2. General

2.1 $\kappa\lambda\epsilon\iota\omega$ programs

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

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

2.1.1 Instructions

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

2.1.1.1 Command words

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

2.1.1.2 Types of instruction

An instruction either changes the way $\kappa\lambda\epsilon\iota\omega$ works in general (*definition instruction*), or causes $\kappa\lambda\epsilon\iota\omega$ to perform a specific action (*executable instruction*).

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

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

2.1.1.3 Specification

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

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

An empty line *always* terminates the preceding instruction.

2.1.1.3.1 Parameters

Every specification consists of a series of *parameters*.

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

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

2.1.1.3.1.1 Parameter values

Parameter values are either:

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

2.1.1.3.1.1.1 Constants

Constants are

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

2.1.1.3.1.1.2 Names

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

Names may consist of combinations of the following characters:

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

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

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

2.1.1.3.1.1.3 Keywords

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

2.1.1.3.1.1.4 Built-in functions

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

2.1.1.3.1.1.5 Lists of values

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

2.1.1.4 Standard format

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

$$< \mathit{command\ word} >< \mathit{space} >< \mathit{parameter\ name} >=< \mathit{parameter\ value} >;$$
$$< \mathit{parameter\ name} >=< \mathit{parameter\ value} > \ldots$$

2.1.2 Definitions

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

2.1.2.1 Commands which open definitions

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

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

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

2.1.2.2 Definition directives

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

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

2.1.2.3 `exit` directive

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

2.1.2.4 Standard format

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

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

2.2 Tasks

$\kappa\lambda\epsilon\iota\omega$ attempts to link instructions and definitions together in a sequence, creating an executable program in the process. $\kappa\lambda\epsilon\iota\omega$ continues to do this until it finds one of the three following situations:

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

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

Unless $\kappa\lambda\epsilon\iota\omega$ finds an error, it will now start to execute the task.

2.2.1 Effective duration of instructions and definitions

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

If, on the other hand, you use the `continue` command to indicate the end of the task, $\kappa\lambda\epsilon\iota\omega$ will "forget" all executable instructions in this particular task and prepare to process the next task.

However, the system does remain aware of all definitions describing logical objects which appear in this task until the $\kappa\lambda\epsilon\iota\omega$ program finally terminates; similarly, the effects of any definition instructions continue to apply until the end of the program.