5. Structure definitions

5.1 General

The following sections (5.2 to 5.7) start by defining the commands available to users for creating simple $\kappa\lambda\epsilon\iota\omega$ databases. These simple databases fulfil the following conditions:

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

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

5.1.1 Function of a structure definition

Structure definitions

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

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

5.1.2 General conventions

A structure definition consists of:

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

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

Directives in structure definitions are formatted in exactly the same way as $\kappa\lambda\epsilon\iota\omega$ commands,

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

5.2 The `database` command

The general form of the `database` command is:

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

This command declares the new database's general characteristics.

5.2.1 Defining structure designations

If all the parameters required to identify the structure are present $\kappa\lambda\epsilon\iota\omega$ instructs your computer's operating system to create the necessary number of physical files and then initialises the tables which describe these files.

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

5.2.1.1 The `name=` parameter

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

5.2.1.2 The `first=` parameter

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

5.2.2 Saving your structure definition

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

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

5.2.2.1 The `type=` parameter

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

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

5.2.2.2 The `overwrite=` parameter

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

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

5.2.3 Controlling the scope of printed output

By default $\kappa\lambda\epsilon\iota\omega$ prints out the most important codes for the entire database you are creating once you have completed your structure definition. These include the valid data signal character set, document names, setup date, etc.

5.2.3.1 The `write=` parameter

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

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

5.2.4 Identifying groups

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

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

5.2.4.1 The `identification=` parameter

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

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

5.2.4.2 The `more=` parameter

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

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

5.2.5 Examples

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

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

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

5.3 The part directive

The general form of the part directive is:

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

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

5.3.1 Defining group names

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

5.3.1.1 The name= parameter

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

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

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

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

5.3.1.2 The `write=` parameter

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

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

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

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

5.3.2 Defining the properties of a group

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

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

5.3.2.1 The `source=` parameter

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

5.3.2.2 Defining a group identifier

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

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

By default each group's identifier is relatively addressable.

5.3.2.2.1 The `sign=` parameter

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

5.3.2.2.2 The `identification=` parameter

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

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

5.3.3 Deciding which elements to include

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

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

By default when the system is saving elements it ignores the order in which they appeared in the input data. In each group elements are held in the order in which the element names were made known to $\kappa\lambda\epsilon\iota\omega$.

5.3.3.1 The `position=` parameter

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

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

5.3.3.2 The `also=` parameter

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

5.3.3.3 The `guaranteed=` parameter

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

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

5.3.3.4 The `order=` parameter

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

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

5.3.3.5 The `read=` parameter

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

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

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

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

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

5.3.4 Deciding which groups to include

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

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

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

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

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

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

5.3.4.1 The `part=` parameter

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

5.3.4.2 The `always=` parameter

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

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

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

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

5.3.4.3 The `only=` parameter

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

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

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

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

5.3.4.4 The `arbitrary=` parameter

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

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

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

5.3.4.5 The `sequence=` parameter

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

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

5.3.5 Using generic group identifiers

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

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

5.3.5.1 The `prefix=` parameter

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

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

5.3.5.2 The `suffix=` parameter

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

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

5.3.6 Behaviour in networks

The following applies by default:

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

5.3.6.1 The `relation=` parameter

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

5.3.7 Defining complex structures

5.3.7.1 The `start=` parameter

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

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

This parameter is discussed in detail in section 5.8.1.

5.3.7.2 The `alias=` parameter

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

This parameter is discussed in detail in section 5.8.2.

5.3.7.3 The `usage=` parameter

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

This parameter is discussed in detail in section 5.8.3.

5.3.7.4

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

5.3.7.5 The `more=` parameter

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

This parameter is described in detail in section 5.8.3.3.

5.3.7.6 The `substitution=` parameter

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

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

5.3.7.7 The `cumulate=` parameter

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

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

5.3.8 Examples

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

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

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

5.4 The `element` directive

The general form of the `element` directive is:

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

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

5.4.1 Defining element names

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

5.4.1.1 The `name=` parameter

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

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

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

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

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

5.4.1.2 The `write=` parameter

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

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

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

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

5.4.2 Defining an element's characteristics

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

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

5.4.2.1 The `source=` parameter

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

5.4.3 Selecting the preferred data type

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

5.4.3.1 The `type=` parameter

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

5.4.3.2 The `first=` parameter

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

5.4.3.3 The `second=` parameter

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

5.4.4 Assigning aspects and entries

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

5.4.4.1 The `order=` parameter

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

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

5.4.5 Using an element as a group identifier

5.4.5.1 The `identification=` parameter

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

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

5.4.5.2 The `only=` parameter

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

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

5.4.5.3 The `cumulate=` parameter

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

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

5.4.6 Using generic element names

Any element identifier mentioned in an `element` directive's `name=` parameter is regarded as part of the description of this database's logical structure. This means that $\kappa\lambda\epsilon\iota\omega$ assumes that the identifier actually appears in this form in the input data.

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

5.4.6.1 The `prefix=` parameter

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

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

5.4.6.2 The `suffix=` parameter

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

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

5.4.7 Parameters for specific data types

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

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

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

5.4.7.1 The `text=` parameter

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

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

5.4.7.2 The `date=` parameter

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

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

5.4.7.3 The `number=` parameter

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

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

5.4.7.4 The `category=` parameter

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

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

5.4.7.5 The `relation=` parameter

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

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

5.4.7.6 The `location=` parameter

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

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

5.4.7.7 The `image=` parameter

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

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

5.4.8 Defining complex structures

5.4.8.1 The `alias=` parameter

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

This parameter is discussed in detail in section 5.9.2.

5.4.9 Examples

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

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

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

5.5 The `limit` directive

The general form of the `limit` directive is:

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

The directive allows you to redefine the symbols for the "end of input data; terminate $\kappa\lambda\epsilon\iota\omega$ program" or "end of input data; continue $\kappa\lambda\epsilon\iota\omega$ program" (cf. section 6.1).

5.5.1 The `name=` parameter

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

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

5.5.2 The `type=` parameter

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

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

5.6 The `series` directive

The general form of the `series` directive is:

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

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

5.6.1 The `name=` parameter

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

5.6.2 The `position=` parameter

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

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

5.6.3 The `also=` parameter

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

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

5.7 The `exit` directive

The general form of the `exit` directive is:

NAME=<filename> (MUST be specified!)

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

5.7.1 The `name=` parameter

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

5.8 Example for sections 5.2 to 5.7

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

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

5.9 Information units and their names

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

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

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

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

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

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

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

For this reason $\kappa\lambda\epsilon\iota\omega$ provides you with a range of tools for drawing a sharper and clearer distinction between these three aspects of your database definition.

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

5.9.1 Several types of document in one database

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

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

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

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

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

would thus be interpreted by the system as:

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

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

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

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

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

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

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

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

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

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

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

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

The following fragment of a structure definition describes this condition:

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

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

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

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

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

In practice, this means that the task

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

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

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

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

Similarly, the task:

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

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

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

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

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

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

will cause all four types of document to print out.

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

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

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

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

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

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

is also quite easy to follow.

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

It results in something like this (for example):

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

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

So in fact the following model would be more suitable:

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

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

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

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

5.9.3.1 The usage= parameter

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

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

5.9.3.2 `connection` definitions

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

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

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

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

5.9.3.3 The `more=` parameter

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

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

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

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

5.9.3.4 Closing remarks

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

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

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