Series: Monografias em Ciência da Computação
Nº 18/78

JACKDAW II - Preliminary Specification

Part 3
The Definition Language

by

Michael F. Challis

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 225 – CEP-22453
Rio de Janeiro – Brasil

Series: Monografias em Ciência da Computação No. 18/78

Series Editor: Michael F. Challis October, 1978

Jackdaw II - Preliminary Specification*

Part 3

The Definition Language

by

Michael F. Challis

*This research has been sponsored by FINEP and CNPq.

Abstract

This monograph forms part of the preliminary definition of the Jackdaw II data base package (see the Preface).

It contains a formal description of the language used to define the structure of a data base (often called the 'DDL'). This language also includes facilities for modifying the structure of an existing data base.

Keywords: data base, DDL

Resumo

Esta monografia faz parte da definição preliminar do sistema de banco de dados Jackdaw II (veja o Prefácio).

Ela contém uma descrição formal da linguagem utilizada para definir a estrutura de um banco de dados (a chamada 'DDL'). Esta linguagem inclui também facilidades para atualizar a estrutura de um banco de dados que já existe.

Palavras chave: banco de dados, DDL

CONTENTS

Preface

This monograph forms part of the preliminary definition of the Jackdaw II data base package. The complete definition will include the following sections:

1. Introduction

   This section gives an overview of the package and introduces its major components.

2. Structural Description

   This describes the facilities provided for the representation of data and data relationships in a data base, and the 'access paths' which may be followed to retrieve items stored within it.

3. The Definition Language

   This contains a description of the language used to define the structure of a data base (often called the 'DDL'). The language also includes facilities for modifying the structure of an existing data base.

4. The Procedural Interface

   The 'nucleus' of the package will consist of a library of procedures which may be called to create, modify or delete items within a data base. This section describes these 'interface procedures'. The package will be written in BCPL, and the library of interface procedures will also initially be available only to BCPL application programs.

5. Concurrent Access

   The options available for data base access in both single-user and multi-user environments are described, including the facilities provided to help maintain data base consistency and integrity.

6. The SLIDE system

This section describes the language intended for the 'end-user'. SLIDE includes facilities for 'loading' (the addition of relatively large volumes of data) and report generation as well as facilities for casual interrogation and updating. The language has two levels - a 'low level', with commands which correspond approximately to the interface procedures of the nucleus, and a 'high level' with more powerful primitives.

The high level language may be easily extended to generate languages 'tailored' to particular applications.

Jackdaw II is a development of the Jackdaw data base package, which was originally developed by the author at the University of Cambridge, England (where it has been in production use since 1973), and which is now also available on the IBM 370/165 at PUC-RJ.

References

[1] - Richards, M., 1974. "The BCPL Programming Manual", Computer Laboratory, University of Cambridge, Cambridge, England.

[2] - Challis, M.F., 1974. "The JACKDAW Data Base Package", Proc. SEAS Spring Technical Meeting, St. Andrews, Scotland, April 1974. (Also available as "Technical Report No. 1" from the Computer Laboratory, University of Cambridge, Cambridge, England).

[3] - Challis, M.F., 1978. "Técnicas de Integridade no Sistema de Jackdaw de Bancos de Dados", to appear in the proceedings of "V Seminário Integrado de Software e Hardware Nacionais", Rio de Janeiro, Brasil. (Also available in English as "Integrity Techniques in the Jackdaw Database Package", Monografia em Ciência da Computação no. 9/77, Dept. de Informática, Pontifícia Universidade Católica, Rio de Janeiro, Brasil).

[4] - Challis, M.F., 1978. "Database Consistency and Integrity in a Multi-user Environment", to appear in Proc. International Conference on Data Bases: Improving Usability and Responsiveness, Israel, 1978 - Academic Press, New York. (An earlier version of this paper with the same title is available as Monografia em Ciência da Computação no. 6/78, Dept. de Informática, Pontifícia Universidade Católica, Rio de Janeiro, Brasil).

3.1 Introduction

The data base definition language described in this chapter may be used both for the definition of a new data base and for modifying the structure of an existing data base.

Instructions for the creation or modification of the data base structure are presented as a sequence of commands, some of which require sub-commands to specify in more detail the actions required.

Examples:

```
i)  DELETE CLASS WORKMAN
```

- to remove the class WORKMAN from the data base

```
ii) AMEND CLASS EMPLOYEE
    BEGIN
      ADD INT INSURANCE_NUM
      DELETE INT AGE
    END
```

- this example of the AMEND CLASS command includes two sub-commands; the first defines a new primitive field, and the second removes an existing primitive field.

Execution of the commands may result not only in changes to the class definitions of the data base, but also in modifications to the entries themselves. For example when a class definition C is deleted, all entries in that class must be deleted; and the removal of C also implies that any link field definition whose partner class is C must also be removed, which may affect further entries in other classes. Particularly expensive alterations are those affecting keys and indices: the alteration of a class key means that all the entries of that class must be physically re-ordered.

Sections 3.3 to 3.15 define the data base definition language. The syntax of the commands is given using a variant of BNF (described in section 3.2), together with some "rewriting rules" which show how certain command sequences may be written in an abbreviated form. A summary of the syntax appears in section 3.16.

3.2 The meta-language

The meta-language used to define the syntax of the commands is a more concise variant of BNF (Backus Naur Form). This is informally described below.

3.2.1 Syntactic constructs

The names of syntactic constructs (i.e. of the objects being defined) are composed of lower-case letters and hyphens ('-'), and are not enclosed in angle brackets.

Examples:

```
classid
add-class-comm
```

3.2.2 Terminal symbols

Terminal symbols (i.e. the basic symbols of the language being defined) appear as upper-case words or as the symbols themselves.

Examples:

```
ADD
CLASS
;
```

3.2.3 Production rules

In a production rule, the symbol '->' is used to separate the name of the construct from its definition, and vertical bars ('|') are used to separate alternative definitions.

Example:

```
domestic-animal -> cat | dog | canary
```

3.2.4 Bracketing

Angle brackets ('<' and '>') are used as meta-parentheses so that definitions may be factored; and square brackets ('[' and ']') are used to enclose parts of a definition which may be omitted.

Example:

```
        cat -> head body [ < short-tail | long-tail > ]
```

is equivalent to:

```
        cat -> head body |
               head body short-tail |
               head body long-tail
```

3.2.5 Repetition

The form 'x S ...' is used to indicate a sequence of one or more 'x's separated by 'S's; that is, 'x S ...' stands for 'x|xSx|xSxSx| ...'. 'x' and 'S' may be the name of a syntactic construct, a terminal symbol, or a bracketted sequence.

Examples:

```
        stt-list -> stt ; ...
```

- a 'stt-list' is a sequence of one or more 'stt's separated by semi-colons.

```
        stt-list -> < [ label ] stt > ; ...
```

- in this case, each statement in the list may optionally be preceded by a label.

3.2.6 Subscripted constructs

The name of a syntactic construct appearing within a definition is sometimes followed by a digit; this is simply a subscript to aid identification of that particular occurrence and does not form part of the construct name.

