time it takes to run these two tasks. But theoretically the first solution is quicker than the second. In the first task κλειω goes to each of the groups where there is a surname and extracts it, then when it has collected all the surnames, it processes them in accordance with the instructions on the following line. Thus it sorts them alphabetically and displays them on the screen. In the second case where the index is already in alphabetical order, κλειω still goes through the whole list extracting all the surnames, and then sorts them into alphabetical order, which takes just as long whatever their order.

15.2 Databases made up from more than one source

The Reference Manual (Section 5.8.1) is very clear in its description of how to have several types of document in the same database.

The main points to remember are that within the structure declaration of a database, each different 'document' must be explicitly described with a **start=** parameter in the **part** directive. This example uses data from East Indian Army registers, which are simply lists of all the soldiers in the British Army serving in the East Indian army. The volumes of registers contain many different lists of individuals which contain some information in common but refer to different groups of people. It would be perfectly possible for κλειω to understand different documents within the same database, so that information relating to different groups of people can be gathered from the source, or information relating to all the people can be output.

A possible solution to the problem is shown below:

```
database name=eir1858;first=doc1;overwrite=yes
part name=doc1;
        start=yes;part=doc2;position=title
part name=doc2;
        part=doc3;position=type
part name=doc3
        part=ann;position=area
part name=ann;
        position=id,surname,forename,middle,appointed,retired
part name=doc4;
        start=yes;part=doc5;position=title
part name=doc5;
        part=doc6;position=area
part name=doc6;
        part=doc7;position=station,arrived
part name=doc7;
        part=sol;position=station,arrived
part name=sol;

position=id,appointed,surname,forename,middle,honour,rank,
     regiment,army,remarks
exit name=eir1858


read name=eir1858
doc1$EIR & AL 1858
doc2$Civil service Annuitants
doc3$Bengal Establishment
ann$a1-228/Alexander/George//1824/1845
ann$a2-228/Alexander/W/S/1825/1852
ann$a3-228/Armstrong/H//1824/1850
ann$a4-228/Ainslie/Montague//1807/1836
doc4$EIR & AL 1858
doc5$Bengal
doc6$Third European Regiment
doc7$Agra/19.5.56
sol$s1-104/1819/Huyshe/G//C.B./24.11.1853/28.11/1854/On
furlough
sol$s2-104/1826/Swatman/W//12.1.1853/12.1.1856/Do#on furlough
```

There are two comments to make about this solution. First, the two entries for the element title in the two different documents (doc1 and doc4) are the same (EIR & AL 1858). This causes what is known in κλειω as disambiguation.

```
doc1 (1 = "EIR") : ann (4 = "ann-4")
     id           a4-228
     surname      Ainslie
     forename     Montague
     appointed    1807
     retired      1836
```

```
doc4 (2 = "EIRa") : sol (1 = "sol-1")
        id              s1-104
        surname         Huyshe
        forename        G
        appointed       1819
        honour          C.B.
        rank            24.11.1853
        regiment        28.11
        army            1854
        remarks         On furlough
```

Notice that the identifier for the second person here is EIRa. There are two solutions to this problem. First, this group could be moved down the hierarchy so that an artificial identifier could be used to construct the identifying prefix.

The other solution is simpler to implement but is likely to lead to subsequent problems.

```
database name=eir1858;first=doc1;overwrite=yes;more=yes
```

Adding the **`more=yes`** parameter to the **`database`** command means that absolutely addressable identifiers can appear more than once in a database. In that case an id may describe more than one document. κλειω will try and interpret all references to such an id by resolving it as a series of documents. However, you lose the possibility of addressing one of them explicitly.

The other comment to make with this particular example is that the names of the groups are rather confusing. If this database was made up of 30 to 40 different documents, the use of the names doc1, doc2 etc for groups could become unwieldy, leading to group names such as doc100, etc.

This problem may be easier to see, using the **`describe`** command in a different way to that which has been described before.

The following task asks κλειω to describe a 'logical object' called eir1858, which is related to the database eir1858, and also to describe the structure of this logical object:

```
describe name=eir1858;source=eir1858;write=structure;
  usage=source;type=permanent
stop
```

Part of the result of this task is displayed below:

```
Schema of data structure for documents named: doc4

* doc4
*
*  - title
*
* * doc5
* *
* *  - area
* *
* * * doc6
* * *
* * *  - station
* * *  - arrived
* * *
* * * * doc7
* * * *
* * * *  - station
* * * *  - arrived
* * * *
* * * * * sol
* * * * *
* * * * *  - id
* * * * *  - appointed
* * * * *  - surname
* * * * *  - forename
* * * * *  - middle
* * * * *  - honour
* * * * *  - rank
* * * * *  - regiment
* * * * *  - army
* * * * *  - remarks
```

(This result is similar to that achieved using the **write=** parameter with the **database** command.)

Another example which solves the problem of using more than one source to make a database is shown below. This data is from the same type of source but uses slightly different information. The example also demonstrates the use of the **cumulate=** parameter.

```
database name=indies;first=civser;overwrite=yes
part name=civser;start=yes;
       position=class,seasap,title,fname,surname,funct,station
part name=orders;start=yes;
       position=order,class,rank,title,fname,surname,estab,
     date
exit name=indies
```

```
read name=indies
civser$first class/1794//John/French/
civser$first class/1807//Robert/Lowther/commissioner of revenue
   and circuit/Allahabad division
civser$first class/1809//Abercromby/Dick/judge of sudder
   dewanny and nizamut adawlut/presidency
civser$first class/1810//Chas./Du Pre Russell/collector/Purneah
civser$first class/1812//Ralph John/Tayler/civil and sessions
   judge/Jaunpore
civser$first class/1817//William/Dampier/commissioner of
   revenue and circuit/Patna division
civser$first class/1817/Sir/Robt./Barlow, Bart./judge of the
   sudder dewanny and nizamut adawlut/presidency
civser$first class/1817//George Alexander/Bushby/resident/
   Hyderabad
civser$first class/1818//George Charles/Cheap/civil and
   sessions judge/Rajeshye
civser$first class/1818//William/Popham Palmer/civil
   auditor/Bengal
civser$first class/1819//Alfred William/Begbie/judge of the
   courts of sudder dewanny and sudder foujdarry adawlut/
   N.W. provinces
orders$order of the Bath/military knights Grand Cross/general/
   Sir/James L./Lushington/Madras/20.7.1836
orders$order of the Bath/military knights Grand Cross/Lieut.
   Gen./Sir/George/Pollock/Bengal/2.12.1842
orders$order of the Bath/military knights Grand Cross/Lieut.
   Gen./Sir/John H./Littler/Bengal/3.2.1848
orders$order of the Bath/military knights Grand Cross/General/
   Sir/James L./Caldwell/Madras/30.8.1848
orders$order of the Bath/civil knights Grand Cross/Sir/John/
   M'Neill#retired from the service/Bombay/15.4.1839
orders$order of the Bath/civil knights Grand Cross/The Right
   Hon. Sir/H./Pottinger, Bart./Bombay/2.10.1842
orders$order of the Bath/military knights commanders/General/
   Sir/Hopetoun S./Scott/Madras/27.9.1831
orders$order of the Bath/military knights commanders/General/
   Sir/David/Leighton/Bombay/10.3.1837
orders$order of the Bath/military knights commanders/General/
   Sir/James/Russell/Madras/10.3.1837
orders$order of the Bath/military knights commanders/General/
   Sir/Robert/Houston/Bengal/10.3.1837
orders$order of the Bath/military knights commanders/General/
   Sir/William/Richards/Bengal/20.7.1838
orders$order of the Bath/military knights commanders/General/
   Sir/John/Doveton/Madras/20.7.1838
orders$order of the Bath/military knights commanders/Maj.
   Gen./Sir/John/Cheape/Bengal/9.6.1849
orders$order of the Bath/military knights commanders/Maj.
   Gen./Sir/H.M./Wheeler/Bengal/17.8.1850
orders$order of the Bath/military knights commanders/Maj.
   Gen./Sir/S.W./Steel/Madras/9.12.1853
```

Consider the second half of the data file for this database. There is an extraordinary amount of repetition of information, all of which can be simplified by a single parameter. Consider the following datafile (the related structure declaration can be left unchanged):

```
read name=indies;cumulate="
civser$civ-1/first class/1794//John/French/
civser$civ-2/"/1807//Robert/Lowther/commissioner of revenue and
   circuit/Allahabad division
civser$civ-3/"/1809//Abercromby/Dick/judge of sudder dewanny
   and nizamut adawlut/presidency
civser$civ-4/"/1810//Chas./Du Pre Russell/collector/Purneah
civser$civ-5/"/1812//Ralph John/Tayler/civil and sessions
   judge/Jaunpore
civser$civ-6/"/1817//William/Dampier/commissioner of revenue
   and circuit/Patna division
civser$civ-7/"/1817/Sir/Robt./Barlow, Bart./judge of the sudder
   dewanny and nizamut adawlut/presidency
civser$civ-8/"/1817//George Alexander/Bushby/resident/Hyderabad
civser$civ-9/"/1818//George Charles/Cheap/civil and sessions
   judge/Rajeshye
civser$civ-10/"/1818//William/Popham Palmer/civil
   auditor/Bengal
civser$civ-11/"/1819//Alfred William/Begbie/judge of the courts
   of sudder dewanny and sudder foujdarry adawlut/N.W. provinces
orders$ord-1/order of the Bath/military knights Grand Cross/
   general/Sir/James L./Lushington/Madras/20.7.1836
orders$ord-2/"/military knights Grand Cross/Lieut.Gen./Sir/
   George/Pollock/Bengal/2.12.1842
orders$ord-3/"/"/Lieut. Gen./Sir/John H./Littler/Bengal/
   3.2.1848
orders$ord-4/"/"/General/Sir/James L./Caldwell/Madras/30.8.1848
orders$ord-5/"/civil knights Grand Cross/Sir/John/
   M'Neill#retired from the service/Bombay/15.4.1839
orders$ord-6/"/"/The Right Hon. Sir/H./Pottinger, Bart./Bombay/
   2.10.1842
orders$ord-7/"/military knights commanders/General/Sir/Hopetoun
   S./Scott/Madras/27.9.1831
orders$ord-8/"/"/General/Sir/David/Leighton/Bombay/10.3.1837
orders$ord-9/"/"/General/Sir/James/Russell/Madras/10.3.1837
orders$ord-10/"/"/General/Sir/Robert/Houston/Bengal/10.3.1837
orders$ord-11/"/"/General/Sir/William/Richards/Bengal/20.7.1838
orders$ord-12/"/"/General/Sir/John/Doveton/Madras/20.7.1838
orders$ord-13/"/"/Maj. Gen./Sir/John/Cheape/Bengal/9.6.1849
orders$ord-14/"/"/Maj. Gen./Sir/H.M./Wheeler/Bengal/17.8.1850
orders$ord-15/"/"/Maj. Gen./Sir/S.W./Steel/Madras/9.12.1853
```

15.2.1 The `cumulate=` parameter

The parameter **`cumulate=`** in a **`read`** command can be followed by any special character (defined by the user) to simulate a ditto mark. In this case the '"' symbol has been used. This character does not have to be used, but as the quotation mark is very unlikely to be valid as the *only* character in an element there should be no conflict. Look at the result of a simple task using this parameter. κλειω will produce the correct entry for any entry where this symbol has been used. However, there is one important limitation to using this parameter.

Consider the following data (adapted from the 1881 Winchester census):

```
read name=qqq;cumulate="
schedule$164
head$mz/William/Holloway/60/Upholsterer;Paper Hanger/
  Winchester/Hants
relp$fz/Laura/Holloway/wife/56//"/"
relp$mz/Alfred/Holloway/son/26/Carpenter/"/"
relp$fz/Ellen/Holloway/daughter#in law/23//"/"
relp$fs/Mary/Holloway/daughter/28/Washerwoman/"/"
```

Compare the results for Ellen and Mary Holloway:

```
schedule (1 = "164") : relp (3 = "rel-3")
        status          female, married
        fname           Ellen
        surname         Holloway
        age             23
        birthto         Winchester
        birthco         Hants
        relation        daughter
         comment        in law

schedule (1 = "164") : relp (4 = "rel-4")
        status          female, unmarried
        fname           Mary
        surname         Holloway
        age             28
        occupation      Washerwoman
        birthto         Winchester
        birthco         Hants
        relation        daughter
```

If the datafile is changed to look like this:

```
read name=qqq;cumulate="
schedule$164
head$mz/William/Holloway/60/Upholsterer;Paper Hanger/
  Winchester/Hants
```

```
relp$fz/Laura/Holloway/wife/56//"/"
relp$mz/Alfred/Holloway/son/26/Carpenter/"/"
relp$fz/Ellen/Holloway/daughter#in law/23//"/"
relp$fs/Mary/Holloway/"/28/Washerwoman/"/"
```

the result will look like this:

```
schedule (1 = "164") : relp (3 = "rel-3")
        status          female, married
        fname           Ellen
        surname         Holloway
        age             23
        birthto         Winchester
        birthco         Hants
        relation        daughter
         comment        in law

schedule (1 = "164") : relp (4 = "rel-4")
        status          female, unmarried
        fname           Mary
        surname         Holloway
        age             28
        occupation      Washerwoman
        birthto         Winchester
        birthco         Hants
        relation        daughter
         comment        in law
```

Notice that both Mary and Ellen are described as daughters-in-law.

