

8907-87

4851903 0066803 7

Q-70-41

INTERNATIONAL STANDARD

**ISO
8907**

First edition
1987-06-15



INTERNATIONAL ORGANIZATION FOR STANDARDIZATION
ORGANISATION INTERNATIONALE DE NORMALISATION
МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ

Information processing systems — Database language NDL



Documents
Reference number
ISO 8907:1987 (E)
Document Sharing Hub

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work.

Draft International Standards adopted by the technical committees are circulated to the member bodies for approval before their acceptance as International Standards by the ISO Council. They are approved in accordance with ISO procedures requiring at least 75% approval by the member bodies voting.

International Standard ISO 8907 was prepared by Technical Committee ISO/TC 97, *Information processing systems*.

Users should note that all International Standards undergo revision from time to time and that any reference made herein to any other International Standard implies its latest edition, unless otherwise stated.

Contents

1. Scope and field of application	1
2. References	3
3. Overview	5
3.1 Organization	5
3.2 Notation	5
3.3 Conventions	6
3.4 Conformance	7
4. Concepts	9
4.1 Data type	9
4.1.1 Character strings	9
4.1.2 Numbers	9
4.2 Components	10
4.3 Records	10
4.3.1 Database keys	10
4.3.2 System record	11
4.4 Set types	11
4.4.1 Singular sets	11
4.4.2 Recursive sets	11
4.5 Schema	11
4.6 Subschemas	12
4.7 Database	12
4.8 Modules	12
4.9 Procedures	12
4.10 Standard programming languages	13
4.11 Temporary sets	13
4.12 Parameters	13
4.12.1 STATUS parameter	13
4.12.2 TEST parameter	13
4.12.3 RECORD parameter	13
4.13 Session	14
4.14 Transactions	14
4.15 Integrity constraints	14
4.15.1 Check conditions	15
4.15.2 Default values	15
4.15.3 Uniqueness constraints	15
4.15.4 Set ordering criteria	15
4.15.5 Set membership	15
5. Common elements	17
5.1 <condition>	17
5.2 <operand>	19
5.3 <identifier>	20
5.4 <literal>	22
5.5 <data type>	24
5.6 <occurs clause>	26
5.7 <subscripts>	27

5.8 Comments, spaces, and key words	28
6. Schema definition language	31
6.1 <schema>	31
6.2 <schema name clause>	32
6.3 <record type>	33
6.4 <record name clause>	34
6.5 <record uniqueness clause>	35
6.6 <component type>	36
6.7 <component name clause>	37
6.8 <default clause>	38
6.9 <record check clause>	39
6.10 <set type>	40
6.11 <set name clause>	41
6.12 <owner clause>	42
6.13 <order clause>	43
6.14 <member clause>	45
6.15 <member record name clause>	46
6.16 <insertion clause>	47
6.17 <retention clause>	49
6.18 <member uniqueness clause>	50
6.19 <key clause>	51
6.20 <member check clause>	54
6.21 <component identifier>	56
7. Subschema definition language	59
7.1 <subschema>	59
7.2 <subschema name clause>	60
7.3 <record view>	61
7.4 <component view>	62
7.5 <set view>	63
8. Module language	65
8.1 <module>	65
8.2 <module name clause>	67
8.3 <temporary set specifications>	68
8.4 <procedure>	69
9. Data manipulation language	75
9.1 <commit statement>	75
9.2 <connect statement>	76
9.3 <disconnect statement>	77
9.4 <erase statement>	78
9.5 <find statement>	80
9.6 <get statement>	85
9.7 <modify statement>	86
9.8 <nullify cursor statement>	88
9.9 <ready statement>	89
9.10 <reconnect statement>	91
9.11 <rollback statement>	92
9.12 <store statement>	93
9.13 <test database key equal statement>	96
9.14 <test database key null statement>	97
9.15 <test set empty statement>	98
9.16 <test set membership statement>	99
9.17 <database key identifier>	100

9.18 <component view identifier>	101
9.19 <parameter identifier>	103
9.20 <to parameter move> and <to database move>	104
10. Auxiliary operations	107
10.1 <insert operation>	107
10.2 <remove operation>	109
11. Interpretive state	111
11.1 <session state>	111
11.2 <cursors>	112
11.3 <temporary sets>	114
11.4 <ready list>	115
12. Status codes	117
13. Levels	119
Annexes	121
A. Example suppliers and parts problem	121
A.1 Example suppliers and parts <schema>	121
A.2 Example suppliers and parts <subschema>	122
A.3 Example suppliers and parts program	122
A.4 Example suppliers and parts <module>	123
B. Example recursive set problem	125
B.1 Example recursive set <schema>	125
B.2 Example recursive set <subschema>	125
B.3 Example recursive set program	125
B.4 Example recursive set <module>	127
C. Example bill of materials problem	129
C.1 Example bill of materials <schema>	129
C.2 Example bill of materials <subschema>	130
C.3 Example bill of materials program	130
C.4 Example bill of materials <module>	131
Index	133

Information processing systems — Database language NDL

1. Scope and field of application

This standard specifies the syntax and semantics of three database languages:

- 1) A schema definition language, for declaring the structures and integrity constraints of an NDL database.
- 2) A subschema definition language, for declaring a user view of that database.
- 3) A module language and data manipulation language, for declaring the database procedures and executable statements of a specific database application program.

This standard defines the logical data structures and basic operations for an NDL database. It provides functional capabilities for designing, accessing, maintaining, controlling, and protecting the database.

This standard provides a vehicle for portability of database definitions and application programs between conforming implementations.