Example:

```
        rename-stt -> RENAME id1 as id2
```

- both 'id1' and 'id2' are described by the production rule for the construct 'id'.

3.2.7 Primitive constructs

The definition of certain basic constructs is enclosed in double asterisks; this is a natural language definition.

Example:

```
integer ->
  ** a sequence of between 1 and 8 decimal digits **
```

3.3 Basic constructions, comments and keywords

We first define some of the basic elements of the language.

3.3.1 Names and types

```
classid -> name
primid -> name
groupid -> name
variantid -> name
linkid -> name
markid -> name
indexid -> name

name -> ** a sequence of letters, digits, periods ('.')
           and underlines ('_'), starting with a letter
           **
```

Examples:

```
PROFESSOR
HRS_PER_WEEK
PAY.2
```

```
modname -> ** implementation dependent; up to 8 national
              characters for the 370 implementation **
```

Examples:

```
COMPFNCS
$CF7
```

```
funcname -> ** implementation dependent; a sequence of up
               to 7 letters or digits, starting with a
               letter, for the 370 implementation **

user-type -> BOOL | STRING | INT | REAL | DATE |
             char | special

  char -> ** 'CHAR-n' where 1 <= n <= 256 **

  special -> ** implementation dependent **
```

```
    basic-type -> BOOL | WORD | STRING | byte

      byte -> ** 'BYTE-n' where 1 <= n <= 256 **
```

Examples:

```
    BOOL
    CHAR-5
    BYTE-12
    STRING
```

3.3.2 Constants

```
    string -> ** a sequence of up to 255 characters enclosed
                 in single (') or double (") quotes **

    value -> int-value | hex-value

      int-value -> ** a possibly signed sequence of decimal
                      digits **

      hex-value -> ** up to 8 hexadecimal digits preceded by
                      'X' **
```

Notes:

i) The asterisk ('*') is used as an escape character within strings:

```
     *'  stands for '
     *"  stands for "
     **  stands for *
     *N  stands for 'newline'
     *S  stands for 'space'
```

and the sequence '*<newline><spaces>*' is ignored, so that a string may be split over several lines. Other escape combinations are undefined.

Examples:

```
     "ABC*"*SD"  represents the same string as 'ABC" D'

     "AB*ND" is a string containing the newline character.

     "ABC*
      *D*
            *EF"  represents the same string as "ABCDEF".
```

ii) A hex-value is right-justified.

Example:

15 and XF represent the same value.

3.3.3 Brackets

The constructs 'bra' and 'ket' are always paired within a definition. They may be replaced by '(' and ')' or by 'BEGIN' and 'END'.

Example:

The rule:

```
amend-prim-comm ->
  AMEND PRIM primid bra amend-prim-subcomm ; ... ket
```

stands for the two alternatives:

```
amend-prim-comm ->
  AMEND PRIM primid ( amend-prim-subcomm ; ... ) |
  AMEND PRIM primid BEGIN amend-prim-subcomm ; ... END
```

3.3.4 Layout

Layout is free format, and extra spaces may be inserted between items to improve legibility; on the other hand, spaces are not required between items which are already correctly delimited by some other symbol.

Examples:

i) ADD CLASS PERSON BEGIN STRING NAME, ADDRESS END

ii) ADD CLASS PERSON(STRING NAME , ADDRESS)

iii) but not:

ADDCLASS PERSON ....

A comment may appear wherever a space may appear, and is introduced by '//' and terminated by 'newline' (i.e. the rest of the line is ignored).

Example:

```
ADD CLASS PERSON    // used to be called PEOPLE
BEGIN
  STRING NAME,      // used to be called SURNAME
         ADDRESS
END
```

The semicolons separating the commands within a sequence may be omitted if each command is written on a separate line.

Example:

```
ADD CLASS C ( INT X, Y; STRING Z)
```

may be written as:

```
ADD CLASS C (
  INT X, Y
  STRING Z  )
```

The exact rule is that any newline is treated as a semicolon if:

i) a semicolon is syntactically correct at this point,

and

ii) it is not immediately followed by 'BEGIN', '(', 'END', ')', ',' or another newline.

Any other newline is treated as a space.

3.3.5 Keywords

The following table lists synonyms for various keywords that appear in the language. None of the keywords are reserved words, and so there are no restrictions on the choice of names for classes and fields.

Keyword	Synonyms
ADD	NEW
MODULE	MODULES
CLASS	CLASSES
LINK	LINKS
PRIM	PRIMS
GROUP	GROUPS
MARK	MARKS
INDEX	INDICES, INDEXES
COMPFUNC	COMPFUNCS
VARIANT	VARIANTS, UNION, ONEOF
AS	=
IS	=
ON	IS, =, BY
DESCRIPTION	COMMENT
OF	IN

3.4 Rewriting rules

Certain sequences or forms of commands may be rewritten in abbreviated formats, and this section describes the possible transformations. The rewriting rules (if any) which may be applied to a particular command are indicated by a character in brackets to the left of the production rule for that command.

Example:

(R) delete-mod-comm -> DELETE MODULE modname

'(R)' indicates that the rewriting rule for repetition may be applied to consecutive occurrences of this command.

3.4.1 Repeatable commands (R)

The general form of the definition of a "repeatable command" is:

(R) construct -> keywords argument

where 'construct' is the name of the construct, 'keywords' is a sequence of one or more terminal symbols, and 'argument' is the rest of the definition.

A consecutive sequence of occurrences of the construct of the form:

```
keywords argument1;
keywords argument2;
    ......
keywords argumentn
```

may be rewritten as:

```
keywords argument1, argument2, ...... argumentn
```

<u>Example</u>:

```
(R)  delete-mod-comm -> DELETE MODULE modname
```

The sequence:

```
DELETE MODULE M1; DELETE MODULE M2;
DELETE MODULE M3
```

may be rewritten as:

```
DELETE MODULE M1, M2, M3
```

3.4.2 <u>Complex commands (C)</u>

The general form of the definition of a "complex command" is:

```
(C) construct -> keywords argument [ bra subcomm ; ... ket ]
```

where 'construct' is the name of the construct and 'keywords' is a sequence of one or more terminal symbols. 'subcomm' and 'argument' are, respectively, the parts of the definition which describe the possible subcommands, and the item to which they apply.