15.2.2 The `substitution=` parameter

In fact all we wanted to repeat was that they were both daughters of the head of household. If the data file is again changed to this:

```
read name=qqq;cumulate=";substitution=&
schedule$164
head$mz/William/Holloway/60/Upholsterer;Paper Hanger/
  Winchester/Hants
relp$fz/Laura/Holloway/wife/56//"/"
relp$mz/Alfred/Holloway/son/26/Carpenter/"/"
relp$fz/Ellen/Holloway/daughter#in law/23//"/"
relp$fs/Mary/Holloway/&/28/Washerwoman/"/"
```

the result will appear as follows:

```
schedule (1 = "164") : relp (3 = "rel-3")
        status          female, married
        fname           Ellen
        surname         Holloway
        age             23
        birthto         Winchester
        birthco         Hants
        relation        daughter
         comment        in law

schedule (1 = "164") : relp (4 = "rel-4")
        status          female, unmarried
        fname           Mary
        surname         Holloway
        age             28
        occupation      Washerwoman
        birthto         Winchester
        birthco         Hants
        relation        daughter
```

The addition of the **substitution=** parameter followed by a **&** can be said to tell κλειω only to repeat the main information in the preceding element (if the symbol & replaces the " symbol) and not any information contained in either the comment or original aspects of that element.

The Reference Manual describes in detail what is happening here. (See Sections 6.2.3.2.1, 6.2.3.2.2, 6.2.3.2.2.1 and the following section for how to use this feature in more complex data structures.)

Returning to the **start=** parameter, one most important thing to remember is that a group can either be described as a document (i.e. in a **first=** parameter of a **database** command or with a **start=yes** parameter within a **part** command) or it can be defined in a **part=** parameter of a **part** command. It cannot, however, be defined in both, as this would make one document subordinate to another, which is an impossibility in κλειω.

A few further points referring to the **cumulate=** parameter used in a **read** command. Consider the following .mod file:

```
database name=aaa;first=m;overwrite=yes
part name=m;
       position=surname,fname;
       part=f
part name=f;
       position=maidenname,fname;
       part=c
part name=c;
       position=surname,fname
exit name=aaa
```

and the following data file:

```
read name=aaa;cumulate="
m$Smith/John
f$"/Anne
c$"/Eric
```

κλειω would bring up this error message:

```
***** Error: The specification you are trying to repeat is unknown.
m (1 = "Smith"),f (1 = "*  (Unknown)"), maidenname
```

This is because κλειω only repeats elements which have the same name. In this case the surnames of men and children are different to those of women. The error message suggests that there should already be a maiden name in the database for κλειω to ditto.

15.3 The `identification=` parameter

This parameter allows the user to change the identification of a group. Normally these are made by κλειω, depending on the name of the group.

In the colebro database, the following line contains the group identification for the group schedule. κλειω has made the group identifier out of the name of the group, by taking the first three letters of the word schedule and adding a hyphen and the group's ordinal number.

```
schedule (1 = "sch-1")
```

As all groups can be addressed by their identification, there are certain cases where it may be useful for different groups in a database to have identifiers that are not based on the name of the group but rather on the contents of the element within that group.

If, for example, the database made up of the East India lists contained 20,000 individuals with, say, 400 different documents, it might be useful to be able to look through parts of the database rather than documents. If one uses the **`identification=yes`** parameter within the structure definition of a database different groups or documents can be given an absolutely addressable identifier which means that κλειω accesses only those groups or documents specified in the task rather than sorting through the whole database. This is achieved in a similar way to a catalogue in κλειω, but in this case the user has only to specify how groups are to be identified.

In the following structure file the **`identification=yes`** parameter has been added to both of the documents in the database, and a new **`element`** command has been added which tells κλειω that the element to use to construct the new document identifier should be id. In this case the database has also been changed to give each individual in the database a unique identifier.

```
database name=eastind;first=civser;overwrite=yes
element name=id;identification=yes
part name=civser;
        start=yes;
        position=id,class,seasap,title,fname,surname,funct,
     station;
        identification=yes
part name=orders;
        start=yes;
        position=id,order,class,rank,title,fname,surname,
     estab,date;
        identification=yes
exit name=eastind

read name=eastind;cumulate="
civser$1/first class/1794//John/French/
civser$2/"/1807//Robert/Lowther/commissioner of revenue and
  circuit/Allahabad division
civser$3/"/1809//Abercromby/Dick/judge of sudder dewanny and
  nizamut adawlut/presidency
civser$4/"/1810//Chas./Du Pre Russell/collector/Purneah
civser$5/"/1812//Ralph John/Tayler/civil and sessions
  judge/Jaunpore
```

Once this has been compiled, it would be possible to question the database in such a way as to bring up those people who occurred in the second five of the database.

```
query name=eastind;part=sign[civserv,"6"-"10"]
write part=each[]
stop
```

However, as these people are absolutely addressable by their group identifier, it would be possible to get exactly the same results as in the above task by asking the following task:

```
query name=eastind;part=sign[orders,"6"-"10"]
write part=each[]
stop
```

The other benefit of using some system like this is that these absolutely addressable identifiers can be used within the **bridge** command, making the process of linking various entries in various databases much easier than in the way described in Chapters 12 and 13.

The notes above suggest that if it is necessary to use identifications that are not unique, a possible way to avoid disambiguation is to use the **identification=** parameter in a **database** command. For instance with a structure definition like this:

```
database name=y;first=y;identification=no
....
....
exit name=y
```

data like this would be acceptable:

```
y$abc/
y$abc/
```

Why are we interested in these identifications at all? As we mentioned above, the use of these identifications can assist in searching a very large database. In conjunction with the **sign[]** function it is possible to ask for all the instances of a document for a specific group which are within a given range of document identifications in the same order as they have been entered in the data file.

If an **index** command is very complicated, and a database very large, it may be worth testing the task on small portions of the database until all errors have been removed. This can be achieved using an **element** directive which included an **identification=yes** parameter.

```
query name=database;part=sign[x,"y1"-"y"]
index part=<complex command>
```

This access mechanism allows the user to go straight to part of a database to check whether data has been put in correctly – mainly useful for large scale databases! – and also 'checks' for redundant data, say when κλειω is unable to execute a command as there is no data on which the command can be performed.

These identifications can also be used for more complicated purposes. For example if you had a database with documents like this:

```
x$n325
```

this identification could be used in a command like this:

```
bridge first="n325"
```

However there is a problem here with the East India database. Part of the datafile could look like this:

```
y$bengal
division$military
regiment$12th
officer$David/Leighton
```

In order for this material to be used with a **bridge** command a symbolic address would have to made for this officer to relate back to the document level. See Chapter 13.

```
bridge first="bengal/division=1/regiment=1/officer=1";
```

But if a unique identification were to be added (on an individual level), it would be possible to replace the above line with something like this:

```
bridge first="m324"
```

In this case both bengal and m324 need to be absolutely addressable. Therefore κλειω needs to know that the identifier m324 in the database can be addressed in the same way as κλειω would understand "bengal".

There are two stages in this process:

First, κλειω needs to put m324 into the group identification. For example:

```
officer (1 = "off-1")

officer (1 = "m324")
```

Second, this identification needs to be able to be used in the **bridge** command.

Both of these problems can be solved with the use of the **identification=** parameter. These are used in the following places:

```
database name=x;first=y;
element name=id;identification=yes        - solves 1)
....
....
part name=officer;
  position=id,surname,fname;
  identification=yes                      - solves 2)
exit name=x
```

Obviously if one were using data such as that below, the structure definition above would need to be completed. We leave this to you.

```
read name
y$bengal
division$military
regiment$12th
officer$m234/John/Smith
```

15.4 Elements

15.4.1 Visibilities

Elements and groups within κλειω databases share two properties. These two properties are known as the *visibility* and the *view* of an element or a group. If you were even more unsure of your original data, and were not happy in giving an entry a status tag, it would be possible to denote the visibility of an element or a group. In the case of an element you can give a numerical value to denote its 'quality'. Using other κλειω tools, you can ask κλειω not to process data unless it has a visibility of, say, 95%. In other words the visibility of an element is a level of confidence. The same applies to groups.

Visibility clauses *must* follow an element name. In the case of the example below, we can see that the element name occupation is followed by a colon, followed by a question mark, followed by an equals sign and then the entry contained in the element.

```
relp$fs/Ann//Fox/servant/28/occupation:?=General Serv
  /birthto=Andover/birthco=Hants
```

There are two predefined symbols for this level of confidence. The question mark denotes 50% confidence and the exclamation mark denotes 95% confidence. Any other level can be defined by the user replacing the symbol by a numerical value which can be any number between 0 and 100. 65 would represent 65% confidence. If an element has no visibility clause, κλειω assumes that we have given it a level of confidence of 100%.

Remember that if you specify the name of an element in a data file, all the remaining elements must also have their names specified.

If the database was queried the database with a task like this:

```
query name=censsamp;part=relp:occupation
index part=:surname;
   part=:firstname;
   part=:occupation
stop
```

we would achieve a result like the following:

```
Bacon George Grocer's shopman RG11-5490
Cook Jane General Serv RG11-5490
Doyle Catherine Barmaid RG11-5490
Farmer Jane Corset Maker RG11-5490
Johnson Emma Corset Maker RG11-5490
Johnson Henry Organist RG11-5490
Morrison Elizabeth Annuitant RG11-5490
```

```
Newton Henrietta scholar RG11-5490
Newton William scholar RG11-5490
White Sophia General Serv RG11-5490
```

where there is no result for Ann Fox.

This is because κλειω only accepts data that we have ascribed at least 95% confidence unless we ask it to do otherwise, and we have allocated a visibility of 50% to Ann Fox's occupation.

There is a command that allows us to define the level of confidence that we wish κλειω to use, when specifying a task (i.e. this does not have to be built in during the design stage). If the task above were changed to this:

```
options minimum=0.45
query name=censsamp;part=relp:occupation
index part=:surname;
   part=:firstname;
   part=:occupation
stop
```

The result would be as follows:

```
Bacon George Grocer's shopman RG11-5490
Cook Jane General Serv RG11-5490
Doyle Catherine Barmaid RG11-5490
Farmer Jane Corset Maker RG11-5490
Fox Ann General Serv RG11-5490
Johnson Emma Corset Maker RG11-5490
Johnson Henry Organist RG11-5490
Morrison Elizabeth Annuitant RG11-5490
Newton Henrietta scholar RG11-5490
Newton William scholar RG11-5490
White Sophia General Serv RG11-5490
```

The **`options`** command added before a **`query`** command and followed by the parameter **`minimum=`**, with a parameter value of a number between 0 and 1, tells κλειω the lowest level of visibility to accept when processing a task.

There is a similar parameter, **`maximum=`** which denotes the highest level of confidence that we wish κλειω to accept. By default these two parameters are set at 0.95 (minimum) and 1 (maximum) (i.e. levels of 95% and 100%).

Groups can be given a level of confidence, in much the same way as elements. However, if you assign a level of confidence to a group it is also given to the elements that are contained in it. This can often produce some surprising results given that the visibility of an element is equal to its visibility multiplied by the visibility of the group containing it.

Groups are assigned a visibility in a similar way to elements. For example:

```
head:65$mz/Michael//Morrison/31/Licensed Victualler
   /Islington/Middlesex
relp$fz/Mary/J/Morrison/wife/29//Candover/Salop
relp$f/Ellen//Morrison/daughter/7m//Shrewsbury/Salop
relp$fl/Elizabeth//Morrison/mother/58/Annuitant/Shrewsbury
   /Salop
relp$fs/Ann//Fox/servant/28/General Serv/Andover/Hants
relp$fs/Catherine//Doyle/servant/24/Barmaid/country=Ireland
```

Here the group name head is followed by a colon and the number 65, indicating that the group head here has a visibility of 0.65.