This standard specifies two levels. Level 2 is the complete NDL database language. Level 1 is the subset of Level 2 defined in clause 13, "Levels" on page 119.

This standard does not specify some of the facilities that might be provided in operational database environments, such as the following:

- 1) An access control facility for granting access and operational privileges to specific users
- 2) Additional integrity control capabilities for specifying more complex integrity constraints on the database
- 3) A facility to import and export schema definitions
- 4) A database unload facility for copying record and set populations to standard files for information interchange
- 5) A schema database for making schema and subschema information available to accessing application programs
- 6) A schema manipulation language for creating, modifying, or deleting portions of the schema or subschemas
- 7) Interfaces to a data dictionary
- 8) Application program pre-processing facilities for producing separate standard database modules and standard language programs

- 9) A data storage definition language for defining physical storage structures and physical access methods
- 10) Database procedures for user specified assertions and triggers
- 11) A natural language query facility for ad hoc access to the database
- 12) Report generator facilities for producing output tables and charts
- 13) Graphics capabilities for direct database interface with standard graphics systems
- 14) A distributed database facility for defining and accessing data at different nodes in a communications network

Standards for such additional facilities could be specified in a manner that is upward compatible with this standard. Some of the additional capabilities might be specified by subsequent versions of this standard, others could be specified by separate standards, and some may always be implementor-defined.

This standard applies to implementations that exist in an environment that may include application programming languages, end-user query languages, report generator systems, data dictionary systems, program library systems, and distributed communication systems, as well as various tools for database design, data administration, and performance optimization.

2. References

ISO 1539, *Programming Languages - FORTRAN*, (endorsement by ISO of American National Standard X3.9).

ISO 1989, *Programming Languages - COBOL*, (endorsement by ISO of American National Standard X3.23).

ISO 6160, *Programming Languages - PL/I*, (endorsement by ISO of American National Standard X3.53).

ISO 7185, *Programming Languages - Pascal*, (endorsement by ISO of British Standard 6192 for the English text).

3. Overview

3.1 Organization

The organization of this standard is as follows:

- 1) 3.2, "Notation" on page 5 and 3.3, "Conventions" on page 6 define the notations and conventions used in this standard.
- 2) 3.4, "Conformance" on page 7 defines conformance criteria.
- 3) Clause 4, "Concepts" on page 9 defines terms and presents concepts used in the definition of NDL.
- 4) Clause 5, "Common elements" on page 17 defines language elements that occur in several parts of NDL language.
- 5) Clause 6, "Schema definition language" on page 31 defines the NDL facilities for specifying a database.
- 6) Clause 7, "Subschema definition language" on page 59 defines the NDL facilities for specifying portions of a database that can be referenced by an application program.
- 7) Clause 8, "Module language" on page 65 defines NDL modules and procedures.
- 8) Clause 9, "Data manipulation language" on page 75 defines the data manipulation statements of NDL.
- 9) Clause 10, "Auxiliary operations" on page 107 defines certain operations that are used in the definition of NDL data manipulation statements.
- 10) Clause 11, "Interpretive state" on page 111 defines the "session state", which is used in the definition of NDL data manipulation statements.
- 11) Clause 12, "Status codes" on page 117 defines the values of the STATUS parameter after the execution of NDL data manipulation statements.
- 12) Clause 13, "Levels" on page 119 defines the two levels of NDL.

3.2 Notation

The syntactic notation used in this standard is BNF ("Backus Normal Form" or "Backus-Naur Form"), with the following extensions:

- 1) Square brackets ([]) indicate optional elements.
- 2) Ellipses (...) indicate elements that may be repeated one or more times.
- 3) Braces ({} group sequences of elements.

In the BNF syntax, a production symbol $\langle A \rangle$ is defined to "contain" a production symbol $\langle B \rangle$ if $\langle B \rangle$ occurs someplace in the expansion of $\langle A \rangle$. If $\langle A \rangle$ contains $\langle B \rangle$, then $\langle B \rangle$ is "contained in" $\langle A \rangle$. If $\langle A \rangle$ contains $\langle B \rangle$, then $\langle A \rangle$ is the "containing" $\langle A \rangle$ production symbol for $\langle B \rangle$.

A production symbol $\langle A \rangle$ "immediately contains" a production symbol $\langle B \rangle$ if $\langle A \rangle$ contains $\langle B \rangle$ and if $\langle B \rangle$ occurs in the first expansion of $\langle A \rangle$.

The lexical units of this standard (e.g. $\langle \text{separator} \rangle$, $\langle \text{key word} \rangle$, $\langle \text{identifier} \rangle$, and $\langle \text{literal} \rangle$) are defined elsewhere in this standard. The following characters have special significance in some production rules:

'	In $\langle \text{escape identifier} \rangle$
"	In $\langle \text{character string literal} \rangle$
(	In $\langle \text{comment} \rangle$, $\langle \text{condition} \rangle$, and $\langle \text{subscripts} \rangle$
)	In $\langle \text{comment} \rangle$, $\langle \text{condition} \rangle$, and $\langle \text{subscripts} \rangle$
*	In $\langle \text{comment} \rangle$
	In $\langle \text{numeric literal} \rangle$
-	In $\langle \text{numeric literal} \rangle$
<	In $\langle \text{relation} \rangle$
>	In $\langle \text{relation} \rangle$
=	In $\langle \text{component identifier match} \rangle$, $\langle \text{relation} \rangle$, and $\langle \text{test database key equal statement} \rangle$
E	In $\langle \text{approximate numeric literal} \rangle$
_	In $\langle \text{identifier} \rangle$
.	In $\langle \text{component identifier} \rangle$, $\langle \text{component view identifier} \rangle$, $\langle \text{numeric literal} \rangle$, $\langle \text{subschema name clause} \rangle$, and $\langle \text{subschema specification} \rangle$

3.3 Conventions

Syntactic elements of this standard are specified in terms of:

- 1) *Function*: A short statement of the purpose of the element.
- 2) *Format*: A BNF definition of the syntax of the element.
- 3) *Syntax Rules*: Additional syntactic constraints not expressed in BNF that the element shall satisfy.
- 4) *General Rules*: A sequential specification of the run-time effect of the element.