One (but not both) of the following rewriting rules may be applied:

i) An occurrence of the construct of the form:

```
keywords argument bra subcomm ket
```

may be rewritten as:

```
keywords argument subcomm
```

ii) A consecutive sequence of occurrences of the construct of the form:

```
keywords argument1 [ bra subcomm ; ... ket ];
keywords argument2 [ bra subcomm ; ... ket ];
                  ......
keywords argumentn [ bra subcomm ; ... ket ]
```

may be rewritten as:

```
keywords argument1 [ bra subcomm ; ... ket ],
         argument2 [ bra subcomm ; ... ket ],
                     ......
         argumentn [ bra subcomm ; ... ket ]
```

<u>Examples</u>:

```
(C)  amend-class-comm ->
       AMEND CLASS classid
         bra amend-class-subcomm ; ... ket
```

i) The command:

```
AMEND CLASS C1 (DELETE PRIM AGE)
```

may be rewritten as:

```
AMEND CLASS C1 DELETE PRIM AGE
```

ii) The sequence:

```
AMEND CLASS C1 (DELETE PRIM P1, P2);
AMEND CLASS C2 (ADD PRIM P1; DELETE PRIM P2)
```

may be rewritten as:

```
AMEND CLASS C1 (DELETE PRIM P1, P2),
            C2 (ADD PRIM P1; DELETE PRIM P2)
```

but not as:

```
AMEND CLASS C1 DELETE PRIM P1, P2,
            C2 (ADD PRIM P1; DELETE PRIM P2)
```

3.5 Contexts and commands

This section includes information about the overall structure of the commands, and introduces various conventions that are used in later sections.

3.5.1 Contexts

The commands of the data base definition language are of two kinds: "basic" commands, which perform some action within the current context, and "compound" commands which may first perform some action within the current context and then derive a new context in which further commands (known as "subcommands" of the compound command) are executed.

Each "context" is composed of a number of "components" as follows:

i) The components of the "initial" context are:

- class definitions
- comparison function module specifications

for the data base.

ii) The components of a "class" context are:

- variant, primitive field, group field, link field, key and index definitions
- user description and miscellaneous information fields

for a particular class.

iii) The components of a "variant" context are:

- variant, primitive field, group field and link field definitions
- user description and miscellaneous information fields

for a particular variant.

iv) The components of a "primitive" context are:

- user description and miscellaneous information fields
- user occurrence and implementation information
- user type and basic type

for a particular primitive field.

v) The components of a "link" context are:

- primitive field, group field, mark field, key and index definitions
- user description and miscellaneous information fields

- user occurrence and implementation information
- its kind (uni-link or multi-link)

for a particular link field.

vi) The components of a "group" context are:

- primitive field, group field, key and index definitions
- user description and miscellaneous information fields
- user occurrence and implementation information

for a particular group field.

vii) The components of a "mark" context are:

- user description and miscellaneous information fields
- user occurrence information

for a particular mark field.

viii) The components of a "key" context are:

- key field definitions (including any comparison functions)

for the key.

ix) The components of an "index" context are:

- index field definitions (including any comparison functions)

for a particular index.

x) The components of a "module" context are:

- the names of the comparison functions

within a particular module.

Most commands also specify a "subject" which is a component of the context in which the command is executed. The command may create, modify or delete its subject.

Examples:

i) DELETE CLASS C

This is a basic command which may only appear in the initial context. Its subject is the class definition for C which is removed from the context.

ii) AMEND CLASS C (DELETE PRIM X)

The 'AMEND CLASS' command is a compound command which may only appear in the initial context. Its subject is the class definition for C from which a corresponding class context is created. The subcommands of 'AMEND CLASS' are then executed within the new context.

In this example, the subcommand is the basic command 'DELETE PRIM X' which, when executed within a class context, locates its subject, the primitive field definition for X, and removes it from the context.

Thus the complete effect of this command is to remove the primitive field definition for X from the class C.

3.5.2 Generic descriptions

Many of the commands available fall into one of four categories of "creation", "amendment", "deletion" and "renaming". The semantics of all the commands in one category are similar, the details depending on the particular subject and the context of execution. To avoid needless repetition in later sections, the semantics of each category are defined here.

In the generic descriptions below, occurrences of <SUBJECT>, <subjectid> and <subject> must be consistently replaced by one of the following seven possibilities:

<SUBJECT>	<subjectid>	<subject>
CLASS	classid	class
VARIANT	variantid	variant
PRIM	primid	prim
LINK	linkid	link
GROUP	groupid	group
MARK	markid	mark
INDEX	indexid	index
MODULE	modname	mod

a) <u>Commands for creation</u>

```
*(C)  ADD <SUBJECT> <subjectid>
          [ bra add-<subject>-subcomm ; ... ket ]
```

or

```
*(C)  <SUBJECT> <subjectid>
          [ bra add-<subject>-subcomm ; ... ket ]
```

The subject of the command is the <subject> definition <subjectid> which is created and added as a new component to the current context. An error occurs if the subject is already present in the current context, or if the naming rules (section 2.9) would be broken by its addition. The subject is then used to define a new <subject> context in which each 'add-<subject>-subcomm' is executed in turn.

The second form of the command (in which the keyword ADD is absent) is used when the command is itself a subcommand of another enclosing creation command.

<u>Example</u>:

```
*(C)  ADD CLASS classid
          [ bra add-class-subcomm ; ... ket ]
```

The current context (which must be the initial context) is extended by the addition of a new definition for the class 'classid ; each 'add-class-subcomm' is then executed within a new class context derived from this definition.

b) <u>Commands for amendment</u>

```
*(C)  AMEND <SUBJECT> <subjectid>
          bra amend-<subject>-subcomm ; ... ket
```

The subject of the command is the <subject> definition <subjectid> in the current context. A new <subject> context is derived from this component of the current context, and each 'amend-<subject>-subcomm' is executed within it in turn.

An error is indicated if the subject is not present in the current context.

Example:

```
*(C)  AMEND GROUP groupid
         bra amend-group-subcomm ; ... ket
```

The subject is the group definition for the group 'groupid'. (The current context might be a class, group, variant or link context.) Each 'amend-group-subcomm' is executed in the group context provided by the subject.

c) Commands for deletion

```
*(R)  DELETE <SUBJECT> <subjectid>
```

The subject of the command is the <subject> definition <subjectid>. This definition is removed from the current context.

An error occurs if the subject is not present in the current context.

Example:

```
*(R)  DELETE VARIANT variantid
```

The subject is the variant definition for the variant 'variantid'. (The current context might be a class or variant context.) The variant definition is deleted.

d) Commands for renaming

```
*(R)  RENAME <SUBJECT> <subjectid>1 AS <subjectid>2
```

The subject of the command is the <subject> definition <subjectid>1. The definition is renamed <subjectid>2.

An error is given if the subject is not found in the current context, or if renaming would break the naming rules of section 2.9.

Example:

```
*(R)  RENAME PRIM primid1 AS primid2
```

The subject is the definition for the primitive field 'primid1'. (The current context might be a class, variant, group or link context.) The primitive field is renamed as 'primid2'.

3.5.3 Format of sections 3.6 to 3.15

Not all of the commands that may be derived from the above generic descriptions actually exist (for example, there is no 'DELETE LINK linkid' command), and the contexts in which any particular command may appear are restricted (for example, 'DELETE PRIM primid' may not appear in the initial context).

Sections 3.6 to 3.15 define the syntax and semantics of the data base definition language, and the possible commands and their permitted contexts may be deduced from the production rules. Note that a production rule for a command whose semantics are correctly described by the corresponding generic description above is preceded by an asterisk.