If the following task is run:

```
query name=censsamp;part=person
write part=:surname
stop
```

the following result will be produced.

```
reference (1 = "RG11-5490") : surname    Newton
reference (1 = "RG11-5490") : surname    Newton
reference (1 = "RG11-5490") : surname    Newton
reference (1 = "RG11-5490") : surname    Newton
reference (1 = "RG11-5490") : surname    Bacon
reference (1 = "RG11-5490") : surname    Cook
reference (1 = "RG11-5490") : surname    Phillips
reference (1 = "RG11-5490") : surname    Phillips
reference (1 = "RG11-5490") : surname    White
reference (1 = "RG11-5490") : surname    Frampton
reference (1 = "RG11-5490") : surname    Frampton
reference (1 = "RG11-5490") : surname    Johnson
reference (1 = "RG11-5490") : surname    Johnson
reference (1 = "RG11-5490") : surname    Johnson
reference (1 = "RG11-5490") : surname    Farmer
reference (1 = "RG11-5490") : surname    Campbell
```

Notice that all of the people related that to that head of household have also been given a visibility of 0.65. In fact κλειω understands that if a group has a certain level of visibility all those groups directly subordinate to that particular group also receive that level of visibility. This is because the visibility of a group is equal to the visibility assigned to it (which defaults to one) multiplied by the visibility of the group that contains it. Defining the visibility of a group or an element is a very powerful tool but we suggest that you think very carefully before using it, and remember what might happen if you do it.

For an initiate of statistics this way of multiplying numerical values between 0 and 1 should look familiar: it is the concept of *conditional probability*. That is, the 'visibility' is interpreted by κλειω semantically as the probability that an element actually describes what we as historians assume it to be.

For the non-statistical historian a slightly more detailed explanation may be appropriate.

- If you say that you are not sure that a person has actually been a witness in a court session, κλειω assumes that all the information is doubtful. So if you assign a visibility of 50% to the group witness, the surname of that person will only be selected in a task if you are interested in data with a lower visibility — even in queries where you don't care whether the role of a person has been understood. Therefore you would have to set the threshold of acceptable visibility to 0.50 to include surnames of those people whose role is unclear.

- However, when you are not sure of an attribute of a person, independently of whether their role is clear, and you specify a visibility for that element, the visibility of that attribute will be higher when you are sure that their role is clear than when it isn't. Therefore of you were to assign a visibility of 50% to an element as well as a visibility of 50% to a group the visibility of that element would be understood to be 0.25. (0.5 multiplied by 0.5).

15.4.2 Views

Views are another particularly powerful tool that κλειω provides for the user. They allow the user to use a database in more than one state. In effect this means that you can tell κλειω only to use certain information found in a database when it is asked for. However all data occurs in all alternative views unless otherwise specified. For example, if the following line from the censsamp data file were changed from this:

```
relp$fs/Catherine//Doyle/servant/24/Barmaid/country=Ireland
```

to this:

```
relp$fs/Catherine//Doyle/servant/24/Barmaid/1=country=Ireland
```

we have told κλειω that we only want the country of origin to be displayed when we are viewing the database in alternative view number 1. All of the other information will be displayed in the usual manner in the original view of the database, while the whole of that group will be displayed in alternative view number 1.

For example, the following task aims to extract the countries of origin of all the related people in the database.

```
query name=censsamp;part=relp
write part=:country
stop
```

The following result would be produced:

```
reference (1 = "RG11-5490") : country      Scotland
```

There is another parameter which we can use in conjunction with the **options** command while allows us to specify that we would like to use the first alternative view. This task:

```
options also=1
query name=censsamp;part=relp
write part=:country
stop
```

produces the following result:

```
reference (1 = "RG11-5490") : country      Ireland
reference (1 = "RG11-5490") : country      Scotland
```

On most machines 16 views, numbered from 1 to 16, are available. On some very powerful workstations 32 views are accessible. Again, views need to be used with care, especially as κλειω does not allow you to use single views. The **also=** parameter of the **options** command actually specifies up to which level you want to be able to use. Thus if you allocated an element to a low level of alternative view, that data would always be present if you allocated a high level to the **also=** parameter in the **options** command

Groups can also be assigned to a single view, in a similar fashion to elements. (See Reference Manual, Section 4.3.4.1.)

Groups and elements share a further property, namely that the same group or element can exist in different views with different names. This is getting rather complex, and if you are not to sure about the concept of views we would suggest that you stop here and move on to the next section.

Elements can be assigned to several views by prefixing it with several names. Each of these names must be separated from each other by an equals sign and the last name must be connected to the element by an equals sign.

In the case of the censsamp database, the following example would not be historically accurate, but it should demonstrate this point easily. If we were to edit the data file so that all groups containing unmarried women were given a second name for their element surname, which denoted their maiden name, as follows:

```
relp$f/Ellen//surname=maidenname=Morrison/relation=daughter
   /age=7m//birthto=Shrewsbury/birthco=Salop
relp$fl/Elizabeth//Morrison/mother/58/Annuitant/Shrewsbury
   /Salop
relp$fs/Ann//surname=maidenname=Fox/relation=servant/age=28
   /occupation=General Serv/birthto=Andover/birthco=Hants
relp$fs/Catherine//surname=maidenname=Doyle/relation=servant
   /age=24/occupation=Barmaid/country=Ireland
```

we would be able to query the database on the original view with a query like this:

```
query name=censsamp;part=relp:status="f"
write part=:surname
stop
```

and receive a result like this:

```
reference (1 = "RG11-5490") : surname      Morrison
reference (1 = "RG11-5490") : surname      Morrison
reference (1 = "RG11-5490") : surname      Morrison
reference (1 = "RG11-5490") : surname      Fox
reference (1 = "RG11-5490") : surname      Doyle
reference (1 = "RG11-5490") : surname      Newton
reference (1 = "RG11-5490") : surname      Newton
reference (1 = "RG11-5490") : surname      Cook
reference (1 = "RG11-5490") : surname      Phillips
reference (1 = "RG11-5490") : surname      White
reference (1 = "RG11-5490") : surname      Frampton
reference (1 = "RG11-5490") : surname      Johnson
reference (1 = "RG11-5490") : surname      Farmer
```

while we could also query the first alternative view of the database like this:

```
options also=1
query name=censsamp;part=relp:status="f"
write part=:maidenname
stop
```

and obtain a result like this:

```
reference (1 = "RG11-5490") : maidenname   Morrison
reference (1 = "RG11-5490") : maidenname   Fox
reference (1 = "RG11-5490") : maidenname   Doyle
reference (1 = "RG11-5490") : maidenname   Newton
reference (1 = "RG11-5490") : maidenname   Cook
reference (1 = "RG11-5490") : maidenname   White
reference (1 = "RG11-5490") : maidenname   Johnson
reference (1 = "RG11-5490") : maidenname   Farmer
```

This result occurs because the element name maidenname has been allocated only to exist in the first alternative view of this database.

This feature works with groups as well as with elements. (See Reference Manual, Section 4.3.4.2.)

Chapter 16

Data entry with menus out-of-date

In this chapter we will be describing a different method of building a database, using the menu system. κλειω allows the user to enter data into a database using a form in the menu system. This is only possible for databases which have already been 'created' in the sense that a .mod file declaring its structure has previously been written and run. Nonetheless the feature is useful because it makes it easier to enter certain kinds of data, in particular data that is heavily structured, or repetitive, or both.

In this chapter we will be using the same source material as we used in Chapter 4 when we created a database for the first time. In order to prevent odd things from happening, create a new directory (we would suggest you call it tester) and copy the file tester.mod (on the tutorial disc) to that directory and compile it. At the DOS prompt, type **`kleio`** to enter the menu system. A menu like this should appear:

```
Kleio Version 5.1.1

  Systematic processing of a database
  Interactive processing of a database
  Creating a new database
```

With the ↑ and ↓ keys, move to creating a new database. Activate this by pressing F1. There should now be a menu that looks like this:

```
Creating a new database

  Preparing raw data: filtered input
```

Press F1 again. You should get a menu which looks rather like this:

```
Preparing the data

 Database      :
 Output file   :
 Ordinal no.   :1
 Extension     :dat
 Replace       :Save
 Characters    :No
 Back-up       :25

Please enter the name of the database
```

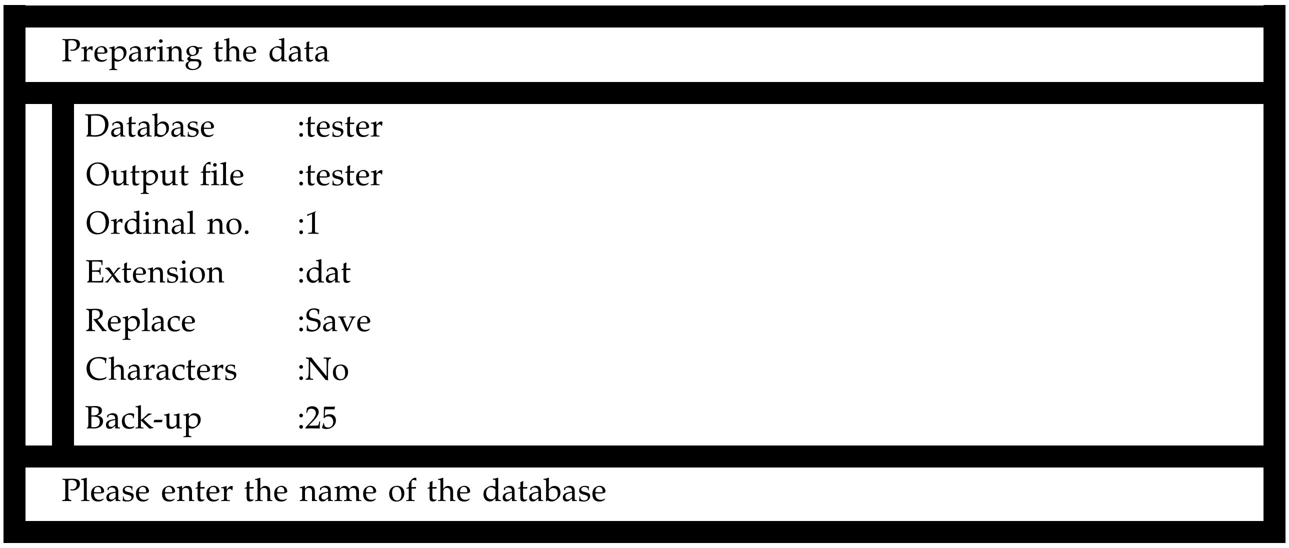
The first line of this menu should be highlighted. In this field you should type the name of the database that you wish to use. κλειω expects the name of a database that has already been created. In this case the **`database`** directive in the structure declaration has a **`name=`** parameter with the value tester. This should be the name that you type here. By default κλειω expects that you want to give the file to which you will be adding data the same name as the name of the database, i.e. tester. If you type tester in the **`Database`** field, κλειω will automatically give that name to the output file (.dat). You can change this.

Run down the other fields using the up and down arrows. You will get some idea of what the other fields in this menu represent. The **`Ordinal number`** refers to the name of the data file. By default, κλειω will make a datafile with the name of the database followed by the number 1, with the suffix .dat. For example, the name of the data file that we are making will be tester1.dat.

If you move down to the **`Replace`** field and toggle with the left and right arrow keys, you will see that κλειω gives three options for this file. It can be saved, overwritten or extended (i.e. your most recent additions will be added to the end of the file). If, however, you tell κλειω to save a file which already exists, κλειω will increment the ordinal number by one until it finds a number that has not already been used.

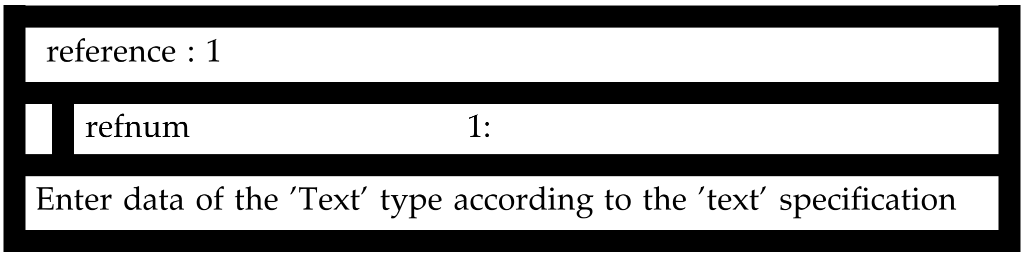
The **`Characters`** field asks you whether or not data signal characters entered in the filters should act as 'hotkeys', as the function they perform are replaced by the menu-system. (More prosaically: if this field is set to no and you enter a slash ("/"), that slash will be ignored. If **`Characters`** is set to yes the slash will take you to the next element in the filter on the screen.) The **`Back-up`** field asks after how many filters the data should be saved. The default here is 25, which is reasonable if you are using material of this kind, but if you were inputting larger strings of full text you may want to reduce this number. Each filter refers to a group. If you had been using material like this and a screen editor to enter the data you might have saved your file every 30 or 40 lines. Setting this to 25 groups would mean that you would be saving your data at about the same frequency.