In the Syntax Rules, the term "shall" defines conditions that are required to be true of syntactically conforming NDL language. The treatment of NDL language that does not conform to the Formats or the Syntax Rules is implementor-defined.

A conforming implementation is not required to perform the exact sequence of actions defined in the General Rules, but shall achieve the same effect on the database as that sequence.

The term "persistent object" is used to characterize objects such as $\langle \text{module} \rangle$ s and $\langle \text{schema} \rangle$ s that are created and destroyed by implementor-defined mechanisms.

In this standard, clauses begin a new odd-numbered page, and in clause 5, "Common elements" on page 17 through clause 11, "Interpretive state" on page 111 subclauses begin a new page. The resulting blank space is not significant.

3.4 Conformance

This standard specifies conforming NDL language and conforming NDL implementations. Conforming NDL language shall abide by the BNF Format and associated Syntax Rules. A conforming NDL implementation shall process standard conforming NDL language according to the General Rules.

A conforming implementation may provide additional facilities or options not specified by this standard. An implementation remains conforming even if it provides user options to process nonconforming NDL language or to process conforming NDL language in a nonconforming manner.

This standard does not define the method or the time of binding between application programs and database management system components.

An implementation that conforms to Level 1 of this standard shall meet the requirements for Level 2 conformance except that it may omit certain facilities as specified in clause 13, "Levels" on page 119.

This standard specifies three languages and defines the syntax and semantics of each of them. Implementors of database management systems may claim conformance for their products to this standard in two ways. An implementation may support the identical syntax of the languages and claim both syntactic and functional conformance, or it may use different syntax to express the same concepts, semantics, and functionality, and claim functional conformance only.

Implementations claiming functional conformance to any of the three standard languages shall specify equivalence with the semantics and functionality of those standard languages. For example, it should be possible to show that a schema displayed by a database management system display utility could be recoded using the standard schema definition language syntax and have equivalent functionality.

4. Concepts

4.1 Data type

A <data type> is a set of representable values. The logical representation of a value is a <literal>. The physical representation of a value is implementor-defined.

A value is primitive, in that it has no logical subdivision within this standard. Values are the basis of definition for data items and arrays.

A value is either a character string or a number. A character string and a number are not comparable values.

4.1.1 Character strings

A character string consists of a sequence of characters of the implementor-defined character set. A character string has a length, which is a positive integer that specifies the number of characters in the sequence.

4.1.2 Numbers

A number is either an exact numeric value or an approximate numeric value. All numbers are comparable values.

An exact numeric value has a precision and a scale. The precision is a positive integer that determines the number of significant decimal digits. The scale is a signed integer. A scale of 0 indicates that the number is an integer. For a scale of N, the exact numeric value is the integer value of the significant digits multiplied by 10 to the power -N.

An approximate numeric value consists of a mantissa and an exponent. The mantissa is a signed numeric value, and the exponent is a signed integer that specifies the magnitude of the mantissa. An approximate numeric value has a precision. The precision is a positive integer that specifies the number of significant binary digits in the mantissa.

Whenever an exact numeric value is assigned to a data item or parameter representing an exact numeric value, its value shall be exactly representable in the data type of the target. The value is converted to have the precision and scale of the target.

Whenever an exact or approximate numeric value is assigned to a data item or parameter representing an approximate numeric value, an approximation of its value is represented in the data type of the target. The value is converted to have the precision of the target.

4.2 Components

A component is either a data item or an array of data items. A **<component type>** defines a collection of component occurrences all having the same **<data type>**.

A data item consists of a single value. An array consists of a sequence of data items.

An array is specified by **<extents>**, which are a list of **<unsigned integers>**. The number of data items that occur in an array is equal to the product of the **<unsigned integer>**s.

A data item not contained in an array is referenced by a **<component name>** and a **<database key>**. A data item contained in an array is referenced by a **<component name>**, a **<database key>**, and **<subscripts>**.

All values of the same **<component type>** have the same **<data type>**. If the **<component type>** defines an array, then each component occurrence is an array with the same dimension and extent integers. All character strings of the same character string component have the same **<length>**. All numbers of the same exact numeric component have the same **<precision>** and **<scale>**. All numbers of the same approximate numeric component have the same **<precision>**.

Every **<component type>** is defined as part of a **<record type>**. The **<component type>** specifies the **<data type>**. If a **<component type>** defines an array, then it specifies the **<extents>**. A character string **<component type>** specifies the **<length>** of its character strings. An exact numeric **<component type>** specifies the **<precision>** and **<scale>** of its numbers. An approximate numeric **<component type>** specifies the **<precision>** of its numbers.

4.3 Records

The record is the basic unit of manipulation in this standard. Records are stored, erased, found, modified, and connected, disconnected, and reconnected in sets.

A record is a collection of components. A **<record type>** defines a collection of record occurrences all having the same **<component type>**s. A **<record type>** defines its **<component type>**s, and specifies integrity constraints that its record occurrences shall satisfy.

