8. Working with individual databases

In order to work with a database, you must first specify:

- which database you want to work with,
- which parts of the database the system should select for processing and
- what you want the system to do with the information you extract.

8.1 General constituents of commands

In addition to the parameter values described in section 2.1.1.3.1.1, which apply to all system components, the parts of the system documented in this chapter accept a series of additional parameter values which are used to extract the set of information you want to process from a database.

8.1.1 Paths

Information you want to extract from a database for a particular purpose is described by a *path*. A path consists of a series of specifications which $\kappa\lambda\epsilon\iota\omega$ uses to address a sequence of entries. Each entry in turn is subjected to the required processing. When the system reaches the last entry, it proceeds down the next path specified in the command. Once it reaches the last path the task is terminated, and the extracted information as a whole is subjected to a final post-processing operation – e.g. a sort – if appropriate.

Paths consist of the following:

- group identifiers,
- group functions,
- element identifiers,
- element functions,
- expressions,
- network transitions and
- conditions.

8.1.1.1 Group identifiers

A group identifier consists of the name of a group, which is included in the data structure of the relevant database. It represents the set of all those groups which can be addressed by this name. The system recognises a group identifier in a parameter by the fact that

- it appears at the head of a path or
- a data signal character 2 appears in front of it.

8.1.1.1.1 Absolute group identifiers

A group identifier is absolute if it appears at the head of a path down which the system can proceed without being dependent on any other path or paths. The following are examples of absolute group identifiers:

- `child`
- `/child`

The system interprets both identifiers as the set of all groups called "child" contained in a given database.

8.1.1.1.2 Relative group identifiers

A group identifier is relative if it is used in a path which is introduced by another group identifier or which depends on another path in the context of a given task.

For example, the group identifier "child" in the following paths is relative in each case:

- `person/child`
- `marriage/child`

The system interprets the first of these identifiers as the set of all those groups called "child" which depend directly on a group called "person" in a given database; the second as the set of all those groups called "child" which depend directly on the group "marriage".

8.1.1.2 Group functions

A group function is a built-in function which occupies the same sort of position in a path in which you might also expect to find a group identifier. Its value consists of a set of groups defined by its arguments.

8.1.1.2.1 Group function arguments

A group function's arguments define a set of groups. Where group functions are nested, the system applies each outer function in turn to every element in the set represented by the inner function.

8.1.1.2.1.1 Keywords

Keywords in group functions can always be written in abbreviated form, consisting simply of the first character in the keyword.

8.1.1.2.1.2 Group specifications

A group specification consists of a group identifier or group function.

8.1.1.2.1.3 Number ranges

A number range consists of a list of comma-delimited interval specifications.

An interval specification comprises:

- a positive whole number (integer),
- two positive integers linked by a hyphen,
- an integer preceded by a hyphen, which the system interprets as "1-n", or
- an integer followed by a hyphen, which the system interprets as "n-1000000".

If a number range is missing from a function which accepts number ranges, the system substitutes the number range "1".

A number falls within a number range if it corresponds to a number contained in that range or occurs between two of the numbers describing an interval specification.

Intervals in a number range do not have to appear in any particular order, and may overlap.

8.1.1.2.1.4 Label ranges

A label range consists of a list of comma-delimited label intervals.

A label interval comprises:

- a constant which the system interprets as a group identification,
- two constants joined by a hyphen,
- a constant preceded by a hyphen, which the system interprets as $< first\ group\ identification\ to\ appear > - < constant >$ or
- a constant followed by a hyphen, which the system interprets as $< constant > - < last\ group\ identification\ to\ appear >$.

If a label range is missing from a function which requires such a range, this is interpreted as an error.

A group is included in a label range if it has a group identification identical to one of the identifications in the range or if it is logically positioned between two groups with group identifications which form a label interval.

Intervals in a label range do not have to appear in any particular order and may overlap.

8.1.1.2.1.5 Numbers

A number is a positive whole number (integer).

8.1.1.2.1.6 Element specifications

An element specification consists of an element identifier or element function.

8.1.1.2.2 Individual group functions

Unless you specify an argument in the following functions, no character other than a number (any number) of spaces should appear between the enclosing parentheses.

Any surplus arguments are treated as errors.

8.1.1.2.2.1 The group function `query[]`

The group function `query[]` is written in one of two ways:

`query[]`

or

`query[always | temporary]`

The system interprets this function as the last group reached by the previous `query` command. The last group reached by `query` is the group at the end of the path in front of the *first* equals sign which introduces a condition. The system takes no account of groups preceded by `and` or `or` or which form comparands.

- If the function is invoked without an argument or with `always`, *all* following paths are relative to the group expressed by `query[]`.
- If it is invoked with `temporary` it instructs the system that the *path immediately adjoining it* is relative to this group, but that it should restore the status prevailing prior to `query[]` for all subsequent paths.
  We recommend that you use `temporary` in built-in functions by preference. Built-in functions in which `query[]` together with `always` would have known and undesirable effects (such as `continue[]`) always imply this argument in any case.

8.1.1.2.2.2 The group function `sign[]`

The group function `sign[]` is written as follows:

`sign[group specification, label range]`

The system interprets it as the set of groups defined by *group specification* located within the *label range*.

8.1.1.2.2.3 The group function `catalogue[]`

The group function `catalogue[]` is written as follows:

```
catalogue[name,complete | start | limit | algorithm,
        element specification | constant]
```

This function accepts arguments which can only be understood in the context of processing catalogues. (cf. section 8.3.2 of this manual for more information on this topic). *Name* specifies the name of a catalogue generated by a `catalogue[]` command.

The second argument specifies which access path the system should choose inside the catalogue:

- `complete` instructs the system to choose the access path which allows it to search for a term as a whole.
- `start` instructs the system to choose the access path which allows it to search for a term beginning with a known character string.
- `limit` instructs the system to choose the access path which allows it to search for a term ending in a known character string.

- `algorithm` instructs the system to choose the access path which demands that the search term should first be converted into a derived form (a lemma, a soundex code or a skeletonising code). You should not (and cannot) tell the system which procedure to use for this purpose: the system ensures that it uses the same procedure for this derivation as it used to implement this access path when the catalogue was first created.

If the second argument is missing, the system assumes that it is `complete`: if you want to specify the third argument explicitly, however, you must ensure that the second argument is written.

If the third argument is missing, the system interprets the function as the set of all groups which caused at least *one* entry to be made in the *name* catalogue. Each group appears in this set the same number of times as it caused entries to be made in the *name* catalogue. These groups are processed in the same order as the entries in the specified access path for the catalogue *name*.

If the third argument is a *constant* the system interprets the function as the set of all groups which caused at least one entry to be made in the *name* catalogue, where the entry can be reached via this *constant* in the specified access path, i.e. where the entry is identical to, starts or ends with this constant, or can be brought into the same derivable form as this constant.

The system examines the third argument to see whether it can be used as a search pattern in the sense defined in section 7.3.1.15.1.2 once the outer delimiting characters have been stripped off. If it can, then the function is defined as the set of all groups which can be reached via this pattern. It is important to remember that the third argument as a whole should be enclosed by a pair of delimiters. Thus in order to access the set of all those groups in which contexts containing the word `church` as well as the word `school` appear, you would need to make the following function call:

```
catalogue[example,complete,"'school' and 'church'"]
```

In this case, the system ignores the second argument, because the access path to be used for the pattern's separate components is evident from the pattern definition. In the interests of consistency, however, you must specify a valid keyword for the second argument, even in this formulation.

If the third argument is an *element specification*, the system interprets the function as the set of all groups which have caused at least one entry to be made in the *name* catalogue, where the entry can be reached via an element in the set of entries of the `text` data type described by the *element specification* in the specified access path, i.e. where the entry is identical to such an entry, or begins or ends with it, or can brought into the same derivable form as this entry.

Each group appears in this set the same number of times as it caused entries of this type to be made in the *name* catalogue. The groups are processed in the same order as the entries in the *name* catalogue.

The only place in a path where any form of this function should appear is at a position where an absolute group name would be valid, i.e. as the first component of a path which does not depend on any other path.

8.1.1.2.2.4 The group function `keyword[]`

The group function `keyword[]` is written as follows:

`keyword[name-1,name-2]`

This function accepts non-standard arguments which can only be understood in the context of catalogue processing (cf. section 8.3.2 of this manual) and interactive information retrieval (cf. Chapter 7 in the Tutorial volume and section 8 of volume B10 of the "Halbgraue Reihe". *Name-1* specifies the name of a catalogue generated by a `catalogue` command. *Name-2* specifies a list of references stored for later use with the help of the `save` option on the main menu for interactive processing of catalogues (cf. section 8.1.2.3 of volume B10 of the "Halbgraue Reihe").

The system interprets this function as the set of all groups which have contributed at least one reference to the list of references *name-2* contained in the catalogue *name-1*. Each group appears in the set the same number of times as it contributed references to the list of references *name-2*. The groups are processed in the same order as the entries in the catalogue *name-1*.

The only place in a path where this function should appear is at a position where an absolute group name would be valid, i.e. as the first component of a path which does not depend on any other path.