Each section groups together the commands associated with one or two particular components of the class definitions (e.g. section 3.12 describes commands for the creation, amendment, deletion and renaming of primitive and group fields), and normally includes the following subsections:

a) Permitted contexts

- a list of the contexts in which the command may appear. This list may be deduced from the production rules, but is included at the head of each section for reference.

b) Syntax

- one or more production rules, possibly separated by short semantic descriptions enclosed between the delimiters '|*' and '*|'.

c) Notes

- extra remarks about the syntax.

d) Semantics

- explanation of the effect of the command(s) described in an immediately preceding 'Syntax' subsection.

e) Examples

- examples of constructs defined in preceding syntax subsections. Note that the rewriting rules and synonyms described in section 3.3 are used in the examples.

Subsections (b) to (e) may appear several times within one section.

3.5.4 Choice of construct names

To increase intelligibility, the names of the syntactic categories defining the various constructs of the language have been chosen according to the following scheme:

i) A name ending in '-subcomm' describes the set of commands which may appear as subcommands of some higher level command.

Example:

'add-class-subcomm' defines the commands which may appear as subcommands of the 'add-class-comm'.

ii) Names starting with 'add-', 'amend-', 'delete-' or 'rename-' and finishing with '-comm' represent individual commands to perform some specific action (creation, amendment, deletion or renaming respectively). They may be basic or compound commands.

Examples:

'add-class-comm' defines the compound command that creates a new class.

'rename-group-comm' defines the basic command that renames some group within a class, variant, link or group field.

iii) Other names ending in '-comm' are used to represent sets of individual commands for convenience.

Example:

A 'descr-comm' may be an 'add-descr-comm' or a 'delete-descr-comm'.

iv) Names ending with '-def' are used to designate the commands which occur as subcommands of some enclosing 'add- ... -comm' - i.e. which occur within the context of a newly-created item. They differ from other commands in that they do not require a keyword to specify their action.

Examples:

i) 'prim-def' defines the command that specifies a primitive field of a newly-created class, variant, link or group field.

ii) 'primgroup-def' is either a 'prim-def' or a 'group-def'.

3.6 The program

Syntax:

```
program -> comm ; ...

comm -> class-comm |
        link-comm |
        module-comm
```

Semantics:

Each 'comm' is executed in turn in the initial context.

3.7 Class commands

Permitted context: initial

Syntax:

```
     class-comm -> add-class-comm |
                   amend-class-comm |
                   delete-class-comm |
                   rename-class-comm

*(C) add-class-comm ->
          ADD CLASS classid
            [ bra add-class-subcomm ; ... ket ]

       add-class-subcomm -> descrdata-def |
                            keyindex-def |
                            primgroup-def |
                            variant-def
```

```
*(C) amend-class-comm ->
          AMEND CLASS classid
            bra amend-class-subcomm ; ... ket

        amend-class-subcomm -> descrdata-comm |
                               keyindex-comm |
                               primgroup-comm |
                               variant-comm |
                               linkfield-comm

*(R) delete-class-comm -> DELETE CLASS classid

*(R) rename-class-comm ->
          RENAME CLASS classid1 AS classid2
```

Examples:

```
  i)    ADD CLASS ROOM
        BEGIN
          INT ROOMNUM, AREA
          KEY IS ROOMNUM
        END

  ii)   AMEND CLASS ROOM DELETE PRIM AREA

  iii)  DELETE CLASSES ROOM, BUILDING

  iv)   RENAME CLASS CUPBORD AS CUPBOARD
```

3.8 Link commands

Permitted context: initial

Syntax:

```
     link-comm -> add-link-comm |
                  delete-link-comm

  (R) add-link-comm ->
          ADD LINK ( linkfield-def1 , linkfield-def2 )
            BETWEEN classorvar1 AND classorvar2

        classorvar -> classid |
                      variantid OF classorvar
```

```
linkfield-def ->
    linkid [ bra add-linkfield-subcomm ; ... ket ]

  add-linkfield-subcomm -> descrdata-def |
                           keyindex-def |
                           primgroup-def |
                           mark-def |
                           linkfield-qualifiers
```

Semantics:

Each 'add-link-comm' defines a new link between the class or variant 'classorvar1' and the class or variant 'classorvar2'. 'linkfield-def1' specifies the creation of the link field in 'classorvar1' and 'linkfield-def2' specifies the partner link field in 'classorvar2'.

Each 'add-linkfield-subcomm' within a 'linkfield-def' is applied to the newly created link field in turn; in the absence of contrary information (specified by 'linkfield-qualifiers' - see section 3.13.1), the link field will be an optional, common multi-link field.

If no key is specified for the link field (by a 'key-def') then the key will be the same as that of its partner class (and the field will be keyless if the partner class has no key).

Syntax:

```
(R) delete-link-comm ->
        DELETE LINK ( linkid1 , linkid2 )
          BETWEEN classorvar1 AND classorvar2
```

Semantics:

The link fields 'linkid1' in class or variant 'classorvar1' and 'linkid2' in class or variant 'classorvar2' must be partner link fields. They are both deleted.

Notes:

'FROM', 'TO' may be used in place of 'BETWEEN', 'AND' in either of these commands.

Examples:

```
i)   ADD LINK ( ROOMS, BUILDING )
       BETWEEN BUILDING AND ROOM

ii)  ADD LINK ( CURRPROJS, EMPLS(MARK LEADER))
       BETWEEN EMPLOYEE AND PROJECT
```

```
  iii) DELETE LINK (MANAGER, EMPLS)
                      FROM EMPLOYEE TO EMPLOYEE,
                    (AUDITPROJ, AUDITOR)
                      BETWEEN EMPLOYEE AND PROJECT

  iv)  ADD LINK ( STUDS, SUPERVISOR(SINGLE) )
         BETWEEN STUDENT OF DEPT_MEMBER
             AND FULL_TIME_PROF IN PROFESSOR IN DEPT_MEMBER
```

3.9 Module commands

Permitted context: initial

Notes:

Any user-provided comparison functions must be pre-compiled and stored in modules, which are later loaded by the nucleus when a data base is opened. Each module may include more than one comparison function, and the purpose of the module commands is to inform the package of the names of the modules and of the comparison functions contained within them.

No two module names or comparison function names may be the same, although the name of a comparison function may be the same as that of a module.