Each record is an occurrence of exactly one **<record type>**, and consists of exactly the components defined by that **<record type>**.

The records of each **<record type>** are maintained in an implementor-defined order.

4.3.1 Database keys

All records are distinguishable. Each record is identified uniquely by a **<database key>**. A **<database key>** is either *null*, in which case it identifies no record, or it is nonnull and identifies exactly one record in the database. A **<database key>** has no logical representation, and is not available through any interface defined by this standard. The physical representation of a **<database key>** is implementor-defined.

The **<database key>**s are used to model cursors in the session state and to define the semantics of statements in the data manipulation language. In the session state, a **<database key>** is referenced by **<database key>**. In data manipulation language statements, a **<database key>** is referenced by a **<database key identifier>**.

4.3.2 System record

This standard assumes the existence of a special SYSTEM <record type>. The SYSTEM <record type> has no <component type>s, no integrity constraints, and exactly one imaginary record occurrence, called the system record.

This standard does not specify a <database key> for the system record. The system record and SYSTEM <record type> exist only for the definition of singular sets.

4.4 Set types

A <set type> is a defined relationship between two or more <record type>s.

In a <set type>, one <record type> is designated as the *owner* <record type>, and each of the other <record type>s is referred to as a *member* <record type> of the <set type>.

For each <set type>, there may be zero, one, or more *sets* in a database.

A set consists of an *owner record*, whose <record type> is the owner <record type>, and zero, one, or more *member records*, each of whose <record type> is a member <record type>.

A set is identified uniquely in a database by its <set name> and its owner record occurrence. When an occurrence of the owner <record type> of a <set type> is stored in the database, a new set of that <set type> is created.

When an occurrence of <record type> is stored in the database, it belongs to at most one set of each <set type> in which the <record type> is a member <record type>.

The member records of each set of a <set type> are maintained in a sequential order determined by the ordering criteria for that <set type>.

4.4.1 Singular sets

A singular <set type> is a <set type> having SYSTEM as its owner <record type>. A singular set is the unique occurrence of a singular <set type>.

4.4.2 Recursive sets

A recursive <set type> is a set type having the same <record type> as both the owner <record type> and as a member <record type>. A recursive set is an occurrence of a recursive <set type>.

4.5 Schema

A <schema> is a persistent object specified by the schema definition language. The <schema> is a logical description of the database. A <schema> consists of a <schema name>, a collection of <record type> descriptions, and a collection of <set type> descriptions.

4.6 Subschemas

A <subschema> is a persistent object specified by the subschema definition language. The <subschema> is a logical description of that portion of a database available to an accessing <module>. A <subschema> consists of a <subschema name>, a collection of <record view>s, and a collection of <set view>s.

A <record view> specifies the available <component type>s of a given <record type>. A <set view> specifies an available <set type>. The <record view>s and <set view>s taken together determine the available member <record type>s of each <set type>.

The <record view>s and <set view>s may define alias <component view name>s, <record view name>s and <set view name>s for <component type>s, <record type>s, and <set type>s. The alias names become the only names available for those <component type>s, <record type>s, and <set type>s to an accessing <module>.

The records available to an accessing <module> are maintained in an implementor-defined order. This order is not necessarily the same as the implementor-defined order of the records of each <record type>.

4.7 Database

A database is the collection of all data defined by a <schema>. It consists of the occurrences of each <record type> and each <set type>. There is exactly one database for each <schema> definition.

4.8 Modules

A <module> is a persistent object specified in the module language. A <module> consists of an optional <module name>, a <language clause>, a <subschema specification>, an optional <temporary set specifications>, and one or more <procedure>s.

An application program is a segment of executable code, possibly consisting of multiple sub-programs. A single <module> is associated with an application program during its execution. An application program shall be associated with at most one <module>. The manner in which this association is specified, including the possible requirement for execution of some implementor-defined statement, is implementor-defined.

4.9 Procedures

A <procedure> consists of a <procedure name>, an optional sequence of <parameter declaration>s, and an optional sequence of <NDL statement>s.

An application program associated with a <module> may reference the <procedure>s of that <module> by a "call" statement that specifies the <procedure name> of the <procedure> and supplies a sequence of parameter values corresponding in number and in <data type> to the <parameter declaration>s of the procedure. A call of a <procedure> causes the sequence of <NDL statement>s that it contains to be executed.

The <subschema specification> of a <module> specifies the <subschema> that defines the records and sets that can be referenced by the <module>.

The <temporary set specifications> of a <module> defines temporary <set type>s that may be referenced by the <module>.

4.10 Standard programming languages

This standard specifies the actions of <procedure>s in <module>s when those <procedure>s are called by programs that conform to specified standard programming languages. The terms "standard COBOL program", "standard FORTRAN program", "standard Pascal program", and "standard PL/I program" refer to programs that meet the conformance criteria of the standards listed in clause 2, "References" on page 3.

4.11 Temporary sets

A <temporary set> is an occurrence of a temporary <set type>. A temporary <set type> is declared by a <temporary set specification> in a <module>.

For each <temporary set specification> in a <module>, an occurrence of the corresponding temporary <set type> is created when the <module> is initiated and destroyed when the <module> is terminated.

4.12 Parameters

A parameter is declared in a <procedure> by a <parameter declaration>. The <parameter declaration> specifies whether the parameter is an elementary data item or an array of data items, and specifies the <data type> of its values. A parameter either assumes or supplies the value or values of the corresponding argument in the call of that procedure.