8.1.1.2.2.5 The group function `brother[]`

The group function `brother[]` is written as follows:

`brother[group specification,left | right,number range]`

The system interprets it as the set of groups

- with the same name as an element in the set expressed by *group specification* (henceforth called *group* for short),
- (if the second argument is `left`) which in the same group as the one containing the *group*, have lower ordinal numbers than the *group* (i.e. appear "in front of" the latter),
- (if the second argument is `right`) which in the same group as the one containing the *group*, have higher ordinal numbers than the *group* (i.e. appear "after" the latter) and
- which are positioned at a distance from the *group* which is still within the *number range*.

8.1.1.2.2.6 The group function `father[]`

The group function `father[]` is written as follows:

`father[element specification]`

The system interprets it as the set of groups containing the elements expressed by *element specification*.

8.1.1.2.2.7 The group function `same[]`

The group function `same[]` is written as follows:

`same[group specification,left | right,number range]`

The system interprets it as the set of groups which

- have the same name as an element in the set expressed by *group specification* (henceforth referred to as *group*),
- (if the second argument is `left`), appear “in front of” the *group* in the database,
- (if the second argument is `right`), appear “after” the *group* in the database and
- are located at a distance from the *group* which is still within the *number range*.

8.1.1.2.2.8 The group function `part[]`

The group function `part[]` is written as follows:

`part[group specification,number range]`

The system interprets it as the set of groups which form an element in the set defined by *group specification* and have an ordinal number within the *number range*.

8.1.1.2.2.9 The group function `forward[]`

The group function `forward[]` is written as follows:

`forward[group specification,number range]`

The system interprets it as the set of groups which are logically immediately subordinated to the elements in the set defined by *group specification* (henceforth referred to as *group* for short) and appear at positions in the sequence of types of groups contained in the *group* which are within the *number range*.

8.1.1.2.2.10 The group function `total[]`

The group function `total[]` is written as follows:

`total[group specification]`

The system interprets it as the set of groups which are logically directly or indirectly subordinated to the elements in the set defined by *group specification*.

8.1.1.2.2.11 The group function `root[]`

The group function `root[]` is written as follows:

`root[number]`

This is defined as the group on logical level *number* which logically contains the group which the system last activated. (Documents appear on logical level zero; groups which are logically immediately subordinated to them on logical level one, etc.). A group located on a logical level with an ordinal number equal to or less than *number* is regarded as its own `root[]`.

8.1.1.2.2.12 The group function `back[]`

The group function `back[]` is written in one of two ways:

`back[number]`

or

`back[group identifier]`

The system interprets the first of these two forms as the group which appears *number* logical levels above the last group to be activated.

If the last group to be activated is less than *number* logical levels from its parent document, the system does not select a group for processing.

The system interprets the second of these two forms as the first group called *group identifier* which logically appears "above" the last group to be activated. If no such group is located anywhere in the hierarchy up to and including the document which contains the present group, the system does not select a group for processing.

This interpretation of the parameter values is best illustrated with the help of a diagram:

If we assume the following data structure:

```
Document
    |
    |
    | → Group A
    |    |
    |    |
    |    | → Group B
    |    |    |
    |    |    |
    |    |    | → Group C
```

In the current group	corresponds	to group
Group C	back[2]	Group A
Group C	back[Group A]	Group A
Group B	back[2]	document
Group B	back[Group A]	Group A
Group A	back[2]	not defined
Group A	back[Group A]	not defined

8.1.1.2.2.13 The group function `continue[]`

The group function `continue[]` is written as follows:

```
continue[group specification,element specification,condition,
        number,self]
```

The system interprets it a set of groups which is derived dynamically from the data. For a better understanding of how this works, we must describe its behaviour in a path. In the path `path_section-1 continue[...]  path_section-2`, `continue[]` is defined by the set of groups reached by the following operation:

a) Once *path_section-1* has been processed, `continue` proceeds along the path defined by *group specification* and then processes *path_section-2*.
b) Now the system attempts to find the *element specification* of the `relation` data type in the group which it has reached.
c) If you specified the third argument, the system determines whether the entries made accessible by the argument satisfy the *condition*, which must be a relational search pattern.
d) Where an entry does fulfil this condition, the system sets up a transition to the group to which the entry refers. (If you did not specify a *condition*, the system does this for all entries).
e) Finally, the system processes *path_section-2* again, and the process proceeds to step b.

In the normal way, the system terminates this process if step b) or c) fails to unearth an element which can be traced any further. However, since it is quite possible to formulate the individual paths concerned in such a way that a group refers back to itself by various roundabout ways, we have included a number of safeguards. By default, the following rules apply:

- The system stops processing step b) if has already been successful 10 times.
- The system stops processing step d) if it returns you to a group which you reached earlier while the system was processing steps a) or d).

You can control both default settings as follows:

- If you specify *number*, the system refrains from processing step b) once it has been successful *number* times. ("Maximum depth of recursion")
- If you define `self` as the fifth argument, the system does not check to see whether it has already processed one and the same group. You can only specify the fifth argument if you also specify the fourth; in this case, we would strongly advise you to set the latter to a low value.

8.1.1.2.2.14 The group function `last[]`

The group function `last[]` is written as follows:

```
last[relative group specification]
```

It is defined as the set of the last occurences of the last part of the group specification within the previous part of that specification.

That is:

```
last[group/subgroup]
```

will for each `group` in the database select the last occurence of `subgroup` within this occurence of group. Or, to give a more meaningful example:

```
last[marriage/child]
```

will access all the last born `children` of all `marriages`, provided they occur within the respective marriages in the order of their births.

All considerations of command relativity apply. That is, the example

```
query name=example;part=marriage
write part=last[child]:each[]
```

will work exactly the same as

```
query name=example
write part=last[marriage/child]:each[]
```

8.1.1.2.2.15 The group function `selection[]`

The group function `selection[]` is written as follows:

```
selection[first | last,relative group specification,
        element specification,condition]
```

It is defined as the set of the *first* or *last* occurences of the last part of the group specification within the previous part of that specification which contains an *element specification* fulfilling *condition*. All considerations of command relativity apply. That is

```
query name=example;part=marriage
write part=selection[last,child,:birthdate,
        before "1 january 1700"]:each[]
```

will display each element of the *last* `child` in each `marriage` with a `birthdate` prior to the 1^{st} of January 1700;

```
query name=example
write part=selection[first,marriage/child,:birthdate,
        after "1 january 1700"]:each[]
```

will display each element of the *first* `child` in each `marriage` with a `birthdate` after the 1^{st} of January 1700.

8.1.1.2.2.16 The group function `none[]`

The group function `none[]` is written as follows:

```
none[group specification,relative element specification,condition]
```

It is defined as the set of all groups out of *group specification* which do *not* contain any *relative element specification* that fulfills *condition*.

That is

```
write part=none[marriage,child:birthdate,before "1.1.1600"]:each[]
```

will display information about all `marriages` where for *all* `children` the `birthdate` is either unknown or later than the 1^{st} of january 1600. Each such `marriage` is displayed once.

8.1.1.3 Element identifiers

An element identifier consists of the name of an element, which is included in the data structure of the relevant database. It represents the set of all those elements which can be addressed by this name. The system recognises an element in a parameter by the fact that it appears after a data signal character 9.

8.1.1.3.1 Absolute element identifiers

An element identifier is absolute if it appears at the head of the name of a path down which the system can proceed without being dependent on any other path or paths. The following is an example of an absolute element identifier:

- `:surname`

The system interprets this identifier as the set of all elements called "surname" contained in a given database.

8.1.1.3.2 Relative element identifiers

An element identifier is relative if it is used in a path which is introduced by a group identifier or which depends on another path in the context of a given task. For example, the element identifier "surname" in the following path names is relative in each case:

- `person/child:surname`
- `marriage/child:surname`

The system interprets the first of these identifiers as the set of all elements called "surname" which depend on a group called "child" in a given database, which in turn depends on a group called"person"; it interprets the second identifier as the set of all elements called "surname" which depend on a group called "child" in a given database, which in turn depends directly on a group called "marriage".

8.1.1.4 Element functions

An element function is an built-in function which occupies the same sort of position in a path in which you might also expect to find an element identifier. Its value consists of a set of elements defined by its arguments.

8.1.1.4.1 Element function arguments

Element function arguments define a set of elements, which – depending on the context – the system may also interpret as a set of the entries contained in their basic information or one of their other aspects, or as a set of entries calculated from other components relating to them. Where element functions are nested, the system applies each outer function in turn to every element in the set represented by the inner function.

8.1.1.4.1.1 Keywords

Keywords in element functions can always be written in abbreviated form, consisting simply of the first character in the keyword.

8.1.1.4.1.2 Element specifications

An element specification consists of an element identifier or element function.

8.1.1.4.1.3 Number ranges

A number range consists of a list of comma-delimited interval specifications.

An interval specification comprises:

- a positive whole number (integer),
- two positive integers linked by a hyphen,
- an integer preceded by a hyphen, which the system interprets as "1-n", or
- an integer followed by a hyphen, which the system interprets as "n-1000000".

If a number range is missing from a function which accepts number ranges, the system substitutes the number range "1".

A number falls within a number range if it corresponds to a number contained in that range or occurs between two of the numbers describing an interval specification.

Intervals in a number range do not have to appear in any particular order and may overlap.