Syntax:

```
     module-comm -> add-mod-comm |
                    amend-mod-comm |
                    delete-mod-comm |
                    rename-mod-comm

*(C) add-mod-comm ->
         ADD MODULE modname
           [ bra add-mod-subcomm ; ... ket ]

       add-mod-subcomm -> funcname , ...

         |*  An 'add-mod-subcomm' specifies the names of the
             comparison functions which belong to the module
             *|
*(C) amend-mod-comm ->
        AMEND MODULE modname
          bra amend-mod-subcomm ; ... ket

      amend-mod-subcomm -> add-func-comm |
                           delete-func-comm |
                           rename-func-comm

    (R) add-func-comm -> ADD funcname

    (R) delete-func-comm -> DELETE funcname

          |*  A comparison function may only be deleted if it
              is not in use; that is, provided it does not
              take part in any key or index field definition
              *|

    (R) rename-func-comm -> RENAME funcname1 AS funcname2

         |*  These subcommands allow the addition, deletion,
             or renaming of functions in the list of contents
             of the module  *|

*(R) delete-mod-comm -> DELETE MODULE modname

       |*  A module may only be deleted when none of its
           comparison functions are being used  *|

*(R) rename-mod-comm -> RENAME MODULE modname1 AS modname2
```

<u>Examples:</u>

```
  i)   ADD MODULE M1 (F1, F2, F3),
                  M2 (G1, G2)

  ii)  AMEND MODULE M1 DELETE F1, F2

  iii) AMEND MODULE M2
       BEGIN
         ADD F1, F2
         RENAME G1 AS F3,
                G2 = F4
       END

  iv)  RENAME MODULE M3 AS M4
```

3.10 User description and miscellaneous field commands

Permitted contexts: class, variant, primitive, group, link, mark

Syntax:

```
    descrdata-def -> descr-def |
                     miscdata-def

    descrdata-comm -> descr-comm |
                      miscdata-comm

    descr-def -> DESCRIPTION IS string

      |*  sets 'string' in the user description field  *|

    descr-comm -> add-descr-comm |
                  delete-descr-comm

    add-descr-comm -> ADD descr-def

      |*  replaces any existing user description by the new
          one  *|

    delete-descr-comm -> DELETE DESCRIPTION

      |*  removes any existing user description  *|
```

Examples:

```
  i)   DESCRIPTION IS "one entry for each employee"

  ii)  ADD COMMENT = 'Surname only'
```

Syntax:

```
    miscdata-def ->
        MISCDATA IS < value | ( value , ... ) >

      |*  sets the specified 'value'(s) as elements of the
          user miscellaneous field  *|

    miscdata-comm -> add-miscdata-comm |
                     delete-miscdata-comm

    add-miscdata-comm -> ADD miscdata-def

      |*  replaces any existing user miscellaneous
          information by the 'value'(s) specified  *|
     delete-miscdata-comm -> DELETE MISCDATA

       |*  removes any existing user miscellaneous information
           *|
```

<u>Examples</u>:

```
  i)   DELETE MISCDATA

  ii)  MISCDATA = 18

  iii) ADD MISCDATA = (19, XCO, -3)
```

3.11 <u>Key and Index commands</u>

<u>Permitted contexts</u>: class, group, link

<u>Syntax</u>:

```
     keyindex-def -> key-def |
                     index-def

     keyindex-comm -> key-comm |
                      index-comm

     key-def -> KEY IS key-spec |
                NOKEY

       key-spec -> simple-key-spec |
                   ( simple-key-spec , ... )

         simple-key-spec -> primid [ ( funcname ) ]
```

<u>Semantics</u>:

The key fields of a key are defined in order by a sequence of 'simple-key-spec's; each 'simple-key-spec' names the corresponding key field 'primid' and possibly specifies a comparison function 'funcname'. The 'NOKEY' option of 'key-def' indicates that the current context is keyless, and is assumed in the absence of any 'key-def'.

<u>Examples</u>:

```
  i)   KEY IS AGE

  ii)  KEY IS (SURNAME, CHRISTIAN_NAMES)

  iii) KEY IS SALARY(REVINT)
```

Syntax:

```
     key-comm -> add-key-comm |
                 amend-key-comm |
                 delete-key-comm

     add-key-comm -> ADD KEY IS key-spec

       |*  specifies a key; may only be used if no key has yet
           been defined for the current context  *|

     amend-key-comm ->
         AMEND KEY < bra amend-keyindex-subcomm ; ... ket |
                     amend-keyindex-subcomm >

       |*  each 'amend-keyindex-subcomm' is applied to the key
           in turn  *|

     delete-key-comm -> DELETE KEY |
                        NOKEY

       |*  any existing key is removed  *|
```

Examples:

```
  i)   ADD KEY = AGE

  ii)  AMEND KEY ADD BIRTH_DATE
```

Syntax:

```
 (R) index-def -> INDEX index-spec

       index-spec -> indexid ON key-spec

       |*  'key-spec' defines the index fields of a new index
           'indexid'  *|
```

Example:

```
       INDEX BYAGE = AGE,
             BYNAME = (SURNAME,
                       CHRISTIAN_NAMES,
                       ID_NUM(IDSORT))
```

Syntax:

```
     index-comm -> add-index-comm |
                   amend-index-comm |
                   delete-index-comm |
                   rename-index-comm
```

```
 (R) add-index-comm -> ADD INDEX index-spec

       |* specifies a new index  *|

*(C) amend-index-comm ->
        AMEND INDEX indexid
          bra amend-keyindex-subcomm ; ... ket

*(R) delete-index-comm -> DELETE INDEX indexid

*(R) rename-index-comm -> RENAME INDEX indexid1 AS indexid2
```

Examples:

```
  i)  ADD INDEX BYAGE = AGE

  ii) AMEND INDEX BYNAME DELETE ID_NUM

  iii) RENAME INDEX I1 AS INDEX1
```

3.11.1 Key and Index field commands

Permitted contexts: key, index

Syntax:

```
     amend-keyindex-subcomm -> add-keyindexfield-comm |
                                elete-keyindexfield-comm |
                               add-compfunc-comm |
                               delete-compfunc-comm

     add-keyindexfield-comm -> ADD key-spec

       |* the fields and comparison functions specified by
          'key-spec' are appended as additional secondary key
          or index fields  *|

     delete-keyindexfield-comm ->
         DELETE < primid | ( primid , ... ) >

       |* Suppose the current context is composed of the
          sequence of key or index fields p1, p2, ... pn;
          then the sequence '<primid|(primid,...)>' must
          specify some subset of the secondary fields
          pj, ... pn (j>=2). These fields are removed from
          the context (so that it is afterwards defined by
          the sequence p1, ... p(j-1))  *|
(R) add-compfunc-comm -> COMPFUNC IS compfunc-spec

      compfunc-spec ->
          funcname FOR < primid | ( primid , ... ) >

          |*  'funcname' is defined as the comparison function
              for the key or index field(s)
              '<primid|(primid,.. )>', replacing any previously
              defined comparison function(s)  *|

(R) delete-compfunc-comm -> DELETE COMPFUNC FOR primid

          |*  Any comparison function defined for the key or
              index field 'primid' is deleted (thus reinstating
              the default ordering for the field)  *|
```

Examples:

```
  i)   ADD (P4, P5(F1))

  ii)  DELETE (P3, P4, P5)

  iii) COMPFUNC IS FINT FOR (P7, P8),
                   FREAL FOR (P1, P2)
```

3.12 Primitive and group field commands

Permitted contexts: class, variant, group, link

Syntax:

```
    primgroup-def -> prim-def |
                     group-def

    primgroup-comm -> prim-comm |
                      group-comm

    prim-def ->
        user-type < [ ( basic-type ) ] primid
          [ bra add-prim-subcomm ; ... ket ] > , ...

      add-prim-subcomm -> descrdata-def |
                          primgroup-qualifiers
```

Semantics:

Each 'prim-def' specifies the creation of one or more new primitive fields 'primid'; 'user-type' gives the user type of each field. The basic type for a field 'primid' is given by the first occurrence of '[(basic-type)]' to the left of 'primid' in the 'prim-def'; if no such occurrence is found, a default is chosen according to the user type as follows:

User type	Default basic type
BOOL	BOOL
STRING	STRING
INT	WORD
REAL	BYTE-8
DATE	BYTE-2
CHAR-n	BYTE-n (1<=n<=256)

'add-prim-subcomm's are used to specify any user or implementation information associated with the primitive field. If no user occurrence information is specified, the field is assumed to be mandatory ('MAND') if it is a primary key field for a class or group, and is assumed to be optional ('OPT') otherwise. If no implementation information is given, the field is assumed to be common ('COMMON').