4.12.1 STATUS parameter

The STATUS parameter is a special parameter of <length> 5 characters. If such a parameter is declared in a <procedure>, then its value is set to a status code that either indicates that a call of the <procedure> was successfully completed or that identifies an exception condition that occurred during execution of the <procedure>.

4.12.2 TEST parameter

The TEST parameter is a special parameter of <length> 1 character. If a test statement is executed in a <procedure> containing this parameter, then the value of the parameter is set to "1" if the test is true, and to "0" if the test is false.

4.12.3 RECORD parameter

The RECORD parameter is a special parameter of <length> 18 characters. If such a parameter is declared in a <procedure>, then it is set to the <record name> of the record identified by the <session cursor> of the <session state>.

4.13 Session

A database operation is the execution of an **<NDL statement>**.

A session is the sequence of database operations performed during the execution of an application program associated with a **<module>**.

A **<session state>** is an ephemeral object associated with a session. A **<session state>** is created prior to the first database operation in a session and is destroyed after the last database operation in the session.

A **<session state>** consists of **<cursors>**, **<temporary sets>**, and a **<ready list>**. The **<cursors>** include: A **<session cursor>** to identify the current session record, a **<record cursor>** to identify the current record of each **<subschema>** **<record type>**, and a **<set cursor>** to identify the owner record and the current member record in the current set of each **<subschema>** **<set type>**. The **<temporary sets>** maintain the contents of each **<temporary set>** defined by the associated **<module>**. The **<ready list>** maintains the **<ready specification>**s for each **<record type>** that has been activated by a **<ready statement>**.

The **<session state>** is available to an application program only through **<NDL statement>**s. The contents of the **<session state>** are modified by the database management system upon execution of **<NDL statement>**s in a **<procedure>** of its associated **<module>**. The data manipulation language includes **<NDL statement>**s to find and compare **<database key>**s and to modify the contents of the **<ready list>** and each **<temporary set>**.

4.14 Transactions

A transaction is a sequence of operations, including database operations, that is atomic with respect to recovery and concurrency. Transactions terminate with a **<commit statement>** or a **<rollback statement>**. If a transaction terminates with a **<commit statement>**, then all changes made to the database by that transaction are made accessible to all concurrent transactions. If a transaction terminates with a **<rollback statement>**, then all changes made to the database by that transaction are canceled. Committed changes cannot be canceled. Changes made to the database by a transaction can be perceived by that transaction, but cannot be perceived by other transactions until that transaction terminates with a **<commit statement>**.

The execution of concurrent transactions is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing transactions that produces the same effect as some serial execution of those same transactions. A serial execution is one in which each transaction executes to completion before the next transaction begins.

The execution of an **<NDL statement>** within a transaction has no effect on the database or **<session state>** other than the effect stated in the General Rules for that **<NDL statement>**. Together with serializable execution, this implies that all read operations are reproducible within a transaction, except for changes explicitly made by the transaction itself.

4.15 Integrity constraints

Integrity constraints restrict the valid states of the database. Integrity constraints may be defined for **<record type>**s or **<set type>**s, and for component constraints between owner and member **<record type>**s.

Integrity constraints are checked after execution of each **<NDL statement>**. If the objects associated with the integrity constraint do not satisfy the constraint, then the **<NDL statement>** has no effect on the database, and the STATUS parameter is set to indicate the specified exception.

4.15.1 Check conditions

A check condition is an expression that shall be satisfied by the values of a record when its status in the database is altered. The expression is specified by a <condition> of a <record check clause> of a <record type> or a <member check clause> of a <set type>.

4.15.2 Default values

A default value is a value to be assumed by all component occurrences in the absence of an explicit value supplied by a <procedure>. A default value is specified by a <literal> in a <default clause> of a <component type>.

4.15.3 Uniqueness constraints

A uniqueness constraint is a specification that no two records of a given <record type>, or no two member records of a set of a given <set type>, may occur in the database with identical values for specified components. Uniqueness constraints are specified by <record uniqueness clause>s in a <record type>, or by <member uniqueness clause>s in a <member clause> of a <set type>.

4.15.4 Set ordering criteria

A set ordering specifies the sequential ordering of member records in a set. The ordering is one of the following:

- 1) Sorted by component values
- 2) Chronological or reversed chronological
- 3) Relative positioning as next or prior to a given member record
- 4) Implementor-defined

Set ordering criteria are specified by the <order clause> of a <set type>, and if the order is sorted, by the <key clause> of each <member clause> contained in that <set type>.

4.15.5 Set membership

Set membership specifies the insertion and retention modes of a record for each set in which it occurs as a member record. Each member <record type> has an insertion mode and a retention mode.

Insertion is either automatic, structural, or manual. If insertion is manual, then records shall be inserted as member records in a set only by an explicit <connect statement>. If insertion is automatic or structural, then each record of a member <record type> becomes a member of some set of that <set type> when it is initially stored in the database. If insertion is automatic or manual, then the owner record is identified by the application program. If insertion is structural, then the owner record is selected by the database management system to have values of specified data items equal to those of the record to be inserted. The insertion mode is specified by an <insertion clause> in each <member clause> of the <schema>.

Retention is either fixed, mandatory, or optional. If retention is fixed, then a record, having once become a member record of some set, remains a member of that set until it is erased from the database. If retention is mandatory, then a record, having once become a member record of a set, remains a member of some set of the same <set type> until it is erased from the database. If retention is optional, then a record, having once

become a member record of some set, need not remain a member of that set or of any set of that <set type>. The retention mode is specified by a <retention clause> in each <member clause> of the <schema>.