8.1.1.4.1.4 Group specifications

A group specification consists of a group identifier or group function.

8.1.1.4.1.5 Conversion specifications

Conversion specifications tell the system which rules to use in order to convert an entry of the `text` data type into a required target data type before processing it. This language element has not yet been implemented.

8.1.1.4.1.6 Constants

A constant is always a string of characters framed by delimiters, as described in section 2.1.1.3.1.1.1.

8.1.1.4.1.7 Definition names

Definition names are the names of logical objects, defined by the `item` command (cf. section 7.3.1.1).

8.1.1.4.1.8 Names of variables

Names of variables are names which select one of several options in a logical object, e.g. a codebook.

8.1.1.4.2 Individual element functions

Unless you specify an argument in the following functions, no characters other than a number (any number) of spaces are allowed to appear between the enclosing parentheses.

Any surplus arguments are treated as errors.

8.1.1.4.2.1 The element function `name[]`

The element function `name[]` is written in one of the two following ways:

`name[]`

or

`name[element specification]`

- In the first case, it is defined as an entry of the `text` data type containing the name of the last group to be activated.
- In the second case, it is defined as the set of all entries of the `text` data type, which can be calculated from the names of the set of elements defined by *element specification*.

8.1.1.4.2.2 The element function `sign[]`

The element function `sign[]` is written as follows:

`sign[]`

It is defined as an entry of the `text` data type containing the group identification of the last group to be activated.

8.1.1.4.2.3 The element function `usage[]`

The element function `usage[]` is written as follows:

`usage[element specification]`

It is defined as a set of entries of the `text` data type which represents the data type of all entries defined by the *element specification*.

8.1.1.4.2.4 The element function `brother[]`

The element function `brother[]` is written as follows:

`brother[element specification,left | right,number range]`

The system interprets it as the set of entries which

- appear in the same element as the last entry to be activated by the *element specification* (and henceforth referred to as *entry*),
- (if the second argument is `left`), appears in the same element “in front of” the entry,
- (if the second argument is `right`), appears in the same element “after” the entry and
- is positioned at distance away from the *entry* which is still within the *number range*.

8.1.1.4.2.5 The element function `cousin[]`
The element function `cousin[]` is written as follows:

`cousin[element specification,first | comment | original]`

It is defined as the set of entries which are located in the aspect defined by the *keyword* of the set of entries defined by the *element specification*.

At the present time this element function can only be used for element functions where the `element` directive has been assigned the parameter `order=multiple`. As a rule, it is used to process databases in which the equivalent entry from another aspect is to be processed for a given entry. This is necessary, for example, if you choose an input convention where an element's basic information includes the names of goods and the associated original text field contains the price quoted in the source, as demonstrated below:

```
commodity=name-1%2.1;name-2%1.3;name-3%2.5;name-4%3.2
```

In this example, the following task would cause the system to print out the prices for a commodity forming the object of a search:

```
query part=:commodity="identifier (forming object of search)"
write part=:cousin[:query[],original]
```

8.1.1.4.2.6 The element function `status[]`
The element function `status[]` is written as follows:

`status[element specification]`

It is defined as a set of entries of the `text` data type which represents the status of all entries defined by the *element specification*.

8.1.1.4.2.7 The element function `first[]`
The element function `first[]` is written as follows:

`first[element specification]`

It is defined by the set of all entries in the "basic information" aspect of the set of all elements defined by the *element specification*.

8.1.1.4.2.8 The element function `original[]`
The element function `original[]` is written as follows:

`original[element specification]`

It is defined by the set of all entries in the "original text" aspect of the set of all elements defined by the *element specification*.

8.1.1.4.2.9 The element function `comment[]`
The element function `comment[]` is written as follows:

`comment[element specification]`

It is defined by the set of all entries in the "comment" aspect of the set of all elements defined by the *element specification*.

8.1.1.4.2.10 The element function `same[]`

The element function `same[]` is written as follows:

`same[element specification, left | right, number range]`

The system interprets it as the set of elements which

- have the same name as an element in the set expressed by *element specification* (henceforth referred to as *element*),
- (if the second argument is `left`), appear "in front of" the *element* in the database,
- (if the second argument is `right`), appear "after" the *element* and
- are located at distance from the *element* which is still within the *number range*.

8.1.1.4.2.11 The element function `part[]`

The element function `part[]` is written as follows:

`part[element specification,number range]`

It is defined as the set of all entries defined by *element specification* which appear in their respective aspects in positions within the sequence of entries in the *number range*.

8.1.1.4.2.12 The element function `offspring[]`

The element function `offspring[]` is written as follows:

`offspring[group specification,number range]`

The system interprets it as the set of elements which are contained in the elements of the set defined by the *group specification* (henceforth referred to as *group*) and appear in the sequence of types of element contained in the *group*, in positions within the *number range*.

8.1.1.4.2.13 The element function `each[]`

The element function `each[]` is written as follows:

`each[]`

It is defined as the set of all elements contained in the last group to be activated.

8.1.1.4.2.14 The element function `total[]`

The element function `total[]` is written as follows:

`total[]`

It is defined as the set of all elements which are contained in the last group to be activated and in all groups logically subordinated to the latter.

8.1.1.4.2.15 The element function `core[]`

The element function `core[]` is written as follows:

`core[element specification]`

It is defined by the set of core information items in the entries defined by *element specification*.

8.1.1.4.2.16 The element function `text[]`
The element function `text[]` is written as follows:

```
text[element specification,number | date | category | relation |
       location | image | others]
```

- If only the first argument is specified, this function is defined as the set of all entries of the `text` data type defined by *element specification*.
- If the second argument is `others`, this function is defined as the result of converting each entry within *element specification* to the data type `text`.
- If the second argument is any of the other keywords, this function is the set of the entries of the data type `text` which is the result of converting all entries of the selected data type within *element specification* to `text`.

8.1.1.4.2.17 The element function `date[]`
The element function `date[]` is written as follows:

```
date[element specification,text,definition name]
```

- If only the first argument is specified, this function is defined as the set of all entries of the `date` data type defined by *element specification*.
- Otherwise the function is defined as the set of all entries of the `date` data type which is the result of converting all entries of data type `text` within *element specification* with the help of *definition name*, which must be the name of a `date` definition.

8.1.1.4.2.18 The element function `number[]`
The element function `number[]` is written as follows:

```
number[element specification,text,definition name]
```

- If only the first argument is specified, this function is defined as the set of all entries of the `number` data type defined by *element specification*.
- Otherwise the function is defined as the set of all entries of the `number` data type which is the result of converting all entries of data type `text` within *element specification* with the help of *definition name*, which must be the name of a `number` definition.

8.1.1.4.2.19 The element function `category[]`
The element function `category[]` is written as follows:

```
category[element specification,text,definition name]
```

- If only the first argument is specified, this function is defined as the set of all entries of the `category` data type defined by *element specification*.
- Otherwise the function is defined as the set of all entries of the `category` data type which is the result of converting all entries of data type `text` within *element specification* with the help of *definition name*, which must be the name of a `category` definition.

8.1.1.4.2.20 The element function `relation[]`
The element function `relation[]` is written as follows:

```
relation[element specification]
```

This function is defined as the set of all entries of the `relation` data type defined by *element specification*.

8.1.1.4.2.21 The element function `location[]`

The element function `location[]` is written as follows:

`location[element specification,text,definition name]`

- If only the first argument is specified, this function is defined as the set of all entries of the `location` data type defined by *element specification*.
- Otherwise the function is defined as the set of all entries of the `location` data type which is the result of converting all entries of data type `text` within *element specification* with the help of *definition name*, which must be the name of a `location` definition.

8.1.1.4.2.22 The element function `image[]`

The element function `image[]` is written as follows:

`image[element specification,text,definition name]`

- If only the first argument is specified, this function is defined as the set of all entries of the `image` data type defined by *element specification*.
- Otherwise the function is defined as the set of all entries of the `image` data type which is the result of converting all entries of data type `text` within *element specification* with the help of *definition name*, which must be the name of an `image` definition.

8.1.1.4.2.23 The element function `lines[]`

The element function `lines[]` is written as follows:

`lines[number]`

It is defined as an entry of the `text` data type which causes the system to advance by *number* lines when you are using commands such as `write` or `index` to transfer data to an output medium or device.

8.1.1.4.2.24 The element function `page[]`

The element function `page[]` is written as follows:

`page[constant]`

It is defined as an entry of the `text` data type which causes the system to advance by one page, at the top of which *constant* appears on a separate line, when you are using commands such as `write` or `index` to transfer data to an output medium or device.

8.1.1.4.2.25 The element function `cumulation[]`
The element function `cumulation[]` is written as follows:

```
cumulation[element specification | :query[],
    small | large | mean | deviation | number | total,
    without | deviation | generation, without | deviation | generation]
```

It is defined as a entry of the `number` data type, which represents a parameter for the statistical distribution of all entries of the `number` data type covered by the *element specification*.