Examples:

```
  i)   INT X, Y, Z             //all WORD fields

  ii)  INT (BYTE-1) X, Y, Z  //all BYTE-1 fields

  iii) INT X, (BYTE-1) Y, Z, (BYTE-2) A
          //X is WORD, Y and Z are BYTE-1, A is BYTE-2

  iv)  STRING (BYTE-8) ID_NUM (MAND),
              (STRING) EX_WIFE (OPT, RARE)

  v)   INT (BYTE-2) SALARY (DESCRIPTION="Pounds per year"
                            OPT
                            COMMON)
```

Syntax:

```
    prim-comm -> add-prim-comm |
                 amend-prim-comm |
                 delete-prim-comm |
                 rename-prim-comm

 (R) add-prim-comm -> ADD PRIM prim-def

        |*  the new primitive fields specified by 'prim-def'
            are added to the current context  *|
*(C) amend-prim-comm ->
        AMEND PRIM primid bra amend-prim-subcomm ; ... ket

      amend-prim-subcomm -> descrdata-comm |
                            primgroup-qualifiers |
                            amend-type-comm

        amend-type-comm ->
            TYPE IS < [ user-type ] ( basic-type ) |
                      user-type >

          |*  This command alters the user type and/or the
              basic type of the primitive field in whose
              context it appears  *|

*(R) delete-prim-comm -> DELETE PRIM primid

      |*  a primitive field may not be deleted if it is a key
          or index field  *|

*(R) rename-prim-comm -> RENAME PRIM primid1 AS primid2
```

<u>Examples</u>:

```
  i)    AMEND PRIM EX_WIFE
        BEGIN
          COMMON
          DESCRIPTION = "Maiden name"
          TYPE IS (BYTE-20)
        END

  ii)   DELETE PRIM P1, P2, P3

  iii)  ADD PRIM STRING S1, S2, DATE DATE, INT X, Y, Z
```

<u>Syntax</u>:

```
*(C) group-def ->
        GROUP groupid [ bra add-group-subcomm ; ... ket ]

      add-group-subcomm -> descrdata-def |
                           keyindex-def |
                           primgroup-def |
                           primgroup-qualifiers

      |*  If no user occurrence information is specified, the
          newly created group is assumed to be optional, and
          if no implementation information is given it is
          assumed to be common  *|
```

Examples:

```
i)   GROUP CHILDREN
     BEGIN
       OPT, RARE
       STRING NAME
       KEY IS NAME
       GROUP TOYS STRING DESCRIPTION
     END
```

Syntax:

```
     group-comm -> add-group-comm |
                   amend-group-comm |
                   delete-group-comm |
                   rename-group-comm

*(C) add-group-comm ->
        ADD GROUP groupid
          [ bra add-group-subcomm ; ... ket ]

*(C) amend-group-comm ->
        AMEND GROUP groupid
          bra amend-group-subcomm ; ... ket

       amend-group-subcomm -> descrdata-comm |
                              keyindex-comm |
                              primgroup-comm |
                              primgroup-qualifiers

*(R) delete-group-comm -> DELETE GROUP groupid

*(R) rename-group-comm -> RENAME GROUP groupid1 AS groupid2
```

Examples:

```
i)   AMEND GROUP CHILDREN
     BEGIN
       COMMON
       AMEND GROUP TOYS ADD PRIM INT AGE
     END

ii)  RENAME GROUPS G1 AS G2,
                   G3 AS G4
```

3.12.1 Qualifiers for primitive or group fields

Permitted contexts: primtive, group

Syntax:

```
    primgroup-qualifiers -> primgroup-qualifier , ...

       primgroup-qualifier -> OPT | MAND | RARE | COMMON
```

Notes:

The set of 'primgroup-qualifier's specified by all 'primgroup-qualifiers' for the current context should include at most one of 'OPT', 'MAND' and at most one of 'RARE', 'COMMON'.

Semantics:

User occurrence information is specified by 'OPT' (optional field) or 'MAND' (mandatory field).

Implementation information is specified by 'RARE' (the field is not usually set) or 'COMMON'.

The user occurrence or implementation information for the primary key field may not be altered.

Examples:

i) OPT

ii) MAND, COMMON

3.13 Variant commands

Permitted contexts: class, variant

Syntax:

```
*(C) variant-def ->
        VARIANT variantid
          [ bra add-variant-subcomm ; ... ket ]

       add-variant-subcomm -> descrdata-def |
                              primgroup-def |
                              variant-def
```

Example:

```
      VARIANTS STUDENT (INT CREDITS),

               SECRETARY (INT WAGES),

               PROFESSOR
               BEGIN
                 INT SALARY
                 VARIANT PART_TIME_PROF
                 VARIANT FULL_TIME_PROF
               END
```

Syntax:

```
     variant-comm -> add-variant-comm |
                     amend-variant-comm |
                     delete-variant-comm |
                     rename-variant-comm

*(C) add-variant-comm ->
         ADD VARIANT variantid
           [ bra add-variant-subcomm ; ... ket ]

*(C) amend-variant-comm ->
         AMEND VARIANT variantid
           bra amend-variant-subcomm ; ... ket

       amend-variant-subcomm -> descrdata-comm |
                                primgroup-comm |
                                variant-comm |
                                linkfield-comm

*(R) delete-variant-comm -> DELETE VARIANT variantid

*(R) rename-variant-comm ->
         RENAME VARIANT variantid1 AS variantid2
```

Examples:

```
  i)   AMEND VARIANT PROFESSOR
         AMEND VARIANT PART_TIME_PROF
           ADD PRIM INT HRS_PER_WEEK

  ii)  DELETE VARIANT SECRETARY
```

3.14 Link field commands

Permitted contexts: class, variant

Syntax:

```
     linkfield-comm -> amend-linkfield-comm |
                       rename-linkfield-comm

*(C) amend-linkfield-comm ->
         AMEND LINK linkid
           bra amend-linkfield-subcomm ; ... ket

       amend-linkfield-subcomm -> descrdata-comm |
                                  keyindex-comm |
                                  primgroup-comm |
                                  mark-comm |
                                  linkfield-qualifiers

*(R) rename-linkfield-comm -> RENAME LINK linkid1 AS linkid2
```

Examples:

```
  i)   AMEND LINK PROJECTS (
         ADD MARK MAJOR
         KEY IS PROJNUM     )

  ii)  RENAME LINK PROJECTS AS PROJS
```

3.14.1 Qualifiers for link fields

Permitted context: link

Syntax:

```
     linkfield-qualifiers -> linkfield-qualifier , ...

       linkfield-qualifier -> OPT | MAND |
                              RARE | COMMON |
                              SINGLE | MULTIPLE
```

Notes:

The set of 'linkfield-qualifier's specified by all 'linkfield-qualifiers' for the current context should include at most one of 'OPT', 'MAND', at most one of 'RARE', 'COMMON', and at most one of 'SINGLE', 'MULTIPLE'.