Once you have changed this menu to look like this:



```
Preparing the data

Database        :tester
Output file     :tester
Ordinal no.     :1
Extension       :dat
Replace         :Save
Characters      :No
Back-up         :25

Please enter the name of the database
```

you should press F1 to activate this menu. You will then see a menu like this:



```
reference : 1

refnum                    1:

Enter data of the 'Text' type according to the 'text' specification
```

This is the highest level of the hierarchy of this database. If you cannot remember the structure of this database, refer either to Chapter 4 or to the .mod file in your newly created directory. You should also refer to the figure in Chapter 4 of the original data.

In this field you are prompted to add the elements within the group reference. You should type RG11-5490 here. Then press F2. This selects a dependent group. If you pressed F1 you would have got another reference menu. Pressing F2 (don't do it a second time) will produce a menu of the dependent group. In this case you should see a menu like this:

```
address : 1

parish          1:
town            1:
ward            1:
parlb           1:
sandis          1:
ecclp           1:

Enter data of the 'Text' type according to the 'text' specification
```

These five fields can easily be filled by typing St Mary, Shrewsbury, Welsh, Shrewsbury, Shrewsbury and St Michael.

Once you have done this, press F2 to move you down to the dependent group.

```
house : 1

status          1:
address         1:
name            1:

Enter data of the 'Category' type according to the 'abbreviation' specification
```

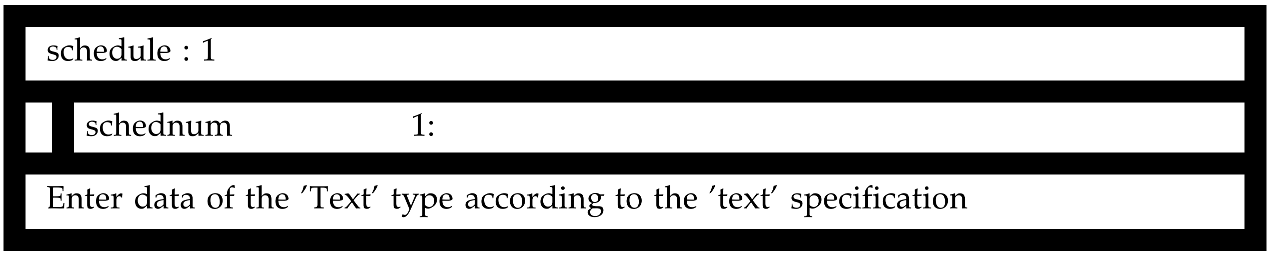
Notice here that the message on the bottom line of the menu has changed. This signifies that κλειω expects data to conform to certain rules. All the previous menus had displayed the message **`Enter data of the 'Text' type according to the 'text' specification`**. This meant that κλειω expected data of **`text`** type as defined by the logical object text. This logical object is usually defined by κλειω, but it would be possible to overwrite κλειω's default logical object for the **`text`** type of data. In this case, a logical object called abbreviation was defined in the file tester.mod, which refers to elements called status. κλειω thus prompts for data that will conform to those rules. If you were to type something that was illegal, κλειω would produce a message like this:

```
***** Error: The following Category expression contains illegal
character(s): r
ESC - continue
```

If you then press Esc, the illegal text remains as the data in the element, but when you try and compile the database it will produce an error message which will prevent the data from being understood. For that reason, if you typed in the letter r here to see what happens, use the up arrow to move up and correct it to i.

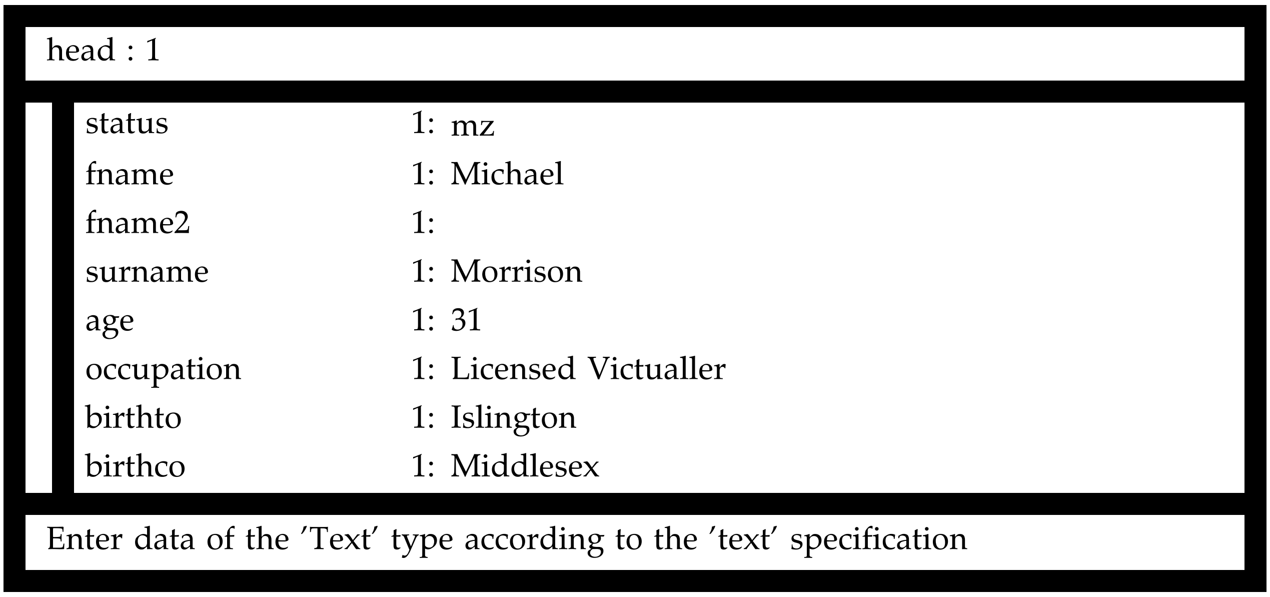
These three fields should be filled with the following information: i, 7 Charlotte Street and Queen's Arms.

Then press F2 again as you want another dependent group.



```
schedule : 1

schednum                 1:

Enter data of the 'Text' type according to the 'text' specification
```

Type 4 in the highlighted field. Then press F2, to move you to a menu for the first head of household in the database. You will see a menu like this (to save on space we have filled it in):



```
head : 1

status                   1: mz
fname                    1: Michael
fname2                   1:
surname                  1: Morrison
age                      1: 31
occupation               1: Licensed Victualler
birthto                  1: Islington
birthco                  1: Middlesex

Enter data of the 'Text' type according to the 'text' specification
```

Press F2 to move you down the hierarchy of the database once more to get to the first menu of related people. This should be filled in like this:

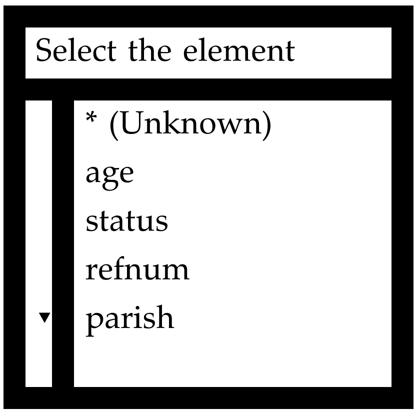
```
relp : 1

status          1: fz
fname           1: Mary
fname2          1: J
surname         1: Morrison
relation        1: wife
age             1: 29
occupation      1:
birthto         1: Cendover
birthco         1: Salop

Enter data of the 'Text' type according to the 'text' specification
```

Now press F1 which, rather than moving you down one level of the hierarchy, allows you to add a further group with the same name. This will give you a menu that looks like this. Again it has been filled in for you:

```
relp : 2

status          1: fs
fname           1: Ellen
fname2          1:
surname         1: Morrison
relation        1: daur
age             1: 7m
occupation      1:
birthto         1: Shrewsbury
birthco         1: Salop

Enter data of the 'Text' type according to the 'text' specification
```

You should now be able to fill in the details for both Elizabeth Morrison and Ann Fox. Try entering the data for these two people and also Catherine Doyle.

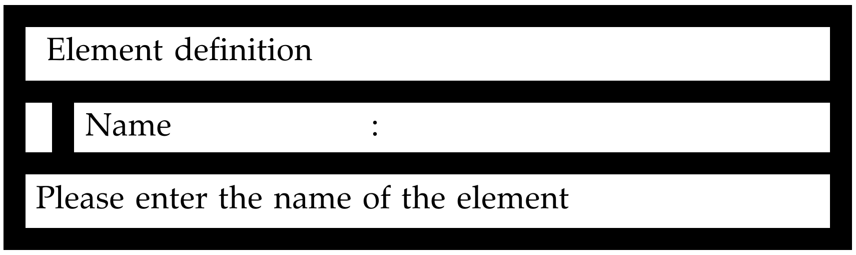
When you get to the end of Catherine Doyle you will notice that there is no field in the menu to put Ireland in. This is because in the .mod file the element country was not defined in a **`position=`** parameter in a part directive. In this case we need to insert an additional element into this group. Press F7. A menu will appear with a list of all the elements in the database mentioned so far:



Select the element

* (Unknown)
age
status
refnum
▾ parish

As we do not want to use an element that has already been defined, we should move the cursor on to the first field in the menu (**`* (Unknown)`**). Press F1 to activate it. This will result in a different menu. (Notice also that this menu has a ▾ symbol in the bottom left hand corner. This means that there are further entries after the entry to the right of that symbol.)

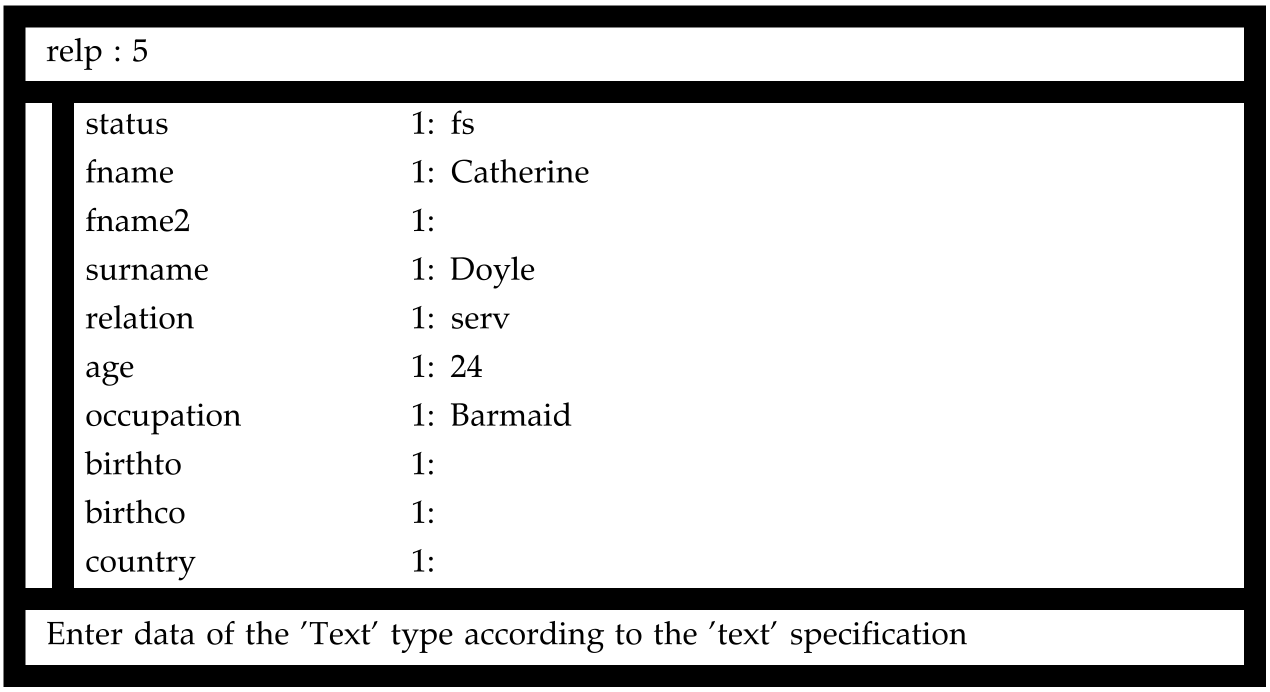


Element definition

Name :

Please enter the name of the element

Enter the name of the new element that you would like to define. In this case you should enter the name country. Then press F1 to activate this menu. Your menu should have changed to take account of this new element:



```
relp : 5

status          1:  fs
fname           1:  Catherine
fname2          1:
surname         1:  Doyle
relation        1:  serv
age             1:  24
occupation      1:  Barmaid
birthto         1:
birthco         1:
country         1:

Enter data of the 'Text' type according to the 'text' specification
```

You can now enter the data in this field.