- If `:query[]` appears as the first argument the system invokes the element specification in the last `cumulation[]` call. If `cumulation[]` is invoked with `:query[]` before a call with *element specification* was made, the system displays an error message.
- You can use the keywords `small`, `large`, `mean`, `deviation`, `number` and `total` to select the minimum, maximum, mean value, standard deviation, number of cases included and the sum of the corresponding distribution respectively.
- You may omit the third and fourth parameters; if you leave them in, they must be consistent. The optional keywords for these parameters have the following effects:
  - `without` instructs the system to exclude all `number` entries expressed as an interval between a minimum and a maximum value from the calculation of statistical classification numbers. (This is how the system proceeds by default.)
  - `deviation`, on the other hand, instructs the system to calculate the mean value from minimum and maximum in such events, and to weight its inclusion in the statistical classification numbers using the following expression:

$$1 - \frac{(maximum - minimum)}{maximum}$$

.

  - `generation` only has consequences if calculations within a network are involved. It evaluates the distances of the individual entries included in the calculation from the group in the database in which the path forming the *element specification* started. In fact, it divides each entry by the square of its distance along the path. This means that an entry appearing in the same group as the original information item is divided by 1; an entry which was reached by making *one* transition via a entry of the `relation` data type is divided by 2; an entry which could only be reached by making *two* such transitions is divided by 4, etc. etc. This option was implemented primarily for the purpose of weighting statistics within genealogical networks.

8.1.1.4.2.26 The element function `codebook[]`

The element function `codebook[]` is written as follows:

```
codebook[element specification-1 | constant | :query[],
        name of variable | element specification-2, definition name]
```

If *name of variable* or *element specification-2* do not contain any further additions, this function is defined as follows:

The set of entries of the `number` data type which are created when the system searches out each entry in turn which can be converted into the `text` data type and which corresponds to *element specification-1* or *constant*, as an identifier in the codebook called *definition name*, and then specifies the numerical value of one entry from each variable which can be derived from *variable name* or the contents – converted into the `text` data type – of each entry in *element specification-2*.

If the *variable name* is followed by one of the two keywords `number` or `text` after a *data signal character 2* (slash) or if *element specification-2* is followed by one of these keywords after the same data signal character, the following definitions apply:

For `number`: the same as if you had not specified a keyword.

For `text`: the set of entries of the `text` data type resulting from the following operation: the system starts by searching out each entry in turn which can be converted into the `text` data type and which corresponds to *element specification-1* or *constant*, as an identifier in the codebook called *definition name* and then retains the numerical value of one entry from each variable which can be derived from *variable name* or the contents – converted into the `text` data type – of each entry in *element specification-2*. Finally, the system searches for the tag corresponding to each of these numerical values in the codebook called *definition name*.

If `:query[]` appears as the first argument, the system attempts to find additional variables in the most recently read specifications for the same codebook. In this case, any attempt to specify the *definition name* again is treated as an error; it is also an error to invoke this function using `:query[]` unless it has first been called by *element specification-1* or *constant*.

8.1.1.4.2.27 The element function `expectation[]`

The element function `expectation[]` is written as follows:

```
expectation[element specification-1 | constant-1,
        element specification-2 | constant-2,
        element specification-3 | constant-3, definition name,
        name of variable | element specification-4, each[] | partial]
```

This built-in function [1] is used to estimate the gap between two points in time about which you only have inadequate information. It is intended primarily as a way of estimating the age of a person at a given moment in time. Because the function can be explained most plausibly with the help of this example, we use a version of it below. However it

[1] This built-in function was developed in collaboration with the Anthropological Institute at Göttingen University, and was defined by Dr. Eckart Voland.

should be noted that this function can also be used with other sequences of interrelated `date` data.

The function always provides you with an entry of the `number` data type which expresses the probable distance in time between *element specification-1* (or *constant-1*) and *element specification-2* (or *constant-2*) in years.

In the following description of the way the function operates, we assume that:

- *element specification-1* or *constant-1* represent a given individual's date of birth, and that
- *element specification-2* or *constant-2* represent the date on which the same individual died.
- *Element specification 3* or *constant 3* represent the date of a "target event", e.g. the date on which this individual was married.
- *Definition name* refers to a codebook which lists the life expectancies of a population classified by sex.
- *Variable name* or *element specification-4* represents an individual's sex. The function calculates the probable age of the individual at the time of the target event: if the date on which the individual was married is unknown, the system prepares a "missing value"; it also prepares a "missing value" if the date of death precedes the target date.

If you only know a *terminus post* or *ante quem* for the target date, the system uses the term you know; if you know both terms for a given interval the system uses the interval's midpoint.

The system completes the two other dates using the following procedures for separate combinations of incomplete information:

date of birth	date of death: exactly known	only *terminus ante quem* known	only *terminus post quem* known	both terms known	date unknown
exactly known	1	2	3	8	4
only *terminus ante quem* known	5	6	7	9	6
only *terminus post quem* known	5	7	6	9	6
both terms known	8	2	3	8	4
date unknown	6	6	6	6	6

1) No missing information about dates.
2) The system assumes that the date of death is the lower value of:
    - the indivividual's life expectancy at birth according to sex,
    - the *terminus ante quem* of the individual's date of death.
3) The system assumes that the individual's date of death is equivalent to the life ex-

pectancy according to sex for the age attained by the *terminus post quem* of the individual's date of death.

4) The system assumes that the individual's date of death is equivalent to the individual's life expectancy at birth according to sex.
5) The system uses the known term as the date of birth.
6) The system is unable to supply the missing information for these combinations.
7) In both cases, the system regards the known term as an exact date.
8) The system assumes that the imprecisely known date is the midpoint between both terms.
9) Where the system only knows one term for a date, it uses the latter; if it knows both terms it uses the midpoint between them.

The function's last argument determines the type of result reproduced as a `number` entry:

- `complete` instructs the system to use the exact result of the numerical calculation as an entry.
- `partial` instructs the system to round up the exact result to the nearest whole year.

8.1.1.4.2.28 The element function `order[]`

The element function `order[]` is written as follows:

```
order[]
```

It is defined as an entry of the `number` data type, which represents the ordinal number of the relevant group.

8.1.1.4.2.29 The element function `days[]`

The element function `days[]` is written as follows:

```
days[element specification]
```

It is defined as a set of entries of the `number` data type, which represent the day specifications for all entries of the `date` data type contained in the *element specification*. The system uses a value of zero for entries in which an interval of time is expressed as calendar dates from two different months.

8.1.1.4.2.30 The element function `month[]`

The element function `month[]` is written as follows:

```
month[element specification]
```

It is defined as a set of entries of the `number` data type which represent the month specifications for all entries of the `date` data type contained in the *element specification*. The system uses a value of zero for entries in which an interval of time is expressed as calendar dates from two different years.

8.1.1.4.2.31 The element function `year[]`

The element function `year[]` is written as follows:

```
year[element specification]
```

It is defined as the set of entries of the `number` data type which represent the year specifications for all entries of the `date` data type contained in the *element specification*.

8.1.1.4.2.32 The element function `weekday[]`

The element function `weekday[]` is written as follows:

`weekday[element specification]`

It is defined as a set of entries of the `text` data type which represent the weekdays for all entries of the `date` data type contained in the *element specification*. In terms of intervals of time, the weekday is treated as unknown. Entries generated by this built-in function are language-specific. In the English language version, they are reproduced as: `Unknown`, `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday` and `Sunday`. In the German language version, they are reproduced as: `Unbekannt`, `Montag`, `Dienstag`, `Mittwoch`, `Donnerstag`, `Freitag`, `Samstag` and `Sonntag`.

8.1.1.4.2.33 The element function `position[]`

The element function `position[]` is written as follows:

`position[]`

It is defined as an entry of the `number` data type, which reproduces the hierarchical distance of the relevant group from document level. The group opening the document has a hierarchical distance of zero; all groups directly contained in this group have a hierarchical distance of one, etc.

8.1.1.4.2.34 The element function `visibility[]`

The element function `visibility[]` is written as follows:

`visibility[]`

It is defined as an entry of the `number` data type, which reproduces the visibility (cf. section 4.3.3) of the given group as a numerical value.

8.1.1.4.2.35 The element function `form[]`

The element function `form[]` is written as follows:

`form[constant, text | date | number | category, definition name]`

It is defined as an entry appearing at the point in the database reached by the path being processed when the built-in function is activated. This entry's data type corresponds to the one specified by the second argument. The *constant* is converted into the relevant internal format according to the following rules:

- Using the logical object called in by default for the relevant data type if the third argument is missing.
- Using the logical object *definition name* from the class of logical objects corresponding to the data type. The system will generate an error message if such a logical object is not found in the local environment and cannot be loaded from the database being processed when this function is called.

Hint: *so you can prepare* `number` *constants which can be used with* `date` *specifications in joint expressions, the system makes the* `number`*-class logical object* `date` *available by default.* This logical object provides you with language-specific qualifiers with which you can formulate suitable constants.

In the German version, these are:

- `Jahr` or `J` with a numerical value of 365.25,
- `Monat` or `M` with a numerical value of 30.5,
- `Woche` or `W` with a numerical value of 7 and
- `Tag` or `T` with a numerical value of 1.

In the English version, they are:

- `Year` or `Y` with a numerical value of 365.25,
- `Month` or `M` with a numerical value of 30.5,
- `Week` or `W` with a numerical value of 7 and
- `Day` or `D` with a numerical value of 1.