Semantics:

User occurrence information is specified by 'OPT' (optional) or 'MAND' (mandatory).

Implementation information is specified by 'RARE' (the field is not usually set) or 'COMMON'.

The kind of link field is specified by 'SINGLE' (a uni-link field) or 'MULTIPLE' (a multi-link field).

Examples:

i) OPT, RARE

ii) SINGLE, OPT, COMMON

3.15 Mark field commands

Permitted context: link

Syntax:

```
*(C) mark-def ->
          MARK markid [ bra add-mark-subcomm ; ... ket ]

        add-mark-subcomm -> descrdata-def |
                            mark-qualifier

        |*  in the absence of any 'mark-qualifier' the created
            field is assumed to be optional  *|
```

Examples:

i) MARKS OWNER, MAJORUSER (OPT)

ii) MARK LARGEST MAND

Syntax:

```
      mark-comm -> add-mark-comm |
                   amend-mark-comm |
                   delete-mark-comm |
                   rename-mark-comm

*(C) add-mark-comm ->
          ADD MARK markid [ bra add-mark-subcomm ; ... ket ]
```

```
*(C) amend-mark-comm ->
        AMEND MARK markid bra amend-mark-subcomm ; ... ket

       amend-mark-subcomm -> descrdata-comm |
                             mark-qualifier

*(R) delete-mark-comm -> DELETE MARK markid

*(R) rename-mark-comm -> RENAME MARK markid1 AS markid2
```

__Examples:__

i) AMEND MARK LARGEST OPT

ii) DELETE MARKS M1, M2

3.15.1 __Qualifiers for mark fields__

__Permitted context:__ mark

__Syntax:__

```
    mark-qualifier -> OPT | MAND
```

__Semantics:__

'OPT' indicates that the mark field is optional, and 'MAND' that it is mandatory.

3.16 __Syntax summary__

The production rules for the language are reproduced here in abbreviated form for reference.

A-	stands for	add-
C-	stands for	amend-
D-	stands for	delete-
R-	stands for	rename-
-C	stands for	-comm
-SC	stands for	-subcomm
-D	stands for	-def
-Q	stands for	-qualifiers

```
    program -> comm ; ...

    comm -> class-C | link-C | module-C


    class-C -> A-class-C | C-class-C |
               D-class-C | R-class-C

(C) A-class-C -> ADD CLASS classid [ ( A-class-SC ; ... ) ]

      A-class-SC -> descrdata-D | keyindex-D |
                    primgroup-D | variant-D

(C) C-class-C -> AMEND CLASS classid ( C-class-SC ; ... )

      C-class-SC -> descrdata-C | keyindex-C |
                    primgroup-C | variant-C | linkfield-C

(R) D-class-C -> DELETE CLASS classid

(R) R-class-C -> RENAME CLASS classid1 AS classid2


    link-C -> A-link-C | D-link-C

(R) A-link-C -> ADD LINK ( linkfield-D1 , linkfield-D2 )
                    BETWEEN classorvar1 AND classorvar2

      classorvar -> classid | variantid OF classorvar

      linkfield-D -> linkid [ ( A-linkfield-SC ; ... ) ]

        A-linkfield-SC -> descrdata-D | keyindex-D |
                          primgroup-D | mark-D | linkfield-Q

(R) D-link-C -> DELETE LINK ( linkid1 , linkid2 )
                    BETWEEN classorvar1 AND classorvar2


    module-C -> A-mod-C | C-mod-C | D-mod-C | R-mod-C

(C) A-mod-C -> ADD MODULE modname [ ( A-mod-SC ; ... ) ]

      A-mod-SC -> funcname , ...

(C) C-mod-C -> AMEND MODULE modname ( C-mod-SC ; ... )

      C-mod-SC -> A-func-C | D-func-C | R-func-C

    (R) A-func-C -> ADD funcname

    (R) D-func-C -> DELETE funcname

    (R) R-func-C -> RENAME funcname1 AS funcname2
```

```
(R) D-mod-C -> DELETE MODULE modname

(R) R-mod-C -> RENAME MODULE modname1 AS modname2

    descrdata-D -> descr-D | miscdata-D

    descrdata-C -> descr-C | miscdata-C

    descr-D -> DESCRIPTION IS string

    descr-C -> A-descr-C | D-descr-C

      A-descr-C -> ADD descr-D

      D-descr-C -> DELETE DESCRIPTION

    miscdata-D -> MISCDATA IS < value | ( value , ... ) >

    miscdata-C -> A-miscdata-C | D-miscdata-C

      A-miscdata-C -> ADD miscdata-D

      D-miscdata-C -> DELETE MISCDATA

    keyindex-D -> key-D | index-D

    keyindex-C -> key-C | index-C

    key-D -> KEY IS key-spec | NOKEY

      key-spec -> simple-key-spec |
                  ( simple-key-spec , ... )

        simple-key-spec -> primid [ ( funcname ) ]

    key-C -> A-key-C | C-key-C | D-key-C

    A-key-C -> ADD KEY IS key-spec

    C-key-C -> AMEND KEY < ( C-keyindex-SC ; ... ) |
                             C-keyindex-SC >

    D-key-C -> DELETE KEY | NOKEY

(R) index-D -> INDEX index-spec

      index-spec -> indexid ON key-spec

    index-C -> A-index-C | C-index-C |
               D-index-C | R-index-C
```

```
(R) A-index-C -> ADD INDEX index-spec

(C) C-index-C -> AMEND INDEX indexid ( C-keyindex-SC ; ... )

(R) D-index-C -> DELETE INDEX indexid

(R) R-index-C -> RENAME INDEX indexid1 AS indexid2


    C-keyindex-SC -> A-keyindexfield-C |
                     D-keyindexfield-C |
                     A-compfunc-C | D-compfunc-C

    A-keyindexfield-C -> ADD key-spec

    D-keyindexfield-C -> DELETE
                            < primid | ( primid , ... ) >

(R) A-compfunc-C -> COMPFUNC IS compfunc-spec

      compfunc-spec -> funcname FOR
                          < primid | ( primid , ... ) >

(R) D-compfunc-C -> DELETE COMPFUNC FOR primid


    primgroup-D -> prim-D | group-D

    primgroup-C -> prim-C | group-C

    prim-D -> user-type < [ ( basic-type ) ] primid
                          [ ( A-prim-SC ; ... ) ] > , ...

      A-prim-SC -> descrdata-D | primgroup-Q

    prim-C -> A-prim-C | C-prim-C |
              D-prim-C | R-prim-C

(R) A-prim-C -> ADD PRIM prim-D

(C) C-prim-C -> AMEND PRIM primid ( C-prim-SC ; ... )

      C-prim-SC -> descrdata-C | primgroup-Q | C-type-C

        C-type-C -> TYPE IS < [ user-type ] ( basic-type ) |
                              user-type >

(R) D-prim-C -> DELETE PRIM primid

(R) R-prim-C -> RENAME PRIM primid1 AS primid2
```

```
(C) group-D -> GROUP groupid [ ( A-group-SC ; ... ) ]

      A-group-SC -> descrdata-D | keyindex-D |
                    primgroup-D | primgroup-Q

    group-C -> A-group-C | C-group-C |
               D-group-C | R-group-C

(C) A-group-C -> ADD GROUP groupid [ ( A-group-SC ; ... ) ]

(C) C-group-C -> AMEND GROUP groupid ( C-group-SC ; ... )

      C-group-SC -> descrdata-C | keyindex-C |
                    primgroup-C | primgroup-Q

(R) D-group-C -> DELETE GROUP groupid

(R) R-group-C -> RENAME GROUP groupid1 AS groupid2


    primgroup-Q -> primgroup-qualifier , ...

      primgroup-qualifier -> OPT | MAND | RARE | COMMON


(C) variant-D -> VARIANT variantid
                   [ ( A-variant-SC ; ... ) ]

      A-variant-SC -> descrdata-D | primgroup-D | variant-D

    variant-C -> A-variant-C | C-variant-C |
                 D-variant-C | R-variant-C

(C) A-variant-C -> ADD VARIANT variantid
                     [ ( A-variant-SC ; ... ) ]

(C) C-variant-C -> AMEND VARIANT variantid
                     ( C-variant-SC ; ... )

      C-variant-SC -> descrdata-C | primgroup-C |
                      variant-C | linkfield-C

(R) D-variant-C -> DELETE VARIANT variantid

(R) R-variant-C -> RENAME VARIANT variantid1 AS variantid2


     linkfield-C -> C-linkfield-C | R-linkfield-C
```

```
(C) C-linkfield-C -> AMEND LINK linkid
                       ( C-linkfield-SC ; ... )

      C-linkfield-SC -> descrdata-C | keyindex-C |
                        primgroup-C | mark-C | linkfield-Q

(R) R-linkfield-C -> RENAME LINK linkid1 AS linkid2


    linkfield-Q -> linkfield-qualifier , ...

      linkfield-qualifier -> OPT | MAND |
                             RARE | COMMON |
                             SINGLE | MULTIPLE


(C) mark-D -> MARK markid [ ( A-mark-SC ; ... ) ]

      A-mark-SC -> descrdata-D | mark-qualifier

    mark-C -> A-mark-C | C-mark-C | D-mark-C | R-mark-C

(C) A-mark-C -> ADD MARK markid [ ( A-mark-SC ; ... ) ]

(C) C-mark-C -> AMEND MARK markid ( C-mark-SC ; ... )

      C-mark-SC -> descrdata-C | mark-qualifier

(R) D-mark-C -> DELETE MARK markid

(R) R-mark-C -> RENAME MARK markid1 AS markid2


    mark-qualfier -> OPT | MAND
```