You should now have entered the whole of that house. Press Esc twice to get to a new schedule menu. You can then continue to enter the data. You should have no problems entering any of the remaining data.

All of the usual features that can be used in a normal .dat file can also be used in the menu version.

Press F10 when you are within a menu, while creating a data file. A menu with a list of all the functions that are currently available are listed. These are:

- F1 Include the present group

  This key includes the group that you are currently inputting into the data file. Once κλειω has added that group to the data file it will consider that you want to include another similar group. In effect, you will get an empty menu for the same group that you have been working on. You also should press this key before you press ESC to stop working on entering data.

- F2 Selection of dependent groups

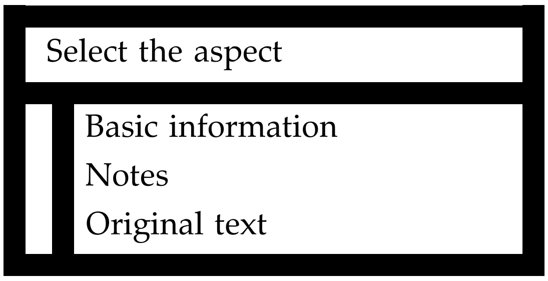
  This key moves you to a menu for a group subordinate to the one that you are currently working on. If you were working with a database with a slightly more complicated structure than this sample database, you would be given the choice of group dependent on the one you were working on. Thus, if there is only one group dependent on the group you are working on and you press F2, κλειω will automatically move you down one level. If you have more than one group dependent on the group that you are working on, κλειω will give you a choice of all the groups currently subordinate to that group. Note, however, that once you have completed a group and all levels of its hierarchy, you must press F1 to include that group. Otherwise you may lose material.

- F3 Repeat element in next filter

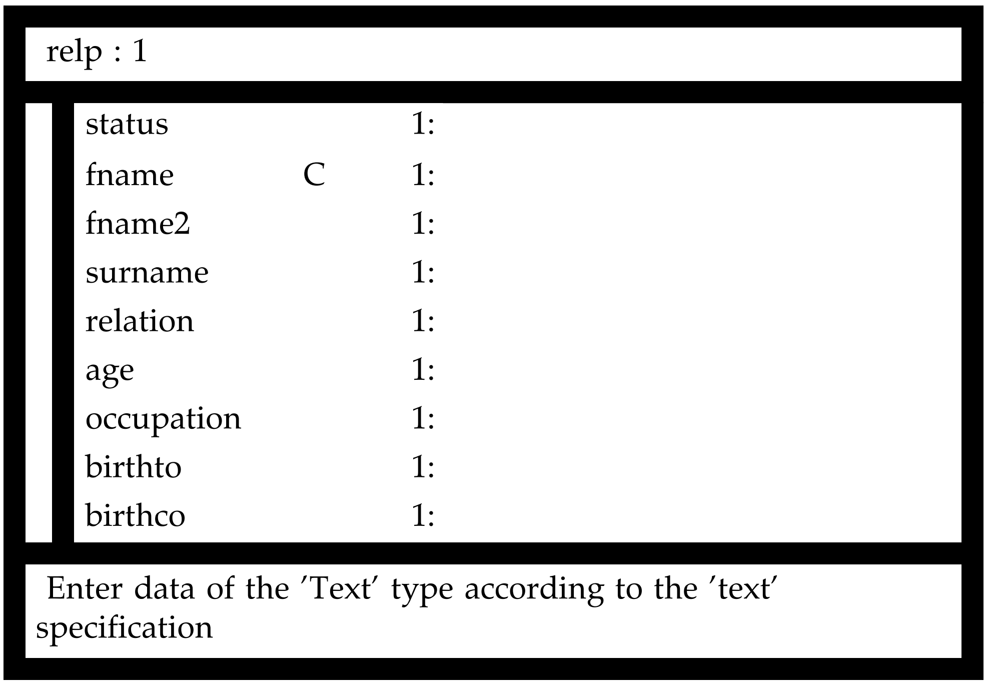
This function key can be used to save time when typing in repetitious material. When you have entered the contents of an element, and the highlighted bar is still on that particular element, you can press F3 to repeat the contents of that element in the next menu. Notice, however, that if you had two groups, person and relp, both with the element surname, and you pressed F3 to repeat a surname from the group head to the group relp, it would not work, as κλειω would think that you wanted to repeat that surname in the next group called head. This works in a similar way to the **`cumulate=`** parameter in conjunction with the **`read`** command.

- F4 Choosing an aspect

This function key allows you to choose the aspects of an element. When you are in an input filter and you press F4, κλειω produces a menu like the one below. This menu refers to the current element.



If you want to select an aspect, move down to the aspect you want and press F1.

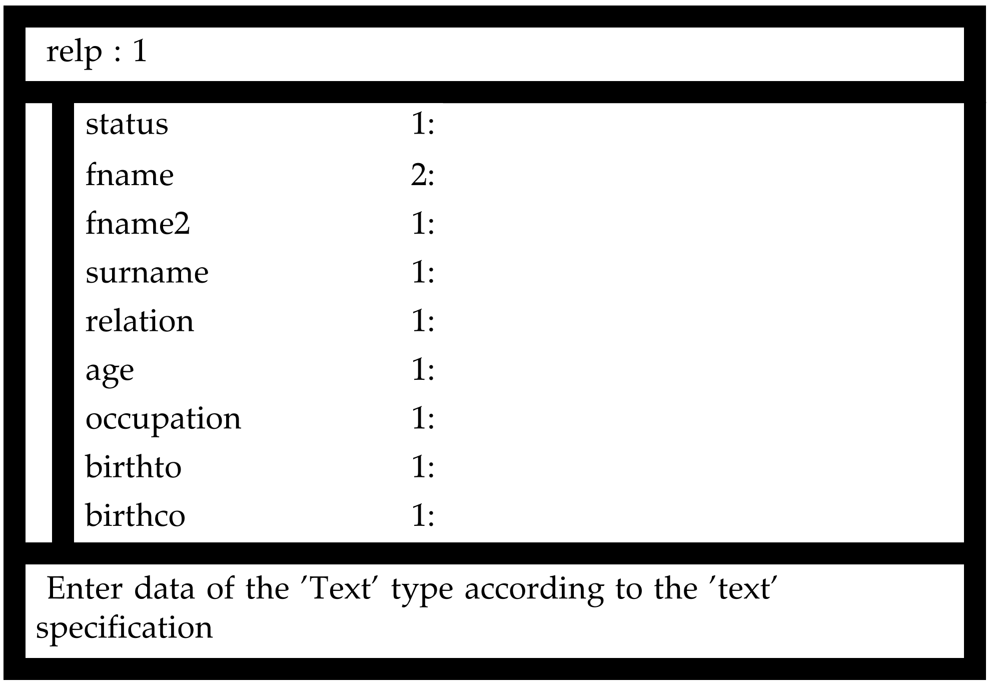


```
relp : 1

status              1:
fname        C      1:
fname2              1:
surname             1:
relation            1:
age                 1:
occupation          1:
birthto             1:
birthco             1:

 Enter data of the 'Text' type according to the 'text'
specification
```

The letter C in this menu represents a comment aspect. If you also wanted to add an 'original' aspect, you would have to press F4 again and select **`original text`**, and press F1. This would return you to the same menu, with an O in the place of the C. If you then wanted to return to the basic information, you would need to repeat the same process, this time selecting **`Basic information`**.

- F5 Insert an entry (after the current one)

This function key replaces the entry separator in the text edited .dat file. When you are in a menu, you can press F5 to insert another entry within the element. For example, if you had pressed F5 in the fname field of a relp menu, the menu would change to look like the one below. Notice that the ordinal number in the middle of that menu has increased from one to two. This designates the second entry in that element.



```
relp : 1

status          1:
fname           2:
fname2          1:
surname         1:
relation        1:
age             1:
occupation      1:
birthto         1:
birthco         1:

Enter data of the 'Text' type according to the 'text'
specification
```

- F6 Insert an entry (in front of the current one)

This function key performs a similar role to the F5 key, except that it places an entry *before* the entry that you have just been working on.

- F7 Insert an element

The role of this key has been explained above.

- F8 From now on, display element in menu

This function key works in conjunction with the F7 function key. If you have defined a new element which was not in the .mod file, you can ask κλειω to display that newly created element in all subsequent menus for that group. This means that you would not have to use the F7 function key so frequently.

- F9 Change the data type

This function key allows one to change the data type of an element.

There are a further seven functions which are performed by holding down the CTRL key and then pressing the appropriate function key. (As these keys negate the role of function keys there is no CRTL/F2).

- CTRL/F1 Delete present group
- CTRL/F3 Do not repeat the element in the next filter
- CTRL/F4 Delete an aspect
- CTRL/F5 Delete an entry
- CTRL/F6 Delete an entry
- CTRL/F7 Delete an element
- CTRL/F8 Do not display an element

With the commands described in this chapter you should be able to input data into most databases with the menu-driven system. There are further commands that could be described to assist in the data entry of a database, but they are all reasonably well documented in the system.

Envoi

If you have managed to read this far you should have a good idea of how to use the system and should need no real assistance from us. If, however, you do have difficulties which you cannot solve with the help of either this volume or the Reference Manual, you might try contacting the U.K. κλειω Support Team at:

`kleio@qmw.ac.uk`

or, by post:

κλειω Support Team
Humanities Computing Centre
Queen Mary & Westfield College
University of London
Mile End Road
London E1 4NS
United Kingdom

Fax: +44 81 980 8400.

They may be able to help. They would also be interested to hear any comments or feedback that you may have.

Good luck!

Answers to exercises

Exercise 2.1

```
query name=burial;part=p:occupation="alderman"
write part=:each[]
stop
```

Exercise 2.2

```
query name=burial;part=relp:occupation="alderman"
write part=:each[]
stop
```

Exercise 2.3

```
query name=burial;part=p
index part=:surname;
  part=:firstname;
  part=:occupation
stop
```

Exercise 2.4

```
query name=burial;part=:surname
index part=:surname;
  part=:firstname;
  part=:occupation
stop
```

Exercise 2.5

```
query name=burial;part=p:status="f"
index part=:surname;
  part=:firstname
  part=:occupation
stop
```

Exercise 2.6

```
query name=baptism;part=relp
index part=:surname;
  part=:firstname;
  part=:occupation
stop
```

Exercise 2.7

```
query name=baptism;part=:occupation
index part=:surname;
  part=:firstname;
  part=:occupation
stop
```

Exercise 2.8

```
query name=burial;part=:occupation
index part=:occupation="clerke"
stop
```

This is the correct answer. However we think that your most natural answer would have been:

```
query name=burial;part=:occupation="clerke"
index part=:occupation
stop
```

Can you see why κλειω produces the result that it does? Look at the datafile, (type **`edit burial.dat`** at the DOS prompt). The answer to this last exercise will be fully described later.

Exercise 4.1

```
query name=censsamp;part=relp:occupation
write part=:occupation,:surname,:firstname
stop
```

Exercise 4.2

```
query name=censsamp;part=:status="f" and :age="29" equal
write part=:each[]
stop
```

Exercise 4.3

```
query name=censsamp;part=:status="f" and :age="29" greater
write part=:each[]
stop
```

Exercise 4.4

```
query name=censsamp;part=:status="z" and "f" and :age="30" less
write part=:each[]
stop
```

Exercise 4.5

```
query name=censsamp;part=house:status="u"
write part=:address
stop
```

Exercise 4.6

```
query name=censsamp;part=relp:birthto=not "Shrewsbury"
index part=:birthto;
  part=:surname
stop
```

Exercise 4.7

```
query name=censsamp;part=:status="m" and (:status="s" or "w")
   and :occupation=not "Grocer"
index part=:surname;
   part=:firstname;
   part=:status;
   part=:occupation
stop
```

Exercise 4.8

```
query name=censsamp;part=:age="16" less and
   :occupation=not "scholar"
index part=:surname;
   part=:firstname;
   part=:occupation
stop
```

This, the logical answer, does not produce a satisfactory result. Try the following task:

```
query name=censsamp;part=:age="16 less and :occupation=null
index part=:surname;
    part=:firstname;
    part=:occupation
stop
```

Exercise 5.1

```
query name=burial;part=:occupation="organist"
index part=:surname;
    part=:firstname;
    part=:query[]
stop
```

Exercise 5.2

```
query name=baptism;part=p:surname="Love"
index part=:query[];
    part=:firstname;
    part=:bapdat
continue
query name=burial;part=p:surname="Love"
index part=:query[];
    part=:firstname;
    part=:burialdate
stop
```

Exercise 5.3

```
query name=baptism;part=:bapdat
index part=:bapdat;substitute="---//---"
stop

query name=baptism;part=:bapdat="3 Mar 1624" or "24 Sept 1600"
index part=:surname;without=yes;
   part=:firstname;without=yes;
   part=:bapdat
stop
```