8.1.1.4.2.36 The element function `collect[]`

The element function `collect[]` is written as follows:

```
collect[element specification]
```

It is defined as an entry of the data type required in the given context. This entry contains the specifications of all entries described by *element specification*.

The way the individual entries are combined depends on data type.

- `text` In this case the system strings the entries together, each separated from the next by data signal character 8 (by default: "`;`").
- `date` In this case, the system generates an interval of time, with the two exact dates or interval boundaries which are furthest apart in all relevant entries, as interval boundaries.
- `number` In this case, the system generates a numerical range, with the two exact numerical data or interval boundaries which are furthest apart in all relevant entries, as interval boundaries
- `category` In this case, the system generates an entry which contains the logical sum of all entries combined. Thus every character held for a particular category appears exactly once.

8.1.1.4.2.37 The element function `target[]`

The element function `target[]` is written as follows:

```
target[element specification,constant]
```

This built-in function is defined as the set of all network transitions from the entries in the *element specification* which contain references to a destination which fulfils the condition for the `relation` data type (cf. section 8.1.2.3.5.1) *constant*. At present, it is *not* possible to write another path instead of *constant*.

8.1.1.4.2.38 The element function `conversion[]`

The element function `conversion[]` is written as follows:

```
conversion[element specification,definition name]
```

It is defined as the set of entries of the `text` data type in the *element specification*, after they have been converted by the conversion definition *definition name*. Cf. section 7.3.1.7 for more information about this class of logical objects.

8.1.1.4.2.39 The element function `before[]`
The element function `before[]` is written as follows:

```
before[element specification]
```

It is defined by a set of entries of the `date` data type which are *always* exact dates.
- Where any `date` entries contained in *element specification* already represent exact dates the system incorporates these dates as they are.
- Where entries which contained intervals of time the system incorporates their respective *termini ante quem*.

8.1.1.4.2.40 The element function `after[]`
The element function `after[]` is written as follows:

```
after[element specification]
```

It is defined by a set of entries of the `date` data type which are *always* exact dates.
- Where any `date` entries contained in *element specification* already represent exact dates the system incorporates these dates as they are.
- Where entries which contained intervals of time the system incorporates their respective *termini ante quem*.

8.1.1.4.2.41 The element function `difference[]`
The element function `difference[]` is written as follows:

```
difference[element specification]
```

It is defined by a set of entries of the `number` data type.
- From all `date` entries contained in *element specification* which represent exact dates the system incorporates zero.
- Where any entries contain intervals of time the system uses the gap in days between the two terms.

8.1.1.4.2.42 The element function `lemma[]`
The element function `lemma[]` is written as follows:

```
lemma[element specification,latin]
```

It is defined by the set of entries of the `text` data type which is created if the set of entries of the same data type specified by *element specification* is lemmatised using the lemmatising procedure described by the second argument.

At the present time, the user can only invoke the lemmatising system used by the Istituto Linguistica Computazionale by specifying the keyword `latin`. This system was developed by Andrea Bozzi; an earlier version has been implemented as a stand-alone program by Giuseppe Cappelli. A fuller description is available on request.

In order to use this system it is important to know:
- the lemma returned by the system is a basic form of the word (nominative singular, first person singular in the present tense, or a non-inflectable form).
- the algorithm supports lemmatising operations, i.e. one original entry may result in several entries, each containing a potential lemma.

8.1.1.4.2.43 The element function `soundex[]`
The element function `soundex[]` is written as follows:

```
soundex[element specification,definition name]
```

It is defined as the set of entries of the `text` data type which are expressed by *element specification* after they have been converted by the soundex definition *definition name*. Cf. section 7.3.1.8 for more information on this class of objects.

8.1.1.4.2.44 The element function `skeleton[]`
The element function `skeleton[]` is written as follows:

```
skeleton[element specification,definition name]
```

It is defined as the set of entries of the `text` data type which are expressed by *element specification* after they have been converted by the skeletonising definition *definition name*. Cf. section 7.3.1.9 for more information on this class of objects.

8.1.1.4.2.45 The element function `query[]`
The element function `query[]` is written as follows:

```
query[]
```

It is defined as the last entry to be reached by `query`.

This function plays a particularly important role when you are processing data from which you are intending to select a very specific entry using `query`. An example follows:

The task

```
query name=example;part=:placename="Bavaria"
write part=:placename
```

is interpreted as follows, in accordance with the general system logic:

"Search the current database for all `placename`s where *at least one entry* contains the character string `Bavaria`. Print out the contents of the *element* `placename` for all such groups."

The interpretation of your instruction described above is normally perfectly acceptable, but when your data is presented as follows:

```
placename=A-village, Bavaria; B-town, Swabia
```

you will find that you cannot produce a list which only contains locations in Bavaria. The system also interprets the following task:

```
query name=example;part=:placename="Bavaria"
index part=:placename
```

according to the same logic, so that you obtain an alphabetical list of all locations which appear in elements containing the character string `Bavaria` *in at least one entry*: and in this case, also including "`B-town, Swabia`".

The element `:query[]` provides the solution to this problem:

```
query name=example;part=:placename="Bavaria"
write part=:query[]
```

is interpreted as follows:

"Search the current database for all `placenames` in which *at least one entry* contains the character string `Bavaria`. Print out the contents of all such *entries*."

Accordingly, the following instruction:

```
query name=example;part=:placename="Bavaria"
index part=:query[]
```

will provide you with an index of all *entries* containing the character string `Bavaria`. "`B-town, Swabia`" will *not* appear in this list.

8.1.1.4.2.46 The element function `keyword[]`

The element function `keyword[]` is written as follows:

```
keyword[name,element specification]
```

It is defined as a set of entries of the `text` data type, each containing the name of one catchword to which the *element specification* in the catalogue *name* is linked. Catchwords are the names of lists of references which were generated by the menu option `Link` in the main menu for interactive processing of a catalogue (cf. "Halbgraue Reihe" volume B10, section 2.2.3) or by a `keyword[]` directive with the parameter `bridge=yes` (cf. section 7.3.1.15.1.3) specified in the context of a catalogue definition.

8.1.1.4.2.47 The element function `also[]`

The element function `also[]` is written in one of the two following ways:

```
also[]
```

or

```
also[element specification]
```

- In the first case, it is defined as a set of entries of the `number` data type, each of which represents a view in which the most recently activated group exists.
- In the second case, it is defined as a set of entries of the `number` data type, each of which represents a view in which at least one element of the set of elements defined by *element specification* exists.

8.1.1.4.2.48 The element function `words[]`

The element function `words[]` is written as follows:

```
words[element specification]
```

It is defined as an entry of datatype `number` which represents the number of words (separated by one or more blank characters) contained in *element specification*.

8.1.1.4.2.49 The element function `last[]`

The element function `last[]` is written as follows:

```
last[element specification]
```

It is defined as the last entry within *element specification*.

8.1.1.5 Expressions

Expressions come into being when you use the *expression operator* to append a second path ending in an element specification to the element specification closing a first path. You should also insert a *link operator* between the expression operator and the following path.

This means that expressions have the general form:

$$< path - 1 >< expression\ operator >< link\ operator >$$
$$< path - 2 >$$

Expression operators are the "ampersand" (&) and the "vertical line" (|). Link operators are variously defined and interpreted for each data type.

The system processes the expression for each entry in *path-1* in turn; it combines the entries in *path-2* to form *a single* entry before performing the requested operation in accordance with the rules described in section 8.1.1.4.2.36.

In practice, of course, you would hardly ever use expression operators to process constants as we do in the following examples: we are using them in this way simply so that it is easy to check the results. The following task illustrates how to use expressions in practice (cf. also section 8.1.1.5.2.1). It instructs the system to output the age of the individuals concerned:

```
query name=source;part=person
write part=:name,:first_name,:date&-:date_of_birth
stop
```

No link operators have been defined for the `relation`, `location` and `image` data types to date.

8.1.1.5.0 Expression operators

The operators defining the way in which two elements shall be combined consist of two parts. The second part, the link operator, defines how the entries shall be connected. The first, the expression operator, defines how missing information shall be handled in the evaluation of an expression.

In both cases expressions are strictly evaluated from left to right. All link operators have the same precedence, bracketing by "(" and ")" is possible, however.

8.1.1.5.0.1 Expression operator &

When this expression operator is used any expression which contains at least one missing information is considered missing as a whole. If within a given group the element `name` contains "`Smith`", the element `first_name` is missing and the element `occupation` contains "`carpenter`", the expression

```
:name&+:first_name&+:occupation
```

would therefore be considered as missing.

8.1.1.5.0.2 Expression operator |

When this expression operator is used, a missing element is ignored, together with the operator which links it to the previous part of the expression. With the data described in the preceding section, the expression

`:name|+:first_name|+:occupation`

would therefore be evaluated as "`Smithcarpenter`".

8.1.1.5.1 Expressions using the `text` data type

If *path-1* is of the `text` data type, *path-2* must also be of the `text` data type.

8.1.1.5.1.1 The `text` link operator '+'

This operator appends the contents of the second path directly to the contents of the first path.

```
:form["Neo-Classical ",text]&+
    :form["architectural style",text]
```

would therefore result in the character string `"Neo-Classical architectural style"`.