5. Common elements

5.1 <condition>

Function

Specify an expression that shall be evaluated as either *true* or *false*.

Format

<condition> ::=
 <alternative> [{OR <alternative>}...]

<alternative> ::=
 <simple condition> [{AND <simple condition>}...]

<simple condition> ::=
 <subcondition>
 | <negated subcondition>
 | <relation condition>

<subcondition> ::= (<condition>)

<negated subcondition> ::=
 NOT (<condition>)

<relation condition> ::=
 <operand> <relation> <operand>

<relation> ::=
 < | <= | = | >= | > | <>

Syntax Rules

- 1) If the first <operand> in a <relation condition> specifies an array, then the second <operand> in that <relation condition> shall not specify an array.
- 2) If the type of the first operand is character string, then the type of the second operand shall be character string. If the type of the first operand is exact numeric, then the type of the second operand shall be exact numeric or approximate numeric. If the type of the first operand is approximate numeric, then the type of the second operand shall be exact numeric or approximate numeric.

General Rules

- 1) A <condition> is *true* if any <alternative> contained in it is *true*; otherwise, it is *false*.
- 2) An <alternative> is *true* if every <simple condition> contained in it is *true*; otherwise, it is *false*.
- 3) A <simple condition> that immediately contains a <subcondition> is *true* if the <subcondition> is *true*; otherwise, it is *false*.

A <simple condition> that immediately contains a <negated subcondition> is *true* if the <condition> immediately contained in the <negated subcondition> is *false*; otherwise, it is *false*.

A <simple condition> that immediately contains a <relation condition> is *true* if the <relation condition> is *true*; otherwise, it is *false*.

- 4) The first (second) operand of a <relation condition> is the value of the first (second) <operand> in that <relation condition>.
- 5) The <relation> of a <relation condition> specifies a comparison to be made between the two operands of the <relation condition>. The comparisons specified by the alternative <relation>s are as follows:

<relation>	Comparison
<	less than
<=	less than or equal
=	equal
<>	not equal
>	greater than
>=	greater than or equal

- 6) If neither operand of a <relation condition> is an array, then the <relation condition> is *true* if the specified comparison of the two operands is *true*; otherwise, the <relation condition> is *false*. If one of the operands of a <relation condition> is an array, then the <relation condition> is *true* if the specified comparison of every value in the array sequence with the other operand is *true*; otherwise, the <relation condition> is *false*.
- 7) Case:
 - a) If operands are of exact numeric or approximate numeric type, then the comparison is performed according to their numeric values.
 - b) If the operands are of character string type, then:
 - i) If the lengths of the two operands are not equal, then the shorter operand is considered to be extended on the right with space characters.
 - ii) The comparison is performed from left to right, comparing the individual characters of the operands. If all such individual comparisons are equal, then the two operands are equal; otherwise, the result of the comparison of the operands is defined to be the result of the leftmost nonequal comparison of the individual characters.
 - iii) The comparison of individual characters is implementor-defined.
- 8) In General Rules, the terms "less than", "greater than", "equal", and "not equal" refer to the corresponding relational operators.

5.2 <operand>

Function

Specify a value.

Format

<operand> ::=
 <component identifier>
 | <component view identifier>
 | <parameter identifier>
 | <literal>

Syntax Rules

- 1) Case:
 - a) If an <operand> is contained in a <schema>, then it shall not be a <parameter identifier> or a <component view identifier>.
 - b) If an <operand> is contained in a <module>, then it shall not be a <component identifier>.
- 2) The data type of an <operand> is the data type of the <component identifier>, <component view identifier>, <parameter identifier>, or <literal>.

General Rules

- 1) The value of an <operand> is the data item or array referenced by the <component identifier>, <component view identifier>, or <parameter identifier>, or the value of the <literal>.

5.3 <identifier>

Function

Specify names for parameters in a <procedure> or objects in the database.

Format

<identifier> ::=
 <regular identifier> | <escape identifier>

<regular identifier> ::=
 <upper case letter>
 [{{<underscore>} <letter or digit>}...]

<underscore> ::= _

<letter or digit> ::=
 <upper case letter> | <digit>

<letter> ::=
 <upper case letter> | <lower case letter>

<upper case letter> ::=
 A|B|C|D|E|F|G|H|I
 |J|K|L|M|N|O|P|Q|R
 |S|T|U|V|W|X|Y|Z

<lower case letter> ::=
 a|b|c|d|e|f|g|h|i
 |j|k|l|m|n|o|p|q|r
 |s|t|u|v|w|x|y|z

<digit> ::=
 0|1|2|3|4|5|6|7|8|9

<escape identifier> ::=
 '<escape identifier character representation>...'

<escape identifier character representation> ::=
 <escape identifier character>
 | <apostrophe representation>

<escape identifier character> ::=
 See Syntax Rule 3.

<apostrophe representation> ::= "

Syntax Rules

- 1) A <regular identifier> shall contain at most 18 characters. An <escape identifier> shall contain at most 18 <escape identifier character representation>s.
- 2) A <regular identifier> shall not be the same sequence of characters as a <key word> (see 5.8, "Comments, spaces, and key words" on page 28).
- 3) An <escape identifier character> is any character in the implementor-defined character set other than the apostrophe character (').
- 4) Each <apostrophe representation> in an <escape identifier> represents a single apostrophe character.
- 5) If an <escape identifier> (apart from the enclosing apostrophes) conforms to the format of a <regular identifier>, and if it is not a <key word>, then it is equivalent to the <regular identifier> formed by removing the enclosing apostrophes.

General Rules

None.