Exercise 5.4

```
query name=baptism;part=:bapdat="3 Mar 1624" or "24 Sept 1600"
index part=:surname;without=yes;
   part=:firstname;
   part=:bapdat;
   first=limit
stop
```

Exercise 5.5

```
query name=burial;part=p:relation="wife"
index part=:firstname;without=yes;
   part=:surname;limit=" the ";
   part=:query[];limit=" of ";
   part=relp:title;
   part=:firstname;
   part=:surname
stop
```

Exercise 5.6

```
query name=baptism;part=p
index part=relp:firstname;without=yes;
   part=:surname;limit="'s ";
   part=back[1]:relation;without=yes;
   part=:firstname;limit=" was baptised on ";without=yes;
   part=:bapdat
stop
```

Exercise 5.7

```
query name=censsamp;part=relp
index part=:fname;without=yes
  part=:surname;
  part=back[1]:firstname;
  part=:surname;
  part=back[2]:address
  part=:town
stop
```

Exercise 5.8

Something like the following answer would go a long way to see whether there was in fact any difference in the months in which children of both sexes were born. However, in order to make any valid conclusions we should be using a much larger sample.

```
query name=baptism;part=:status="m"
index part=:month[bapdat];type=count
continue
query name=baptism;part=:status="f"
index part=:month[bapdat];type=count
stop
```

There is another element function which may be of interest. This function, **`:weekday[]`**, will 'calculate' the day of the week of a certain date. Try running the following task. (Remember that if you use this to ensure that the correct calendar system has been defined. The information output from this result is accurate as the calendar has been defined as being Julian, rather than Gregorian.)

```
query name=baptism;part=:status="m"
index part=:weekday[bapdat];type=count
stop
```

Exercise 7.1

```
query name=probate;part=:material
index part=:material;type=count
stop
```

Exercise 7.2

```
query name=probate;part=:material="wood"
write part=:each[]
stop
```

Exercise 7.3

```
query name=probate;part=:material="wood"
cumulation part=:value
stop
```

Notice that κλειω has only selected a small proportion of the goods made out of wood that were selected in the last task. This is because the **cumulation** command only retrieves those items which have a value. Many of the items in the database only have a collective value rather than an individual value. In this case the same result could have been obtained using the following task:

```
query name=probate;part=piece:material="wood"
cumulation part=:value
stop
```

Exercise 7.4

```
query name=probate;part=:quality="old" or "ould" or "oyld"
index part=:quality;type=count
stop
```

Note that, although this task works, it does not do exactly what the question asked. This is another example of where the **:query[]** function might be usefully employed.

```
query name=probate;part=:quality="old" or "ould" or "oyld"
index part=:query[];type=count
stop
```

Exercise 8.1

```
query name=probate;part=catalogue[objects,complete,"beere"]
write part=:each[]
stop
```

Exercise 8.2

```
query name=probate;part=catalogue[objects,starts,"brewing"]
write part=back[2]/p:each[]
stop
```

Exercise 8.3

```
query name=probate;part=catalogue[objects,starts,"brewing"]
index part=:each[];
  part=back[2]/p:surname;
  part=:firstname
stop
```

Exercise 8.4

```
query name=probate;part=catalogue[objects,starts,"brewing"]
index part=root[1]/p:surname;
  part=:firstname
stop
```

Exercise 8.5

The standard answer to this exercise is likely to be:

```
query name=probate;part=relp
index part=:surname;
  part=back[1]/p:surname;
  part=back[1]/relp:firstname;
  part=back[1]/p:firstname
stop
```

However the first part of the result looks like this:

```
Abraham  Clerke  Mathew  Robert  9
Abraham  Clerke  Richard  Robert  9
Abraham  Clerke  Robert  Robert  9
Abraham  Clerke  Rob[ert]  Robert  9
Abraham  Clerke  Simon  Robert  9
```

From this we might deduce that Mathew Abraham, Richard Abraham etc. were all people related to Robert Clerke. However, if we refer to the original data we find the following:

```
inventory$HR01623B11\/2/5 Sept 1623
relp$ma/Simon/Perdew
relp$ma/Rob[ert]/Lamborne
relp$mai/Mathew/Fox
relp$ma/Richard/Siddon
relp$ma/Robert/Abraham
```

Thus only the third person in the original result is a real person. The other four are 'made-up' names consisting of the related person's surname followed by the firstnames of all those relps found in the database. This occurs because in the first line of the task κλειω is asked to go to the part of the database where there is a group called relp. The first **`part=`** parameter of the **`index`** command asks for the information about a surname found in that group to be displayed; the second **`part=`** parameter asks κλειω to go back one level in the database then follow a path down to the group p and display the surname found there; the third **`part=`** parameter tells κλειω to go from there to *all* the first names in the *group* of related people. This causes *all* the first names to be displayed.

A method of avoiding this is to use the function **`query[]`** as a group function. A 'correct' solution to this problem would be:

```
query name=probate;part=relp
index part=query[]:surname;
   part=back[1]/p:surname;
   part=query[]:firstname;
   part=back[1]/p:firstname
stop
```

The reason why this works differently is that in the third **`part=`** parameter κλειω is *only* being directed to the element in the group encountered in the **`query`** command.

Exercise 10.1

```
options lines=0
query name=colebro;part=:occupation
index part=:form['form text="'];
   limit="";
   part=:occupation;
   limit='";number=99';
   maximum=1;
   signs=39;
   identification=order[];
   write=no;
stop target="c:\kleio\q10.1a"
```

```
form text="Bag Manufacturer                  ";number=99
form text="Baker                             ";number=99
form text="Beer Retailer                     ";number=99
form text="Boot Fitter                       ";number=99
form text="Boot Maker                        ";number=99
form text="Brewers Labourer                  ";number=99
form text="Bricklayer                        ";number=99
form text="Bricklayers Laborer               ";number=99
form text="bricklayers labourer              ";number=99
form text="Bricklayer's Labourer             ";number=99
form text="builder                           ";number=99
form text="Builders Labourer                 ";number=99
form text="Cabinet Maker                     ";number=99
form text="Carman                            ";number=99
form text="Carpenter                              ";number=99
form text="Carter                                 ";number=99
form text="Chair Man                              ";number=99
form text="Charwoman                              ";number=99
form text="Chelsea Pensioner                      ";number=99
form text="Chimney Sweep                          ";number=99
form text="City Crier                             ";number=99
form text="Clerk                                  ";number=99
form text="Coachman                               ";number=99
form text="Cole merchant                          ";number=99
form text="Companion                              ";number=99
form text="Cook                                   ";number=99
form text="Cordwainer                             ";number=99
form text="Cordwainers Assistant                  ";number=99
form text="Corn dealers porter                    ";number=99
form text="Corn Miller                            ";number=99
form text="Corn Store Man                         ";number=99
form text="Dairyman                               ";number=99
form text="Dom                                    ";number=99
form text="Domestic Servant                       ";number=99
form text="Drapers Porter                         ";number=99
form text="Drayman                                ";number=99
form text="Dress Maker                            ";number=99
form text="Dressmaker                             ";number=99
form text="Dyer                                   ";number=99
form text="Errand Boy                             ";number=99
form text="Farm Labourer                          ";number=99
form text="Freeholder                             ";number=99
form text="Gardener                               ";number=99
form text="Gas Stoker                             ";number=99
form text="Gen Lab                                ";number=99
form text="Gen Serv                               ";number=99
form text="General Dealer                         ";number=99
form text="General Laborer                        ";number=99
form text="General Laboror                        ";number=99
form text="Genl Labourer                          ";number=99
form text="Greenwich Pensioner                    ";number=99
form text="Grocer                                 ";number=99
form text="Grocers Porter                         ";number=99
form text="Groom                                  ";number=99
form text="Harness Maker                          ";number=99
form text="Hawker                                 ";number=99
form text="Housekeeper                            ";number=99
form text="Housemaid                              ";number=99
form text="Inn Servant                            ";number=99
form text="Invalid Nurse                          ";number=99
form text="Iron & Brass Founder                   ";number=99
form text="Joiner                                 ";number=99
form text="labourer                               ";number=99
form text="labourer general                       ";number=99
form text="late dressmaker                        ";number=99
form text="late Gen[era]l Serv                    ";number=99
form text="late grocer                            ";number=99
form text="late laundress                       ";number=99
form text="late Nurse                           ";number=99
form text="late publican                        ";number=99
form text="late staymaker                       ";number=99
form text="late [?]                             ";number=99
form text="Laundress                            ";number=99
form text="Licensed Victualler                  ";number=99
form text="Maltster                             ";number=99
form text="Maltsters Laborer                    ";number=99
form text="Mantle Maker                         ";number=99
form text="Mason                                ";number=99
form text="Master Chimney Sweep                 ";number=99
form text="Medical Student                      ";number=99
form text="Miller                               ";number=99
form text="Milliner                             ";number=99
form text="Milliner &c                          ";number=99
form text="Needlewoman                          ";number=99
form text="No occupation                        ";number=99
form text="None                                 ";number=99
form text="Nurse                                ";number=99
form text="Nursemaid                            ";number=99
form text="Painter                              ";number=99
form text="Paper Hanger                         ";number=99
form text="Plough Boy                           ";number=99
form text="Plumbers Labourer                    ";number=99
form text="Policeman                            ";number=99
form text="Porter                               ";number=99
form text="Postman                              ";number=99
form text="Printers Compositor                  ";number=99
form text="prison warder                        ";number=99
form text="Purveyor of Milk                     ";number=99
form text="Scholar                              ";number=99
form text="shoemaker                            ";number=99
form text="Slater                               ";number=99
form text="Stationers Porter                    ";number=99
form text="Stone Mason                          ";number=99
form text="Stonemason                           ";number=99
form text="Straw Hat Manufacturer               ";number=99
form text="Tailor                               ";number=99
form text="Tallow Chandler                      ";number=99
form text="Tinman                               ";number=99
form text="Upholsterer                          ";number=99
form text="Upholstress                          ";number=99
form text="Verger Winton Cathedral              ";number=99
form text="Waiter                               ";number=99
form text="Whitesmith                           ";number=99
form text="[?]                                  ";number=99
form text="[?] Carpenter                        ";number=99
form text="[?] Servant (unemployed)             ";number=99
```

```
item name=occupations;usage=codebook;source=colebro;
  type=permanent;overwrite=yes
part name=system;type=create
form text="Bag Manufacturer                                ";number=6
form text="Baker                                           ";number=4
form text="Beer Retailer                                   ";number=4
form text="Boot Fitter                                     ";number=4
form text="Boot Maker                                      ";number=6
form text="Brewers Labourer                                ";number=18
form text="Bricklayer                                      ";number=6
form text="Bricklayers Laborer                             ";number=18
form text="bricklayers labourer                            ";number=18
form text="Bricklayer's Labourer                           ";number=18
form text="builder                                         ";number=6
form text="Builders Labourer                               ";number=18
form text="Cabinet Maker                                   ";number=6
form text="Carman                                          ";number=17
form text="Carpenter                                       ";number=6
form text="Carter                                          ";number=6
form text="Chair Man                                       ";number=6
form text="Charwoman                                       ";number=13
form text="Chelsea Pensioner                               ";number=24
form text="Chimney Sweep                                   ";number=17
form text="City Crier                                      ";number=99
form text="Clerk                                           ";number=8
form text="Coachman                                        ";number=12
form text="Cole merchant                                   ";number=4
form text="Companion                                       ";number=11
form text="Cook                                            ";number=11
form text="Cordwainer                                      ";number=7
form text="Cordwainers Assistant                           ";number=18
form text="Corn dealers porter                             ";number=18
form text="Corn Miller                                     ";number=4
form text="Corn Store Man                                  ";number=18
form text="Dairyman                                        ";number=3
form text="Dom                                             ";number=12
form text="Domestic Servant                                ";number=12
form text="Drapers Porter                                  ";number=18
form text="Drayman                                         ";number=99
form text="Dress Maker                                     ";number=6
form text="Dressmaker                                      ";number=6
form text="Dyer                                            ";number=6
form text="Errand Boy                                      ";number=18
form text="Farm Labourer                                   ";number=3
form text="Freeholder                                      ";number=14
form text="Gardener                                        ";number=2
form text="Gas Stoker                                      ";number=18
form text="Gen Lab                                         ";number=18
form text="Gen Serv                                        ";number=12
form text="General Dealer                                  ";number=4
form text="General Laborer                                 ";number=18
form text="General Laboror                                 ";number=18
form text="Genl Labourer                                   ";number=18
```

```
   form text="Greenwich Pensioner                    ";number=24
   form text="Grocer                                 ";number=4
   form text="Grocers Porter                         ";number=18
   form text="Groom                                  ";number=12
   form text="Harness Maker                          ";number=6
   form text="Hawker                                 ";number=23
   form text="Housekeeper                            ";number=11
   form text="Housemaid                              ";number=12
   form text="Inn Servant                            ";number=17
   form text="Invalid Nurse                          ";number=11
   form text="Iron & Brass Founder                   ";number=6
   form text="Joiner                                 ";number=6
   form text="labourer                               ";number=18
   form text="labourer general                       ";number=18
   form text="late dressmaker                        ";number=24
   form text="late Gen[era]l Serv                    ";number=24
   form text="late grocer                            ";number=24
   form text="late laundress                         ";number=24
   form text="late Nurse                             ";number=24
   form text="late publican                          ";number=24
   form text="late staymaker                         ";number=24
   form text="late [?]                               ";number=24
   form text="Laundress                              ";number=13
   form text="Licensed Victualler                    ";number=4
   form text="Maltster                               ";number=4
   form text="Maltsters Laborer                      ";number=18
   form text="Mantle Maker                           ";number=6
   form text="Mason                                  ";number=6
   form text="Master Chimney Sweep                   ";number=6
   form text="Medical Student                        ";number=9
   form text="Miller                                 ";number=6
   form text="Milliner                               ";number=6
   form text="Milliner &c                            ";number=6
   form text="Needlewoman                            ";number=13
   form text="No occupation                          ";number=22
   form text="None                                   ";number=22
   form text="Nurse                                  ";number=11
   form text="Nursemaid                              ";number=11
   form text="Painter                                ";number=6
   form text="Paper Hanger                           ";number=6
   form text="Plough Boy                             ";number=18
   form text="Plumbers Labourer                      ";number=18
   form text="Policeman                              ";number=9
   form text="Porter                                 ";number=18
   form text="Postman                                ";number=9
   form text="Printers Compositor                    ";number=6
   form text="prison warder                          ";number=9
   form text="Purveyor of Milk                       ";number=4
   form text="Scholar                                ";number=20
   form text="shoemaker                              ";number=6
   form text="Slater                                 ";number=6
   form text="Stationers Porter                      ";number=18
   form text="Stone Mason                            ";number=6
```

```
form text="Stonemason                                  ";number=6
form text="Straw Hat Manufacturer                      ";number=7
form text="Tailor                                      ";number=6
form text="Tallow Chandler                             ";number=4
form text="Tinman                                      ";number=18
form text="Upholsterer                                 ";number=6
form text="Upholstress                                 ";number=6
form text="Verger Winton Cathedral                     ";number=8
form text="Waiter                                      ";number=18
form text="Whitesmith                                  ";number=6
form text="[?]                                         ";number=99
form text="[?] Carpenter                               ";number=99
form text="[?] Servant (unemployed)                    ";number=99
exit name=occupations
```