8.1.1.5.1.2 The `text` link operator '-'

This operator subtracts the contents of the second path from the contents of the first path.

```
:form["Neo-Classical architectural style",text]&-
    :form["architectural ",text]
```

would therefore result in the character string `"Neo-Classical style"`.

8.1.1.5.1.3 The `text` link operator '*'

This operator retains in the contents of the first path the character string which starts with the first occurrence of the contents of the second path.

```
:form["Neo-Classical architectural style",text]&*
    :form["architectural ",text]
```

would therefore result in the character string `"architectural style"`.

8.1.1.5.1.4 The `text` link operator ':'

This operator retains in the contents of the first path the character string leading up to the start of the string contained in the second path.

```
:form["Neo-Classical architectural style",text]&:
    :form["architectural",text]
```

would therefore result in the character string `"Neo-Classical"`.

8.1.1.5.2 Expressions using the `date` data type

If *path-1* is of the `date` data type, *path-2* must also be of the `date` or `number` data types. In the second case, the system expects *path-2* to represent a number of days.

8.1.1.5.2.1 The `date`–`date` link operator '-'

This operator converts the first path's entries into the `number` data type and reproduces in them the interval in days between the two `date` paths. If one or both of the paths represents an interval of time, the system produces a numerical range which represents the intervals of time between the corresponding interval boundaries. In order to make sure that the minimum limit of the range is less than its maximum limit, the system swaps the two parts round if necessary.

```
:form["13.4.1721",date]&-
    :form["15.4.1721",date]
```

thus results in the numerical value `-2.00`.

```
:form["13.4.1721-16.4.1721",date]&-
    :form["14.4.1721-15.4.1721",date]
```

results in the numerical value `-1.00 - 1.00`.

8.1.1.5.2.2 The `date`–`number` link operator '+'

This operator converts the contents of *path-1* into the `date` specification occurring *path-2* days after the original `date` specification.

```
:form["13.4.1721",date]&+
    :form["1week",number,date]
```

thus results in the calendar date `20.4.1721`.

```
:form["13.4.1721-15.4.1721",date]&+
    :form["1day-1week",number,nodefault]
```

results[2] in the calendar date `14.4.1721 - 22.4.1721`.

8.1.1.5.2.3 The `date`–`number` link operator '-'

This operator converts the contents of *path-1* into the `date` specification occurring *path-2* days before the original `date` specification.

```
:form["13.4.1721",date]&-
    :form["1week",number,date]
```

thus results in the calendar date `6.4.1721`.

```
:form["13.4.1721-15.4.1721",date]&-
    :form["1day-1week",number,nodefault]
```

thus results[3] in the calendar date `6.4.1721 - 14.4.1721`.

[2] This is an *example*. There is no standard `number` definition which calculates intervals in days.

[3] See previous remark.

8.1.1.5.3 Expressions using the `number` data type

If *path-1* is of the `number` data type, *path-2* must also be of the `number` data type.

8.1.1.5.3.1 The `number` link operator '+'

This operator corresponds to the arithmetical operation addition.

```
:form["3",number]&+
    :form["2",number]
```

thus corresponds to the numerical value `5.00`.

```
:form["3-5",number]&+
    :form["2-3",number]
```

corresponds to the numerical value `5.00 - 8.00`.

8.1.1.5.3.2 The `number` link operator '-'

This operator corresponds to the arithmetical operation subtraction.

```
:form["3",number]&-
    :form["2",number]
```

thus corresponds to the numerical value `1.00`.

```
:form["3-5",number]&-
    :form["2-3",number]
```

corresponds to the numerical value `0.00 - 3.00`.

8.1.1.5.3.3 The `number` link operator '*'

This operator corresponds to the arithmetical operation multiplication.

```
:form["3",number]&*
    :form["2",number]
```

thus corresponds to the numerical value `6.00`.

```
:form["3-5",number]&*
    :form["2-3",number]
```

corresponds to the numerical value `6.00 - 15.00`.

8.1.1.5.3.4 The `number` link operator ':'

This operator corresponds to the arithmetical operation division.

```
:form["3",number]&:
    :form["2",number]
```

thus corresponds to the numerical value `1.50`.

```
:form["3-5",number]&:
    :form["2-3",number]
```

corresponds to the numerical value `1.00 - 2.50`.

8.1.1.5.4 Expressions with the `category` data type

If *path-1* is of the `category` data type, *path-2* must also be of the `category` data type.

8.1.1.5.4.1 The `category` link operator '+'

This operator represents the logical total.

```
:form["ab",category]&+
    :form["cd",category]
```

thus corresponds to the series of abbreviations `a, b, c, d`.

8.1.1.5.4.2 The `category` link operator '-'

This operator represents the logical difference.

```
:form["abcd",category]&-
    :form["cd",category]
```

thus corresponds to the series of abbreviations `a, b`.

8.1.1.6 Network transitions

A network transition is expressed by the fact that immediately after an element specification containing entries with the preferred data type `relation`, the path is continued by additional group and/or path definitions.

8.1.1.6.1 Unconditional network transitions

If you note down this element specification without any additions the system proceeds down the section of the path following the group referred to by the network identifiers in the element specification's entries.

Hence the path `person:related:name` starts from the group `person` and goes on to the element `name` in *every* group referred to by a network identifier in one of the entries in the element `related`.

8.1.1.6.2 Conditional network transitions

If, on the other hand, you mention the element specification as the first argument in the element function `target[]` (cf. section 8.1.1.4.2.37), the system only proceeds down the remainder of this path if the element and the group to which the elements refer fulfil certain conditions.

`person:target[:relationship,"debtor"]:name` instructs the system to trace only those network identifiers in the entries in the element `related` which refer to groups named `debtor`.

8.1.1.6.3 Transitions to other databases

In principle, every element of the `relation` data type can refer to any number of different databases. References of this type are declared using the resources described in section 10.

In principle, network transitions which link several databases together have the same syntax as the transitions described above. In order to be able to assess whether the part of a path noted after the transition to another database is correct, $\kappa\lambda\epsilon\iota\omega$ must know the target database for this transition. For this reason, the system always interprets a network transition of the kind described above as an instruction to select only those entries from the total number of available entries in an element which remain in the original database.

If you wish to select transitions to *another* database, you must specify the name of the database to which the transition should be made in a pair of angled brackets immediately after the name of the element in which the transitions are being selected.

To preserve formal consistency you must still do this even if the element being processed only contains transitions to one database.

So if, starting from a database named `source`, you wanted to output information contained in this database as well as information contained in another database named `taxlist`, which can be accessed via an element with the same name, you would set the system the following task:

```
query name=source
write part=:each[],:taxlist<taxlist>:each[],
        :addresses<addresses>:each[]
```

Transitions between databases correspond in every other way to transitions within a database. This means you can also use the conditional form, as shown in the following example.

```
query name=source
write part=:each[],
        :target[:taxlist<taxlist>,"person"]:each[],
        :addresses<addresses>:each[]
```

In order to be selected for processing, the reference in this instance must indicate the group `person` in the database `taxlist`.

8.1.2 Conditions

You can append a condition, prefixed by a data signal character 3, as the very last component at the end of every path; i.e. after the element specification.

8.1.2.1 General

Every condition in $\kappa\lambda\epsilon\iota\omega$ can be reduced to the following form:

$< path >< data\ signal\ character\ 3 >< reference\ pattern >$

Relational operators are in principle constituents of the reference pattern, hence appear to the right of the data signal character.

Every such condition of the named variety can be translated as follows:
"At least one element in the set *path* must fulfil the condition specified by *reference pattern*".

8.1.2.1.1 Logical relationships between conditions

Instead of a condition of the named variety, you can use a *compound condition*, the separate subconditions of which are connected by *logical operators*.

Subconditions may themselves be compound conditions.
The relationship can be expressed in one of the two following ways:

- $< path >< data\ signal\ character\ 3 >< reference\ pattern\ a >$
  $< logical\ operator >$
  $< path >< data\ signal\ character\ 3 >< reference\ pattern\ b >$
  (Type 1)
- $< path >< data\ signal\ character\ 3 >< reference\ pattern\ a >$
  $< logical\ operator >< reference\ pattern\ b >$
  (Type 2)

These two forms are not equivalent.

Type 1 is interpreted as follows: "The condition as a whole is fulfilled if this *logical relationship* between the following subconditions is true: 'At least one element in the set *path* should fulfil the condition laid down by *reference pattern a*' and 'At least one element in the set *path* should fulfil the condition laid down in *reference pattern b*'".

Type 2, on the other hand, is interpreted as follows: "The condition as a whole is fulfilled if at least one element in the set *path* fulfils the condition defined by the *logical relationship* between *reference pattern a* and *reference pattern b*."

There should be at least one blank space before and after every logical relationship.

8.1.2.1.1.1 Logical operator `and`

If two subconditions are linked together by the keyword `and`, both must be true in order to satisfy the compound condition.

8.1.2.1.1.2 Logical operator `or`

If two subconditions are linked together by the keyword `or`, it is sufficient for one of them to be true in order to satisfy the compound condition.

8.1.2.1.2 Negation of conditions

If the keyword `not` appears in front of a subcondition, this condition is treated as true as long as it is *not* fulfilled. There should be one or more spaces before and after the keyword.