5.4 <literal>

Function

Specify a literal value.

Format

<literal> ::=

 <character string literal>
 | <numeric literal>

<character string literal> ::=

 "<character representation>..."

<character representation> ::=

 <nonquote character>
 | <quote representation>

<nonquote character> ::=

 See Syntax Rule 1.

<quote representation> ::=

 ""

<numeric literal> ::=

 <exact numeric literal>
 | <approximate numeric literal>

<exact numeric literal> ::=

 [+ | -] { <unsigned integer>[.<unsigned integer>]
 | <unsigned integer>.
 | .<unsigned integer>}

<approximate numeric literal> ::=

 <mantissa>E<exponent>

<mantissa> ::= <exact numeric literal>

<exponent> ::= <signed integer>

<signed integer> ::= [+ | -] <unsigned integer>

<unsigned integer> ::=

 <digit>...

Syntax Rules

- 1) A <nonquote character> is any character in the implementor-defined character set other than the double quote mark character (").
- 2) The data type of a <character string literal> is character string. The length of a <character string literal> is the number of <character representation>s that it contains. Each <quote representation> in a <character string literal> represents a single quotation mark character in both the value and the length of the <character string literal>.
- 3) An <exact numeric literal> without a decimal point (.) has an implied decimal point following the last <digit>.
- 4) The data type of an <exact numeric literal> is exact numeric. The precision of an <exact numeric literal> is the number of <digit>s that it contains. The scale of an <exact numeric literal> is the number of <digit>s to the right of the decimal point.
- 5) The data type of an <approximate numeric literal> is approximate numeric. The precision of an <approximate numeric literal> is the number of binary digits required to represent the significant digits of its mantissa.

General Rules

- 1) The value of a <character string literal> is the sequence of characters that it contains.
- 2) The numeric value of an <unsigned integer>, <signed integer>, or <exact numeric literal> is derived from the normal mathematical interpretation of signed positional decimal notation.
- 3) The numeric value of an <approximate numeric literal> is the product of the exact numeric value represented by the <mantissa> with the number obtained by raising the number 10 to the power represented by the <exponent>.

5.5 <data type>

Function

Specify a data type.

Format

<data type> ::=

 <character string type>
 | <exact numeric type>
 | <approximate numeric type>

<character string type> ::=

 CHARACTER [<length>]

<exact numeric type> ::=

 FIXED <precision> [<scale>]
 | NUMERIC <precision> [<scale>]
 | INTEGER

<approximate numeric type> ::=

 FLOAT <precision>
 | REAL
 | DOUBLE PRECISION

<length> ::= <unsigned integer>

<precision> ::= <unsigned integer>

<scale> ::= <signed integer>

Syntax Rules

- 1) The value of an <unsigned integer> that is a <length> or a <precision> shall be greater than 0.
- 2) If a <length> is omitted, then it is assumed to be 1. If a <scale> is omitted, then it is assumed to be 0.

General Rules

- 1) A <data type> specifies a class of data item values.
- 2) A data item of <data type> CHARACTER has a character string value with length equal to the value of the specified <length>.
- 3) A data item of <data type> FIXED has an exact numeric value with scale equal to the value of the specified <scale>, and with precision greater than or equal to the value of the specified <precision>.
- 4) A data item of <data type> NUMERIC has an exact numeric value with precision and scale equal respectively to the value of the specified <precision> and <scale>. If the precision is P and the scale is S, then the absolute value of any nonzero NUMERIC data item shall be greater than or equal to 1E-S and less than or equal to 1E(P-S)-1E-S.

- 5) A data item of <data type> INTEGER has an exact numeric value with a scale of 0 and with implementor-defined precision.
- 6) A data item of <data type> FLOAT has an approximate numeric value with binary precision equal to or greater than the value of the specified <precision>.
- 7) A data item of <data type> REAL has an approximate numeric value with implementor-defined precision.
- 8) A data item of <data type> DOUBLE PRECISION has an approximate numeric value with implementor-defined precision that is greater than the implementor-defined precision of REAL.

5.6 <occurs clause>

Function

Define an array by specifying its <extents>.

Format

<occurs clause> ::=
OCCURS <extents>

<extents> ::= <unsigned integer>...

Syntax Rules

- 1) The value of each <unsigned integer> in the <extents> shall be greater than 0.

General Rules

- 1) An <occurs clause> specifies an array of data items. The product of the values of the <unsigned integer>s in the <extents> specifies the number of data items contained in each occurrence of the array.

5.7 <subscripts>

Function

Specify a data item of an array.

Format

<subscripts> ::=
(<operand>...)

Syntax Rules

- 1) The *contextual* <occurs clause> of a <subscripts> is specified in the Syntax Rules of the production symbol that immediately contains the <subscripts>.
- 2) The number of <operand>s shall be equal to the number of <unsigned integer>s in the <extents> of the contextual <occurs clause>.
- 3) Case:
 - a) If an <operand> is a <literal>, then it shall be an <unsigned integer> whose value is positive and is not greater than the value of the corresponding <unsigned integer> of the <extents> of the contextual <occurs clause>.
 - b) If an <operand> is not a <literal>, then its data type shall be exact numeric with a scale of 0.