Exercise 10.2

```
item name=occupations;usage=codebook;source=colebro;
   type=permanent
part name=system;type=insert
write number=1;text="Agricultural Self-employed"
write number=2;text="Skilled Agricultural Workers"
write number=3;text="Agricultural Labourers"
write number=4;text="Shopkeepers, traders"
write number=5;text="Skilled Crafts, non-industrial"
write number=6;text="Manufacturers"
write number=7;text="Skilled Industrial Craftsmen"
write number=8;text="Upper Professional"
write number=9;text="Professional"
write number=10;text="Clerical"
write number=11;text="Upper Servants"
write number=12;text="General Servants"
write number=13;text="Lower Servants"
write number=14;text="Private Income Recipient"
write number=15;text="Rentiers"
write number=16;text="Annuitants"
write number=17;text="Semi-skilled/Service"
write number=18;text="Unskilled"
write number=19;text="Supervisory"
write number=20;text="Children"
write number=21;text="Housewives"
write number=22;text="No occupation"
write number=23;text="Paupers"
write number=24;text="Retired People"
write number=25;text="Visitors"
write number=99;text="Unknown"
exit name=occupations
```

Exercise 10.3

```
query name=colebro;part=:occupation
translation target=pcspss;
   first="jobs.num";
   second="jobs.sps"
case part=:codebook[:occupation,system,occupations];
   name=occupation;write="code";
   part=:age;name=age;write="age"
stop
```

Exercise 11

However you create your result, the only part of the result that may need explanation is the part pertaining to getting the total value of the person's wealth in pounds. This is done like so:

```
part=:value&::form["240",number]
```

Your query might be slightly differently formulated, but you will need something like this to complete this exercise correctly.

Exercise 12.1

Anderson, Andersen = 1647, 1647
Bergman, Brigham = 2736, 2736
Fischer, Fisher, Fisshire = 2370, 2370, 2370
Oldroyd, Holroyd = 1547, 1547

Exercise 12.2

The database should look something like this:

```
database name=newcast;first=d;overwrite=yes
part name=d;
    position=id,name
exit name=newcast
read name=newcast
d$1/Cay
d$2/Kay
d$3/Carr
d$4/Kerr
d$5/Leighton
d$6/Leaton
d$7/Layton
d$8/Turnbull
```

```
d$9/Trumball
d$10/Hindmarsh
d$11/Hynmers
d$12/Atkinson
d$13/Atchison
.
.
.
```

and the soundex commands should look something like this:

```
item name=code;usage=soundex;type=permanent;source=newcast;
  overwrite=yes
conversion without="aeiouywhg"
part signs="ck"
part signs="r"
part signs="l"
part signs="tj"
part signs="nmds"
part signs="vxq"
part signs="bpf"
exit name=code
```

An example of a possible task:

```
query name=newcast;part=:soundex[:name,code]=
  :soundex[:form["Cay"],code
write part=:soundex[:name,code],:name
stop
```

Exercise 13.3

```
query name=deaths;part=:surname
index part=:surname;limit", ";
  part=:fname;limit=" was born on ";
  part=:births<births>:dob;
  identification=:order[];write=no
stop
```

Exercise 14.1

The database needed to create these two shapes can be seen below. Essentially the database is the same used in tasks ex14.1, ex14.2 and ex14.3.

```
database name=exercise;first=x;overwrite=yes
element name=grid-reference;type=location;location=shape
part name=x;
  position=id,name,grid-reference
exit name=exercise

item name=shape;usage=location;source=exercise;type=permanent
type form=arcinfo
location first=shape1
1 1
5 1
3 5
end
location first=shape2
3 0
1 4
5 4
end
exit name=shape

read name=exercise
x$a1/x/shape1
x$a2/x/shape2
```

To display this information the following task would be necessary:

```
query name=exercise;part=:name="x"
mapping part=:grid-reference
stop
```

Exercise 14.2

```
item name=shape;usage=location;source=exercise;type=permanent
type form=arcinfo
location first=shape1;colour=red
1 1
5 1
3 5
end
location first=shape2;colour=blue
3 0
1 4
5 4
end
exit name=shape
```

Exercise 14.3

```
options lines=0
query name=colebro;part=head:occupation
index part=:form['form text="'];limit="";
  part=:occupation;limit='";number=';maximum=1;sign=39;
    identification=:order[];write=no
stop target="ex14.3"
```

This query produces an index of all the occupations of heads of household, with some additional text necessary to add new codes to a codebook. The result of the file is sent to a file ex14.3. Part of the result follows:

```
form text="                                   ";number=
form text="Bag Manufacturer                   ";number=
form text="Baker                              ";number=
form text="Beer Retailer                      ";number=
form text="Boot Maker                         ";number=
form text="Brewers Labourer                   ";number=
```

This result needs to be changed slightly to create the codebook classifying the occupations of these heads of household.

Exercise 14.4

```
item name=status;source=colebr;usage=codebook;type=permanent
part type=insert;name=system
form text="                                   ";number=3
form text="Bag Manufacturer                   ";number=2
form text="Baker                              ";number=2
form text="Beer Retailer                      ";number=2
form text="Boot Maker                         ";number=2
form text="Brewers Labourer                   ";number=1
.
.
.
exit name=status
```

Above is part of the logical object called status which creates a codebook. The code 1 has been allocated to manual labourers, 2 to artisans and 3 to others.

Once this codebook has been created, it is possible to run the following task to display all the objects as specified in the question.

Exercise 14.5

The task that follows is one of many possible solutions to this question. This solution is however slightly faulty. Unless you had studied the data file carefully, it would not be obvious exactly why this is the case. In this task, κλειω is asked to produce a map where those people with the occupational code 1 have their houses displayed in red, and those people whose occupational code is 2 have their houses displayed in green, but κλειω is asked to display all the other houses in the database in blue. However there are some houses in the location file that are NOT represented in the data file. Therefore we do NOT know the occupation of the head of household of these houses. Therefore they should not necessarily be displayed in blue. The second example, shown further below, takes this problem into account and displays those houses for which we know the head of household falls into the occupational category 3 in blue and the remainer of houses in white.

FAULTY ANSWER

```
query name=colebr;
  part=head:codebook[:occupation,system,status]="1"
confirm name=first
mapping always=yes;sign=no
mapping part=back[2]:site;sign=no;colour=red
exit name=first
negate name=second
query part=:codebook[:occupation,system,status]="2"
confirm name=third
mapping part=back[2]:site;sign=no;colour=green
exit name=third
negate name=fourth
mapping total=yes;sign=no;colour=blue
exit name=fourth
exit name=second
stop
```

An English translation of this task might read: I am interested in a database called colebr. Check the codebook for all entries with 1 as the system code. Once those people have been found, display the city wall without a label and these houses in red. For all those people whose occupational code number is not 1, check to see if they equal 2, if so display the houses referring to those people in green. For all those people whose code is neither 1 nor 2 (i.e. all that is left), display their houses in blue.

CORRECT ANSWER

```
query name=colebr;
  part=head:codebook[:occupation,system,status]="1"
confirm name=group1
mapping always=yes;sign=no
mapping part=back[2]:site;sign=no;colour=red
exit name=group1
negate name=group23
query part=:codebook[:occupation,system,status]="2"
confirm name=group2
mapping part=back[2]:site;sign=no;colour=green
exit name=group2
negate name=group3
query part=:codebook[:occupation,system,status]="3"
confirm name=group4
mapping part=back[2]:site;sign=no;colour=blue
negate name=group34
mapping total=yes;sign=no;
exit name=group34
exit name=group4
exit name=group3
exit name=group23
stop
```

Exercise 14.6

```
query name=colebr;part=relp:relation="wife"
  and :occupation=null
mapping part=back[3]:site;
        usage=solid;usage=solid;
        colour=contrast;colour=contrast;
        sign=no;
        write="ex14.4";
        target="14_4.plt";
        overwrite=yes
mapping total=yes;sign=no
mapping always=yes;sign=no
location usage=Postscript
```

Bibliography

Writings of Manfred Thaller in English

'Automation on Parnassus. CLIO — A Databank Orientated System for Historians', *Historical Social Research/Historische Sozialforschung*, 15 (1980), pp. 40–65.

'The Winds of Change', Problems of a Databank Oriented System Using the Concept of Fuzzy Sets, in: Papers Invented at the 1981 Joint Conference of IFDO and IASSIST, Grenoble, 14–18 September 1981.

'Recycling the Drudgery, On the Integration of Software Supporting Secondary Analysis of Machine-Readable Texts into a DBMS', in *Linguistica Computazionale* 3 (1983), Supplement, pp. 253–68.

'Beyond Collecting: the Design and Implementation of CLIO, a DBMS for the Social-Historical Sciences', in *Data Bases in the Humanities and Social Sciences* 2, ed. Robert F.Allen (Paradigm Press, Florida, 1985), pp. 328–34.

'A Draft Proposal for the Coding of Machine Readable Sources', in *Historical Social Research/Historische Sozialforschung*, 40 (1986), pp. 3–46, repr. in *Modelling Historical Data*, ed. D. Greenstein. Halbgraue Reihe zur historischen Fachinformatik, A11 (St. Katharinen, 1991), pp. 19–64.

'Can We Afford to Use the Computer; Can We Afford Not to Use it?', in *Informatique et Prosopographie*, ed. H. Millet (Paris, 1986), pp. 339–52.

'Methods and Techniques of Historical Computation', in *History and Computing*, eds. P. Denley & D. Hopkin (Manchester, 1987), pp. 147–56.

'The Daily Life of the Middle Ages. Editions of Sources and Data Processing', *Medium Aevum Quotidianum*, 10 (1987), pp. 6–29.

'Data Bases v. Critical Editions', in *Data Base Oriented Source Editions*. Papers from two Sessions at the 23rd International Congress of Medieval Studies, Kalamazoo, 5–8 May 1988, ed. M. Thaller, pp. 1–8; also in *Historical Social Research/Historische Sozialforschung*, 13:3 (1988), pp. 129–39.

'A Draft Proposal for a Standard Format Exchange Program', in *Standardisation et échange des bases de données historiques*, ed. J.-P. Genet (CNRS, Paris, 1988), pp. 329–75.