3.17 Example

```
//This section consists of a program in the data base
// definition language which makes some amendments to the
// structure of the data base defined in section 2.13.

//First we add a new class

    ADD CLASS REGION
    BEGIN
      STRING NAME, (BYTE-6) CODE  //'CODE' is also a STRING
      KEY IS CODE
    END

//and a new link between REGION and BUILDING

    ADD LINK ( REGION (MAND, SINGLE), BUILDINGS )
      FROM BUILDING TO REGION

//We define some more comparison functions

    AMEND MODULE CMPFS ADD CF1, CF2, CF3, CF4

//and now make some changes to the class employee

    AMEND CLASS EMPLOYEE
    BEGIN

      ADD DESCRIPTION IS "One entry per employee"
        //this replaces the current description

      DELETE INDICES UPSALARY, DOWNSALARY

      ADD PRIM INT (BYTE-1) YEARS_OF_SERVICE,
               STRING PREV_EMPLOYER

      AMEND GROUP SALARY_HISTORY
      BEGIN
        COMMON
        ADD KEY IS DATE
        RENAME PRIM SALARY AS PREV_SALARY
      END

      AMEND GROUP CHILDREN
        AMEND GROUP VACCINATION_HISTORY
          ADD KEY IS VACCIN(CF1)
    //make changes to 3 link fields
      AMEND LINK AUDITPROJ ( ADD PRIM DATE AUDIT_DATE ),
                    //a new link element parameter
                 MANAGER (MAND),
                 CURRPROJS ( RENAME PRIM HOURS AS
                               HRS_PER_WEEK )

    END

//some changes to the key for the class PERSON

    AMEND CLASS PERSON
      AMEND KEY
      BEGIN
        DELETE ( CHRISTIAN_NAMES, DISCRIMINATOR )
        ADD DISCRIMINATOR (CF2)
          //the key fields are now (SURNAME, DISCRIMINATOR)
        COMPFUNC IS CF3 FOR SURNAME
      END

//changes to class PROJECT

    AMEND CLASS PROJECT
    BEGIN

      AMEND PRIM PROJNUM TYPE IS (BYTE-6)
        //to alter the basic type only

      AMEND LINK MEMBERS
      BEGIN
        AMEND MARK TEABOY (MAND)
        RENAME MARK LEADER AS CHIEF
        AMEND INDEX BYSALARY
          COMPFUNC IS REVINT FOR SALARY
      END

    END
```

```
//now some changes to DEPT_MEMBER and its variants

    AMEND CLASS DEPT_MEMBER
    BEGIN

      ADD PRIM INT (BYTE-1) YEARS_OF_SERVICE

      AMEND VARIANT PROFESSOR
      BEGIN
        ADD PRIM STRING RESEARCH_TOPIC
        ADD GROUP PAPERS
        BEGIN
          STRING TITLE, PERIODICAL (MAND)
          DATE DATE_OF_PUBLICATION
          KEY IS TITLE
        END

        RENAME VARIANTS PART_TIME_PROF AS PT_PROF,
                        FULL_TIME_PROF AS FT_PROF

        AMEND VARIANT FT_PROF
          RENAME LINK DOCTORAL_STUDS AS DR_STUDENTS

      END

      AMEND VARIANT STUDENT
        AMEND LINK COURSES_TAKEN
          ADD PRIM CHAR-1 GRADE

    END
```