General Rules

- 1) Let d be the number of <operand>s in the <subscripts>.
- 2) For i from 1 to d , let S_i be the value of the i -th <operand> in the <subscripts>.
- 3) For i from 1 to d , let E_i be the value of the i -th <unsigned integer> in the <extents> of the contextual <occurs clause>.
- 4) For i from 1 to d , if S_i is less than 1 or S_i is greater than E_i , then raise exception *array reference: subscript out of bounds*.
- 5) Let M_d be 1. For i from 1 to $(d-1)$, let M_i be the product of the terms E_j , for j from $(i+1)$ to d .
- 6) Let J be the sum of the terms $M_i \cdot (S_i - 1)$, for i from 1 to d .
- 7) The value of the <subscripts> is $J + 1$.

5.8 Comments, spaces, and key words

Function

Specify lexical units.

Format

<separator> ::=
 { <comment> | <space> | <newline> }...

<comment> ::=
 (*[<character>...]*)

<character> ::=
 <digit> | <letter> | <special character>

<special character> ::=
 See Syntax Rule 1.

<space> ::=
 space character

<newline> ::=
 implementor-defined end-of-line indicator

<word> ::=
 <key word> | <identifier> | <literal>

<key word> ::=
 ABSOLUTE | ALL | AND | AS | ASCENDING | AUTOMATIC
 | CASCADE | CHARACTER | CHECK | COBOL
 | COMMIT | CONNECT | CONTAINS
 | DEFAULT | DESCENDING | DISCONNECT | DOUBLE | DUPLICATES
 | EMPTY | ERASE | EXCLUSIVE
 | FIND | FINISH | FIRST | FIXED | FLOAT | FOR | FORTRAN | FROM | FULL
 | GET | IN | INSERTION | INTEGER | ITEM | KEY | LANGUAGE | LAST
 | MANDATORY | MANUAL | MEMBER | MODIFY | MODULE
 | NEXT | NOT | NULL | NULLIFY | NUMERIC
 | OCCURS | OF | OPTIONAL | OR | ORDER | OWNER
 | PARTIAL | PASCAL | PLI | PRECISION
 | PRIOR | PROCEDURE | PROHIBITED | PROTECTED
 | READY | REAL | RECONNECT | RECORD | RELATIVE | RENAMED
 | RETAIN | RETENTION | RETRIEVE | ROLLBACK
 | SCHEMA | SESSION | SET | SHARED | SORTED
 | STATUS | STORE | STRUCTURAL | SUBSCHEMA | SYSTEM
 | TEST | TO | TYPE | UNIQUE | UPDATE | WHERE | WITH

Syntax Rules

- 1) A <special character> is any character in the implementor-defined character set other than a <digit> or a <letter>. If the implementor-defined end-of-line indicator (<newline>) is a character, then it is also excluded from <special character>.
- 2) A terminal production of <schema>, <subschema>, or <module> consists of a sequence of <word>s and symbols optionally separated by <separator>s. A <word> shall not be immediately followed by another <word> without an intervening <separator>.
- 3) The string "<character>..." within a <comment> shall not contain the substring "*").

General Rules

None.

6. Schema definition language

6.1 <schema>

Function

Define the logical structure of a database.

Format

```
<schema> ::=  
    <schema name clause>  
    [ {<record type> | <set type>}...]
```

Syntax Rules

None.

General Rules

None.

6.2 <schema name clause>

Function

Name a <schema>.

Format

<schema name clause> ::=
 SCHEMA <schema name>

<schema name> ::= <identifier>

Syntax Rules

- 1) The <schema name> shall be different from the <schema name> of any other <schema> in the same environment. The concept of environment is implementor-defined.

General Rules

- 1) A <schema name clause> defines the <identifier> to be a <schema name> that designates the containing <schema>.

6.3 <record type>

Function

Define a <record type>.

Format

```
<record type> ::=  
    <record name clause>  
    [ { <record uniqueness clause>  
        | <component type>  
        | <record check clause>}...]
```

Syntax Rules

None.

General Rules

None.

6.4 <record name clause>

Function

Name a <record type>.

Format

<record name clause> ::=
RECORD <record name>

<record name> ::= <identifier>

Syntax Rules

- 1) The <record name> shall be different from the <record name> of any other <record name clause> in the containing <schema>.

General Rules

- 1) A <record name clause> defines the <identifier> to be a <record name> that designates the containing <record type>.

6.5 <record uniqueness clause>

Function

Specify a uniqueness constraint for occurrences of a <record type>.

Format

<record uniqueness clause> ::=
 UNIQUE <component identifier>...

Syntax Rules

- 1) If the <component name> of a <component identifier> designates an array, then that <component identifier> shall contain <subscripts>.
- 2) A given <component identifier> shall be contained at most once in a single <record uniqueness clause>.
- 3) The contextual <record type> of each <component identifier> in a <record uniqueness clause> is the containing <record type>.

General Rules

- 1) When the <component identifier>s of a <record uniqueness clause> are referenced for a record occurrence, the contextual <database key> for those <component identifier>s is the <database key> of that record occurrence.
- 2) A <record uniqueness clause> is violated if the database contains two occurrences of the containing <record type> in which the values of each of the components designated by the <component identifier>s in the <record uniqueness clause> in the first record occurrence are *equal* to the values of the corresponding components in the second record occurrence.

6.6 <component type>

Function

Define a <component type>.

Format

```
<component type> ::=  
    <component name clause>  
    <data type>  
    [<occurs clause>]  
    [<default clause>]
```

Syntax Rules

None.

General Rules

None.

6.7 <component name clause>

Function

Name a <component type>.

Format

<component name clause> ::= ITEM <component name>

<component name> ::= <identifier>

Syntax Rules

- 1) The <component name> shall be different from the <component name> of any other <component name clause> in the containing <record type>.

General Rules

- 1) A <component name clause> defines the <identifier> to be a <component name> that designates the containing <component type>.