'The Need for a Theory of Historical Computing', in *History and Computing II*, eds. P. Denley, S. Fogelvik & C. Harvey (Manchester, 1989), pp. 2–11.

'Have Very Large Data Bases Methodological Relevance?', in *Conceptual and Numerical Analysis of Data*, ed. O. Opitz (Berlin, 1989), pp. 311–26.

'Databases and Expert Systems as Complementary Tools for Historical Research', *Tijdschrift voor Geschiedenis*, 103 (1990), pp. 233–47.

'The Historical Workstation Project', in *Computers and the Humanities*, 25 (1991), pp. 149–62.

'The Need for Standards: Data Modelling and Exchange', in *Modelling Historical Data*, ed. D. Greenstein. Halbgraue Reihe zur historischen Fachinformatik, A11 (St. Katharinen, 1991), pp. 1–18.

'The Historical Workstation Project', in *Historical Social Research/Historische Sozialforschung*, 16:4 (1991), pp. 51–61 (introduction to section).

'The Historical Workstation Project', in *Histoire et Informatique*. V^{e} Congrès 'History & Computing', 4–7 Septembre 1990 à Montpellier (Montpellier, 1992), ed. J. Smets, pp. 251–60 (introduction to section).

'On the Conception, Training and Employment of Historical Data and Knowledge Daemons', in *Eden or Babylon?*, ed. J. Oldervoll. Halbgraue Reihe zur historischen Fachinformatik, A13 (St. Katharinen, 1992), pp. 53–67.

'The Processing of Manuscripts', in *Images and Manuscripts in Historical Computing*, ed. M. Thaller. Halbgraue Reihe zur historischen Fachinformatik, A14 (St. Katharinen, 1992), pp. 41–72.

'The Role of Summer Schools in Teaching History and Computing', in *Towards an International Curriculum for History and Computing*, eds. D. Spaeth, P. Denley, V. Davis & R. Trainor. Halbgraue Reihe zur historischen Fachinformatik, A12 (St. Katharinen, 1992), pp. 49–53.

‘What is "source oriented data processing"; what is a "historical information science"?’, paper given to ‘New information technologies in historical research and teaching’, June 1992 in Uzhgorod, Ukraine, published in Russian in *Istoriia i comp'iuter. Novye informatsionnye tekhnologii v istoricheskikh issledovanii akh i obrazovanii*, eds. Leonid I. Borodkin & Wolfgang Levermann. Halbgraue Reihe zur historischen Fachinformatik, A15 (St. Katharinen, 1993), pp. 5–18.

‘The Archive on the Top of your Desk? On Self-Documenting Image Files’, in *Image Processing in History: towards Open Systems*, eds. Jurij Fikfak & Gerhard Jaritz. Halbgraue Reihe zur historischen Fachinformatik, A16 (St. Katharinen, 1993), pp. 21–24.

‘Levels of "Computing in History" Curricula’, in *The Teaching of Historical Computing: An International Framework*, eds. Virginia Davis, Peter Denley, Donald Spaeth & Richard Trainor. Halbgraue Reihe zur historischen Fachinformatik, A17 (St. Katharinen, 1993), pp. 5–9.

‘Historical Information Science: Is There Such a Thing? New Comments on an Old Idea’, in *Discipline umanistiche e informatica. Il problema dell'integrazione* (Seminario, Roma, 8 ottobre 1991), ed. Tito Orlandi. Contributi del Centro Linceo Interdisciplinare ‘Beniamino Segre’, 87 (Accademia Nazionale dei Lincei, Rome, 1993), pp. 51–86.

κλειω. *A Database System*. Halbgraue Reihe zur historischen Fachinformatik, B11 (St. Katharinen, 1993).

Other κλειω literature in English

Peter Becker, ‘Illsex — A Databank for Studying Illegitimacy, *Historical Social Research/Historische Sozialforschung*, 15:1 (1990), pp. 59–65.

Susanne Botzem, Ingo H. Kropač, ‘Integrated Computer Supported Editing, Approaches and Strategies’, in *Historical Social Research/Historische Sozialforschung*, 16:4 (1991), pp. 106–15.

Susanne Botzem, Ingo H. Kropač, ‘As You Like It or Archiving, Editing and Analysing Medieval Manuscripts’, in *Histoire et Informatique*. V^{e} Congrès ‘History & Computing’, 4–7 Septembre 1990 à Montpellier, ed. J. Smets (Montpellier, 1992), pp. 267–78.

Andrea Bozzi & Giuseppe Cappelli, 'A Latin Morphological Analyser', in *Data Base Oriented Source Editions*. Papers from two sessions at the 23rd International Congress of Medieval Studies, Kalamazoo, 5–8 May 1988, ed. M. Thaller, pp. 47–54.

Bettina Callies, Lothar Kolmer, 'A Computerised Medieval City Archive: the Project "Regensburger Bürger- und Häuserbuch"', in *History and Computing II*, eds. P. Denley, S. Fogelvik & C. Harvey (Manchester, 1989), pp. 266–72.

Peter Denley, 'Source-Oriented Prosopogyaphy: κλειω and the Creation of a Data Bank of Italian Renaissance University Teachers and Students', in *Storia & Multimedia*, eds. F. Bocchi and P. Denley (Grafis Edizioni, Bologna, forthcoming, 1994).

Josef Ehmer, 'The Vienna Data Base on European Family History', in *Data Bases in the Humanities and Social Sciences* 2, ed. Robert F. Allen (Paradigm Press, Florida, 1985), pp. 113–16.

Claudia Engel, Eckart Voland, 'Female Choice in Humans: A Conditional Mate Selection Strategy of the Krummhörn Women (Germany 1720–1874), *Ethology* 84 (1990), pp. 144–54.

Thomas Engelke, 'EDP-Based Projects at the Regensburg Archives', in *Histoire et Informatique*. V^e Congrès 'History & Computing', 4–7 Septembre 1990 à Montpellier, ed. J. Smets (Montpellier, 1992), pp. 279–87.

Hans-Christoph Hobohm, 'Using Databases for Everyday Work in Literary History, Exchange and Standardization Problems', in *Standardisation et échange des bases de données historiques*, ed. J.-P. Genet (CNRS, Paris, 1988), pp. 323–28.

Hans-Christoph Hobohm, 'Establishing a Reconstructive Metasource on Censorship of Novels in the Early French Enlightenment', in *Computers in the Humanities and Social Sciences*. Proceedings of the Cologne Computer Conferece 1988, eds. H. Best, E. Mochmann & M. Thaller (Munich, 1991), pp. 130–35.

Kathrin Homann, 'StanFEP — Standardization without Standards', in *Histoire et Informatique*. V^e Congrès 'History & Computing', 4–7 Septembre 1990 à Montpellier, ed. J. Smets 1992), pp. 289–99.

Gerhard Jaritz, 'Daily Life in the Middle Ages, Iconography of Medieval Art and the Use of EDP', *Historical Social Research/Historische Sozialforschung*, 21 (1981), pp. 43–55.

Gerhard Jaritz, 'Daily Life in Medieval Literature', *Medium Aevum Quotidianum*, Newsletter 2 (1984), pp. 2–23.

Gerhard Jaritz, 'Finding the Signs, Pictures of Medieval Life', in *Data Base Oriented Source Editions*. Papers from two Sessions at the 23rd International Congress of Medieval Studies, Kalamazoo, 5–8 May 1988, ed. M. Thaller, pp. 15–28.

Gerhard Jaritz, 'Toward Standards of Very Different Materials: Problems of Standardization in EDP-Supported Research in the Material Culture of the Middle Ages', in *Standardisation et échange des bases de données historiques*, ed. J.-P. Genet (CNRS, Paris, 1988), pp. 153–60.

Gerhard Jaritz, 'The Image as Historical Source or: Grabbing Contexts', in *Historical Social Research/Historische Sozialforschung*, 16:4 (1991), pp. 100–105.

Gerhard Jaritz, '"New Patterns of Response" – Digital Image Processing and the Explanation of Medieval Pictures', in *Histoire et Informatique*. V^{e} Congrès 'History & Computing', 4–7 Septembre 1990 à Montpellier (Montpellier, 1992), ed. J. Smets, pp. 261–66.

Gerhard Jaritz, *Images. A Primer of Computer-Supported Analysis with* κλειω *IAS*. Halbgraue Reihe zur historischen Fachinformatik, A22 (St. Katharinen, 1993).

Gerhard Jaritz & Albert Müller, 'The History of Medieval and Early Modern Migration. Computer-Supported Methods and Results', in *History and Computing II*, eds. P. Denley, S. Fogelvik & C. Harvey (Manchester, 1989), pp. 161–68.

Gerhard Jaritz & Barbara Schuh, 'Describing the Indescribable', in *Images and Manuscripts in Historical Computing*, ed. M. Thaller. Halbgraue Reihe zur historischen Fachinformatik, A14 (St. Katharinen, 1992), pp. 143–53.

Ingo H. Kropač, 'Homo ex Machina. Prosopography and Cartularies', in *Data Base Oriented Source Editions*. Papers from two Sessions at the 23rd International Congress of Medieval Studies, Kalamazoo, 5–8 May 1988, ed. M. Thaller, pp. 37–45, and in *Computers in the Humanities and Social Sciences*. Proceedings of the Cologne Computer Conferece 1988, eds. H. Best, E. Mochmann & M. Thaller (Munich, 1991), pp. 97–102.

Ingo H. Kropač, 'Who's Who in the Southeast of Germany: the Design of the Prosopographical Data Bank of Graz University', in *History and Computing II*, eds. P. Denley, S. Fogelvik & C. Harvey (Manchester, 1989), pp. 273–79.

Ingo H. Kropač, 'The Prosopographical Data Bank on the History of the South-East Territories of the Old "Reich" up to 1250', in *Data Bases in the Humanities and Social Sciences*, ed. L. J. McCrank (Paradigm Press, Florida, 1990), pp. 383–90.

Ingo H. Kropač, Ursula Leiter-Köhrer, 'Parsing the Past – Reflections on the Analytical Semantic Parsing System (ASPS)', *Histoire et Informatique*. V^{e} Congrès 'History & Computing', 4–7 Septembre 1990 à Montpellier, ed. J. Smets (Montpellier, 1992), pp. 301–13.

Ursula Leiter-Köhrer, 'Linguistic Knowledge as a Background Component of an Application Oriented Workstation', in *Historical Social Research/Historische Sozialforschung*, 16:4 (1991), pp. 89–99.

B. Pöttler, 'Modelling Historical Data. Probate Inventories as a Source for the History of Everyday Life', in *Storia & Multimedia*, eds. F. Bocchi and P. Denley (Grafis Edizioni, Bologna, forthcoming, 1994).

Wolfgang Levermann, 'Historical Data Bases and the Context Sensitive Handling of Data. Towards the Development of Historical Data Base Management Software', in *Historical Social Research/Historische Sozialforschung*, 16:4 (1991), pp. 74–88.

Carola Lipp, 'Symbolic Dimensions of Serial Sources. Hermeneutical Problems of Reconstructing Political Biographies Based on Computerized Record Linkage', *Historical Social Research/Historische Sozialforschung* 15:1 (1990), pp. 30–40.

D. Sabean, *Property, Production, and Family in Neckarshausen, 1700–1870* (Cambridge, 1990).

J. Smets, 'South French Society and the French Revolution: the Creation of a Large Database with CLIO', in *History and Computing*, eds. P. Denley & D. Hopkin (Manchester, 1987), pp. 49–58.

Peter Teibenbacher, 'The Computer, Oral History and Regional Studies', in *History and Computing II*, eds. P. Denley, S. Fogelvik & C. Harvey (Manchester, 1989), pp. 286–90.

Elisabeth Vavra, 'CLIO, a Computer Program Supporting the Interpretation of the Iconographic Content of Medieval Pictorial Sources', in *Automatic Processing of Art History Data and Documents*, Pisa, Scuola Normale Superiore, 24–27 September 1984, I, pp. 407–20.

Eckhard Voland, 'Differential Reproductive Success within the Krummhörn Population (Germany, 18th and 19th Centuries)', *Behavioral Ecology and Sociobiology*, 26 (1990), pp. 65–72.

Thomas Werner, 'Transforming Machine Readable Sources', in *Historical Social Research/Historische Sozialforschung*, 16:4 (1991), pp. 62–73.

For a bibliography of works in German, see 'κλειω-Bibliographie', in Thomas Engelke, Jürgen Nemitz & Carolin Trenkler (eds.), *Historische Forschung mit* κλειω, Halbgraue Reihe zur historischen Fachinformatik, A8 (St. Katharinen, 1990).

Index of topics

Index of terms

Max-Planck-Institut für Geschichte, Göttingen