8.1.2.1.3 Bracketing

The following applies by default:

- `or` divides a compound condition if it appears before `and`.
- `not` negates the immediately following subcondition, until the next logical condition.

You can modify both procedures by inserting a pair of parentheses. Parentheses combine compound conditions to form a single subcondition, which can then be logically linked to another subcondition or negated.

8.1.2.2 Reference patterns which do not depend on data type

Certain relation conditions do not take the type of information to be treated into consideration. They relate to one property of a set expressed by a path.

8.1.2.2.1 The reference pattern `null`

If a reference pattern consists of the keyword `null`, no element of the set of elements specified by the path should exist, if the system is to regard the condition for an element in the set of containing groups as fulfilled. It also does so if one of the groups in the path which lead to the element specification, does not exist.

If this reference pattern is negated, at least one element in the set of elements defined by the last component in the path must exist.

8.1.2.3 Reference patterns which depend on data type

Most reference patterns relate to a particular data type. In order to process them, the system only calls those entries from the set defined by the path which possess the required data type.

8.1.2.3.0 Reference patterns

Reference patterns consist of

- one and only one *comparand* and
- (optionally) a number of *comparison modifiers*.

You can specify these components in any order, each separated from the others by one or more spaces.

8.1.2.3.0.1 Comparands

Comparands consist of data of the specified data type. The data either appears in a user-defined constant, or it is taken from other entries.

8.1.2.3.0.1.1 Comparisons using constants

These comparisons have the general form:

$< path - 1 >=< constant >$

If the comparand is expressed as a constant, at least one element in the set defined by the path must fulfil the condition set by the constant.

8.1.2.3.0.1.2 Comparisons using the contents of other entries

Comparisons of this type have the general form:

$< path - 1 >=< path - 2 >$

If you want the comparison to be successful, at least one of the entries contained in *path-1* must fulfil the condition set by at least one of the entries contained in *path-2*.

The system always checks all the entries contained in *path-1*; thus *path-1* can fulfil a condition as many times as it contains entries.

The comparison with the entries in *path-2*, on the other hand, terminates as soon as it is successful: hence each entry in *path-1* can only fulfil the condition at most once.

8.1.2.3.0.2 Comparison modifiers

Comparison modifiers are keywords which must be written in the prevailing upper or lower case. They are specific to the data type in each instance, and modify the standard form of the comparison which is being performed.

8.1.2.3.1 The `text` data type

Reference patterns for the `text` data type check two character strings, to see whether one is contained in the other in the manner specified.

8.1.2.3.1.1 Comparison modifiers

If no comparison modifier is present, the system expects the entries appearing in *path-1* to contain the comparand in any position.

8.1.2.3.1.1.1 Comparison modifier `start`

If the reference pattern contains the keyword `start`, the comparand should appear at the beginning of one of the entries in *path-1*.

8.1.2.3.1.1.2 Comparison modifier `limit`

If the reference pattern contains the keyword `limit`, the comparand should appear at the end of one of the entries in *path-1*.

8.1.2.3.1.1.3 Comparison modifier `equal`

If the reference pattern contains the keyword `equal`, the comparand should exactly match one of the entries in *path-1*. You can formulate this in another way by specifying the keyword `start` together with the keyword `limit`.

8.1.2.3.2 The `date` data type

Reference patterns for the `date` data type check two calendar dates, to see if one is related to the other in the manner specified.

8.1.2.3.2.1 Comparison modifiers

If no comparison modifier is present, the system requires the calendar dates in the entries in *path-1* to match the comparand exactly. However this match only applies to the day, month and year as displayed in the system's own internal format, after conversion.

If the comparand is an interval of time, the condition can only be fulfilled by calendar dates which are also intervals of time with exactly the same *termini post* and *ante quem*.

8.1.2.3.2.1.1 Comparison modifier `equal`

The comparison modifier `equal` demands that in addition to matching, as they should if no comparison modifier were present, all other parts of the calendar date should also match; i.e. the two calendar dates should be written in an identical style, and neither or both of them should be distinguished by question marks or exclamation marks, etc.

8.1.2.3.2.1.2 Comparison modifier `before`

In order to fulfil a condition which includes this comparison modifier[4] , the exact date (or *terminus ante quem*) of an entry from *path-1* must occur before the exact date (or *terminus post quem*) of the comparand.

8.1.2.3.2.1.3 Comparison modifier `after`

In order to fulfil a condition which includes this comparison modifier[5] , the exact date (or *terminus post quem*) of an entry from *path-1* must occur after the exact date (or *terminus ante quem*) of the comparand.

8.1.2.3.2.1.4 Comparison modifiers `before equal`

In order to fulfil a condition which includes these two comparison modifiers[6] , the exact date (or *terminus ante quem*) of an entry from *path-1* must occur before the exact date (or *terminus post quem*) of the comparand, or else be identical to the latter.

8.1.2.3.2.1.5 Comparison modifiers `after equal`

In order to fulfil a condition which includes these two comparison modifiers[7] , the exact date (or *terminus post quem*) of an entry from *path-1* must occur after the exact date (or *terminus ante quem*) of the comparand, or else be identical to the latter.

---

[4] The system accepts `less` as a synonym.

[5] The system accepts `greater` as a synonym.

[6] The system accepts `less equal` as a synonym.

[7] The system accepts `greater equal` as a synonym.

8.1.2.3.3 The `number` data type

Reference patterns for the `number` data type check two items of numerical data to see whether one relates to the other in the manner specified.

8.1.2.3.3.1 Comparison modifiers

If no comparison modifier is present, the system demands that the numerical data in the entries in *path-1* should exactly match the comparand. However, this match only applies to the numerical value resulting from the conversion into the system's own internal format.

If the comparand is a numerical range the condition can only be fulfilled by data which is equal to the same numerical range, with exactly the same minimum and maximum values.

Hint: *It is almost entirely true to say that the more complex and less common of the following comparison modifiers only ever affect numerical ranges specified in terms of minimum and maximum values. In order make it easier to understand the following definitions, we are going to base them on such numerical values. An exact number may be understood as a numerical range in which the minimum and maximum values are equal.*

8.1.2.3.3.1.1 Comparison modifier `equal`

The comparison modifier `equal` demands that in addition to matching, as they should if no comparison modifier were present, all other components of the numerical data should also match: i.e. both numbers should have the same uncertainty operators and neither or both of them should be distinguished by question marks or exclamation marks, etc.

8.1.2.3.3.1.2 Comparison modifier `circa`

In order to fulfil a condition which includes this comparison modifier, an entry from *path-1* should intersect the comparand by a bare minimum. To do so it is enough if either

- the entry's minimum value is less than or equal to the comparand's maximum value, and simultaneously the entry's maximum value is greater than or equal to the comparand's minimum value,

  or
- the comparand's minimum value is less than or equal to the entry's maximum value and simultaneously the comparand's maximum value is greater than or equal to the entry's minimum value.

8.1.2.3.3.1.3 Comparison modifier `less`

In order to fulfil a condition which includes this comparison modifier, an entry from *path-1* must have a maximum value which is less than the comparand's minimum value.

8.1.2.3.3.1.4 Comparison modifier `greater`

In order to fulfil a condition which includes this comparison modifier, an entry from *path-1* should have a minimum value which is greater than the comparand's maximum value.

8.1.2.3.3.1.5 Comparison modifiers `equal circa`

In order to fulfil a condition which includes these two comparison modifiers, an entry from *path-1* should intersect the comparand rather more narrowly than would be the case if you defined this condition with the `circa` comparison modifier alone. To fulfil this condition, it is enough if either

- the entry's minimum value is less than or equal to the comparand's minimum value and simultaneously the entry's maximum value is greater than or equal to the comparand's maximum value,
  or
- the comparand's minimum value is less than or equal to the entry's minimum value and simultaneously the comparand's maximum value is greater than or equal to the entry's maximum value.

8.1.2.3.3.1.6 Comparison modifiers `equal greater`

In order to fulfil a condition which includes these two comparison modifiers, an entry from *path-1* should have a minimum value greater than or equal to the comparand's maximum value.

8.1.2.3.3.1.7 Comparison modifiers `equal less`

In order to fulfil a condition which includes these two comparison modifiers, an entry from *path-1* should have a maximum value less than or equal to the comparand's minimum value.

8.1.2.3.3.1.8 Comparison modifiers `circa greater`

In order to fulfil a condition which includes these two comparison modifiers, an entry from *path-1* should have a maximum value greater than or equal to the comparand's minimum value.

8.1.2.3.3.1.9 Comparison modifiers `circa less`

In order to fulfil a condition which includes these two comparison modifiers, an entry from *path-1* should have a minimum value less than or equal to the comparand's maximum value.

8.1.2.3.3.1.10 Comparison modifiers `equal circa greater`

In order to fulfil a condition which includes these three comparison modifiers, an entry from *path-1* should have a maximum value greater than or equal to the comparand's maximum value.

8.1.2.3.3.1.11 Comparison modifiers `equal circa less`

In order to fulfil a condition which includes these three comparison modifiers, an entry from *path-1* should have a minimum value less than or equal to the comparand's minimum value.

8.1.2.3.4 The `category` data type

Reference patterns for the `category` data type check two combinations of categorical abbreviations to see whether one contains the other.

