4. Setting up a database

$\kappa\lambda\epsilon\iota\omega$ uses an internal format to manage data. This is specific to $\kappa\lambda\epsilon\iota\omega$ and *cannot* be processed by other programs. Before $\kappa\lambda\epsilon\iota\omega$ can begin to process data, it must convert it to this internal format.

This should be done by entering the data as a simple ASCII file using the text editor of your choice and then importing the ASCII file into $\kappa\lambda\epsilon\iota\omega$ for subsequent processing.

$\kappa\lambda\epsilon\iota\omega$ needs two things in order to convert data imported in this form:

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

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

4.1 Entries

The smallest unit of information in a database which $\kappa\lambda\epsilon\iota\omega$ can access is called an *entry*. Each entry consists of:

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

4.1.1 Core information

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

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

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

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

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

4.1.1.1 Rules for inputting specific data types

The following data types have been defined to date:

4.1.1.1.1 `text` data

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

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

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

4.1.1.1.2 `date` data

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

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

4.1.1.1.3 `number` data

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

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

4.1.1.1.4 `category` data

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

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

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

4.1.1.1.5 `relation` data

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

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

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

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

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

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

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

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

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

4.1.1.1.6 `location` data

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

The names of topographical objects are standard $\kappa\lambda\epsilon\iota\omega$ names.

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

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

4.1.1.1.7 `image` data

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

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

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

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

4.1.2 Status

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

4.1.3 Data type and entry preamble

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

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

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

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

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

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

4.1.4 Examples

We include some examples of correctly-written entries below:

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

4.2 Elements

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

An element is described in terms of the following specifications:

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

4.2.1 Names of elements

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

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

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

4.2.2 Visibility of elements

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

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

4.2.2.1 Visibility clauses

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

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

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

4.2.3 How an element is viewed

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

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

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

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

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

In the task

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

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

4.2.3.1 Elements in a single view

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

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

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

4.2.3.2 Elements in several views

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

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

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

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

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

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

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

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

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

4.2.3.3 Restriction

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

4.2.4 Entries contained in an element

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

4.2.4.1 Aspects

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

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

4.2.4.2 Joining up entries

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

4.2.5 Examples

Here are some examples of correctly written elements:

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

4.3 Groups

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

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

4.3.1 Names of groups

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

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

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

4.3.2 Identification

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

Identifications are constructed as follows:

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

4.3.2.1 Addressability of an identification

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

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

4.3.3 Visibility of groups

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

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

4.3.3.1 Visibility clauses

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

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

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

4.3.4 How a group is viewed

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

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

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

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

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

In the task

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

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

4.3.4.1 Groups in a single view

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

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

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

4.3.4.2 Groups in several views

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

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

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

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

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

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

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

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

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

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

4.3.5 Relationships with other groups

4.3.5.1 Implicit association

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

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

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

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

4.3.5.2 Explicit association

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

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

The following association characters have been defined:

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

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

4.3.5.3 Relationships in other views

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

4.3.5.4 Relocating groups within the structure

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

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

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

4.3.5.4.1 More dramatic relocations within the structure

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

4.3.5.5 Ordinal number of a group

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

4.3.6 Elements held in groups

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

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

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

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

4.3.7 Examples

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

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

4.4 Documents

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

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

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

A document's identification is always absolutely addressable.

4.5 Dynamic fixed-field formats

4.5.1 Preliminary remarks

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

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

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

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

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

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

4.5.2 Fixed formats

4.5.2.1 Input

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

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

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

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

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

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

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

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

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

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

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

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

4.5.2.2 Structure description and the `read` command

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

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

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

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

would cause the value

```
        2225
```

to be interpreted as

```
        numbers=22.25
```

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

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

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

```
        02031786 ...  12102
```

as

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

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

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

the read command

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

would cause the input data

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

to be imported as if it was identical to

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

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

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

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

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

If you use the part directive

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

the following data

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

is correctly interpreted as

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

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

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

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

would result in an error message.

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

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

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

would also result in an error message.

4.5.3 Dynamic formats and special cases

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

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

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

Using the `part` directive

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

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

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

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

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

and entering the data as follows:

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

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

4.5.3.1 Input options

There is an option available for dealing with such cases. It is based directly on the following underlying definition of $\kappa\lambda\epsilon\iota\omega$'s dynamic fixed-field formats:

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

Between this delimiting pair of commas, you can enter the element using any of the options $\kappa\lambda\epsilon\iota\omega$ provides. In this case the data can be represented as follows:

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

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

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

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

Thus by using the following `read` command:

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

we could enter the data listed above as follows:

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

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

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

Or to put it another way:

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

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

4.5.3.2 Extra features in the structure description

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

4.5.3.2.1 Comments to Lists as a whole

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

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

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

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

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

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

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

the following additional rules apply:

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

Thus the data

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

would be interpreted as follows:

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

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

Hence

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

is thus interpreted in both cases as:

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

4.5.3.2.2 Dynamic formats with explicit labels

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

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

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

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

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

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

This would be accomplished by the `series` directive:

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

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

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

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

the input data

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

will be stored as if the would have been entered as

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

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