8.1.2.3.4.1 Comparison modifiers

If no comparison modifier is present, the system expects the entries which appear in *path-1* to contain the comparand; it ignores the presence or absence of additional categorical abbreviations.

8.1.2.3.4.1.1 Comparison modifier `equal`

This comparison modifier specifies that except for the comparand, no other categorical abbreviations should be included in entries in *path-1*.

8.1.2.3.4.1.2 Comparison modifier `others`

This comparison modifier specifies that in addition to the comparand, at least one other categoric abbreviation should be included in the entries in *path-1*.

8.1.2.3.5 The `relation` data type

Reference patterns for the `relation` data type differ widely, depending on whether you are comparing the data with a *constant* or with *path-2*.

8.1.2.3.5.1 `relation` comparisons with constants

If you are making a comparison with constants, reference patterns for the `relation` data type do not allow you to specify comparison modifiers. In this type of comparison a constant consists of a path in which built-in functions and the names of substitution definitions are illegal (cf. section 7.3.1.11 for more on this), and which ends with the name of an element.

The condition is fulfilled by entries from *path-1* which contain at least one network identifier referring to an element which is correctly described by the given path.

The path given as the constant can be as long as you like. The shortest possible constant of this type thus consists of the name of an element: this means that the exact whereabouts of this element within the database structure is of no importance with respect to fulfilling the condition. The longest possible constant of this type represents the entire path between the document and this particular element.

8.1.2.3.5.2 `relation` comparisons with another path

Comparisons of this type expect a match between two subnetworks in one of the networks of the `relation` data type. (cf. section 4.1.1.1.5 for more about these concepts).

8.1.2.3.5.2.1 Comparison modifiers

The system treats a condition without a comparison modifier as fulfilled if the comparand's subnetwork is completely contained in a subnetwork (described by a entry) in *path-1.* [8]

8.1.2.3.5.2.1.1 Comparison modifier `start`

In order to fulfil a condition which includes this comparison modifier, the comparand's subnetwork must match the beginning of one of subnetworks in *path-1*.

8.1.2.3.5.2.1.2 Comparison modifier `limit`

In order to fulfil a condition which includes this comparison modifier, the comparand's subnetwork must match the end of one of the subnetworks in *path-1*.

8.1.2.3.5.2.1.3 Comparison modifier `equal`

A condition which includes this comparison modifier is fulfilled if the comparand's subnetwork is completely identical to a subnetwork in *path-1*.

You can formulate this condition in another way by specifying `start` and `limit` simultaneously.

8.1.2.3.6 The `location` data type

For purposes of comparison, the system currently treats data of this data type as if it was `text` data. The definitions in section 8.1.2.3.1 also apply in this case.

8.1.2.3.7 The `image` data type

For purposes of comparison, the system currently treats data of this data type as if it was `text` data. The definitions in section 8.1.2.3.1 also apply in this case.

[8] When you set up a database, the subnetworks (assuming they overlap) are also identical by definition. Comparison modifiers only make sense in databases where you have made changes to individual entries deleting single relationships between the elements defining the network. This can be done only with the database editor, deleting individual entries of such an element. This editor is at the time of printing available only with version 6 of $\kappa\lambda\epsilon\iota\omega$. More

8.1.3 Examples

Examples of paths without conditions:

```
person
name
person/father/migration:date
person/father/part[migration,3-4]:comment[:date]
property:each[]
:each[]
```

Examples of paths with conditions:

```
:name="Smith"
:each[]=("Red" or "red") and not "cloth" limit
person:first_name1="Mary" or :first_name2="Anne"
food_supplies:identifier="oats" and query[]/creditor:name="Smith"
```

8.2 Selecting part of a database for processing

Whenever you set κλειω the task of processing a database, this task must relate to a portion of the data held in the database. This means that κλειω must know which database to process.

Once you make a database known to the system, it applies all subsequent analysis instructions to this database until you specify a new one. Unless you explicitly tell the system which part of the database to process, it processes all documents in the order in which they were imported into the database.

8.2.1 The `query` command

The general form of the `query` command is:

NAME=<filename> **(default: none)**
PART=<pathlist> **(default: none)**

It describes the parts of the database which the system should process. For this purpose, it specifies a series of paths, each of which is processed in turn. As it proceeds down each path, it remembers the position of each element it finds in the set defined by that path, and starts to execute all the instructions which specify which items of information should be processed. Hence all paths specified in these commands are relative to the paths contained in the `query` command.

If you do not issue an explicit `query` command, the paths of all analysis instructions are relative to the group defining each document.

If a `query` command's paths contain conditions, the system only processes those parts of the database which fulfil these conditions.

8.2.1.1 The `name=` parameter

The `name=` parameter accepts the name of a previously encountered structure definition as a value. This name is used to set up relations with all parts of the database which are required during the various phases of the processing.

If a `query` command does not contain this parameter, the system continues to process the last-named database.

The first task in every κλειω program which is intended for processing a database should therefore include a `query` command with the name of the database in its `name=` parameter.

8.2.1.2 The `part=` parameter

The `part=` parameter expects a list of paths as a value.

The system processes these paths *one after the other*.

8.2.2 Block-structured κλειω tasks

Normally, κλειω tasks consist of a (sometimes implicit) `query` command and one or more analysis instructions.

All analysis instructions are executed for all parts of the database which have been selected by `query`.

In this case we say that the κλειω program consists of an *unnamed block* of commands.

In addition, you have the option of issuing several `query` commands in a $\kappa\lambda\epsilon\iota\omega$ program, each of which processes a different part of the database using different analysis instructions.

Every time $\kappa\lambda\epsilon\iota\omega$ decides whether a particular part of a database fulfils a condition set within a `query` command, the system checks to see whether an instruction block exists which should only be executed once a positive (or negative) decision has been made. If it finds a suitable block, the system proceeds to process it.

Blocks consists of

- a *block initialisation command*,
  - `confirm` or
  - `negate`,
- any number of $\kappa\lambda\epsilon\iota\omega$ instructions and
- an `exit` command.

Blocks may contain other blocks – hence also `query` commands.

The `query` command in each block should appear before any analysis instructions.

8.2.2.1 The `confirm` command

The general form of the `confirm` command is:

NAME=<name of block>	**(MUST be specified!)**
TYPE=Permanent | Temporary	**(Default: Permanent)**

This command appears (a) after a `query` command which includes a condition and before all subsequent analysis commands or (b) after an `exit` command which was used to close a block initialised by `negate` for which no parallel block initialised by `confirm` has yet been declared.

It initialises an instruction block which is only executed if the preceding `query` command's conditions are fulfilled.

8.2.2.1.1 The `name=` parameter

The `name=` parameter accepts any user-defined name as a value, by which the block is then known. This name must be different from the names of all other blocks which have not yet been closed by `exit`.

8.2.2.1.2 The `type=` parameter

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

- `permanent` instructs the system to consider any `query` command contained within the block defined by this `confirm` command to become the target for any `query[]` built-in function encountered thereafter. This is the default behaviour of the system.
- `temporary` asks $\kappa\lambda\epsilon\iota\omega$ to let following `query[]` built-in functions refer to the last `query` command encountered before the `confirm` block in question.

8.2.2.2 The negate command

The general form of the negate command is:

NAME=<name of block> **(MUST be specified!)**
TYPE=Permanent | Temporary **(Default: Permanent)**

The command appears (a) after a query command which includes a condition, but in front of any subsequent analysis instruction or (b) after an exit command closing a block initialised by confirm for which no parallel block initialised by negate has yet been declared.

It initialises an instruction block which the system only executes if the preceding query command's conditions are not fulfilled.

8.2.2.2.1 The name= parameter

The name= parameter accepts any user-defined name as a value, by which the block is then known.This name must be different from the names of all other blocks which have not yet been closed by exit.

8.2.2.2.2 The type= parameter

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

- permanent instructs the system to consider any query command contained within the block defined by this negate command to become the target for any query[] built-in function encountered thereafter. This is the default behaviour of the system.
- temporary asks $\kappa\lambda\epsilon\iota\omega$ to let following query built in functions still refer to the last query command encountered before the negate block in question.

8.2.2.3 The exit command

The general form of the exit command is:

NAME=<name of block> **(MUST be specified!)**

This instruction closes one or more blocks which have been previously defined by confirm or negate.

8.2.2.3.1 The name= parameter

The name= parameter accepts any user-defined name as a value, by which a block which has not yet been closed by exit is then known.

exit is used to close all blocks which were previously opened by confirm or negate, in the opposite order to that in which they were initialised, up to and including the block with the same name as the one in the exit command's name= parameter.

If the system cannot find such a block, the task is invalidated, but any blocks which were closed in error are not opened again.

8.2.3 Examples

Examples of how to select information using simple `query` commands:

```
query name=source

query name=source;part=:identifier

query name=source;part=commodity:identifier

query name=source;part=commodity:identifier="oats" or "barley"
```

Example of a block structure:

```
query name=source;part=person:origin="Aberdeen"
confirm name=aberdonian
          |
    analysis instructions
          |
exit name=aberdonian
negate name=others
query part=:origin="Scotland"
confirm name=scotsmen
          |
    analysis instructions
          |
exit name=scotsmen
negate name=remainder
          |
    analysis instructions
          |
exit name=remainder
exit name=others
```