The Consultative Committee for Space Data Systems

Recommendation for Space Data System Standards

THE DATA DESCRIPTION LANGUAGE EAST SPECIFICATION (CCSD0010)

DRAFT RECOMMENDED STANDARD

CCSDS 644.0-P-2.1.1

DRAFT BLUE BOOK

September 2008

AUTHORITY

Issue:	Draft Recommended Standard, Issue 2.1.1
Date:	September 2008
Location:	Washington, DC, USA

This document has been approved for publication by the Management Council of the Consultative Committee for Space Data Systems (CCSDS) and represents the consensus technical agreement of the participating CCSDS Member Agencies. The procedure for review and authorization of CCSDS Recommendations is detailed in the *Procedures Manual for the Consultative Committee for Space Data Systems*, and the record of Agency participation in the authorization of this document can be obtained from the CCSDS Secretariat at the address below.

This document is published and maintained by:

CCSDS Secretariat
Space Communications and Navigation Office, 7L70
Space Operations Mission Directorate
NASA Headquarters
Washington, DC 20546-0001, USA

STATEMENT OF INTENT

The Consultative Committee for Space Data Systems (CCSDS) is an organization officially established by the management of its members. The Committee meets periodically to address data systems problems that are common to all participants, and to formulate sound technical solutions to these problems. Inasmuch as participation in the CCSDS is completely voluntary, the results of Committee actions are termed **Recommended Standards** and are not considered binding on any Agency.

This **Recommended Standard** is issued by, and represents the consensus of, the CCSDS members. Endorsement of this **Recommendation** is entirely voluntary. Endorsement, however, indicates the following understandings:

- o Whenever a member establishes a CCSDS-related **standard**, this **standard** will be in accord with the relevant **Recommended Standard**. Establishing such a **standard** does not preclude other provisions which a member may develop.

- o Whenever a member establishes a CCSDS-related **standard**, that member will provide other CCSDS members with the following information:

  - -- The **standard** itself.

  - -- The anticipated date of initial operational capability.

  - -- The anticipated duration of operational service.

- o Specific service arrangements shall be made via memoranda of agreement. Neither this **Recommended Standard** nor any ensuing **standard** is a substitute for a memorandum of agreement.

No later than five years from its date of issuance, this **Recommended Standard** will be reviewed by the CCSDS to determine whether it should: (1) remain in effect without change; (2) be changed to reflect the impact of new technologies, new requirements, or new directions; or (3) be retired or canceled.

In those instances when a new version of a **Recommended Standard** is issued, existing CCSDS-related member standards and implementations are not negated or deemed to be non-CCSDS compatible. It is the responsibility of each member to determine when such standards or implementations are to be modified. Each member is, however, strongly encouraged to direct planning for its new standards and implementations towards the later version of the Recommended Standard.

FOREWORD

Through the process of normal evolution, it is expected that expansion, deletion, or modification of this document may occur. This Recommended Standard is therefore subject to CCSDS document management and change control procedures, which are defined in the *Procedures Manual for the Consultative Committee for Space Data Systems*. Current versions of CCSDS documents are maintained at the CCSDS Web site:

http://www.ccsds.org/

Questions relating to the contents or status of this document should be addressed to the CCSDS Secretariat at the address indicated on page i.

At time of publication, the active Member and Observer Agencies of the CCSDS were:

Member Agencies

- Agenzia Spaziale Italiana (ASI)/Italy.
- British National Space Centre (BNSC)/United Kingdom.
- Canadian Space Agency (CSA)/Canada.
- Centre National d'Etudes Spatiales (CNES)/France.
- China National Space Administration (CNSA)/People's Republic of China.
- Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR)/Germany.
- European Space Agency (ESA)/Europe.
- Federal Space Agency (FSA)/Russian Federation.
- Instituto Nacional de Pesquisas Espaciais (INPE)/Brazil.
- Japan Aerospace Exploration Agency (JAXA)/Japan.
- National Aeronautics and Space Administration (NASA)/USA.

Observer Agencies

- Austrian Space Agency (ASA)/Austria.
- Belgian Federal Science Policy Office (BFSPO)/Belgium.
- Central Research Institute of Machine Building (TsNIIMash)/Russian Federation.
- Centro Tecnico Aeroespacial (CTA)/Brazil.
- Chinese Academy of Sciences (CAS)/China.
- Chinese Academy of Space Technology (CAST)/China.
- Commonwealth Scientific and Industrial Research Organization (CSIRO)/Australia.
- Danish National Space Center (DNSC)/Denmark.
- European Organization for the Exploitation of Meteorological Satellites (EUMETSAT)/Europe.
- European Telecommunications Satellite Organization (EUTELSAT)/Europe.
- Hellenic National Space Committee (HNSC)/Greece.
- Indian Space Research Organization (ISRO)/India.
- Institute of Space Research (IKI)/Russian Federation.
- KFKI Research Institute for Particle & Nuclear Physics (KFKI)/Hungary.
- Korea Aerospace Research Institute (KARI)/Korea.
- MIKOMTEK: CSIR (CSIR)/Republic of South Africa.
- Ministry of Communications (MOC)/Israel.
- National Institute of Information and Communications Technology (NICT)/Japan.
- National Oceanic and Atmospheric Administration (NOAA)/USA.
- National Space Organization (NSPO)/Chinese Taipei.
- Naval Center for Space Technology (NCST)/USA.
- Space and Upper Atmosphere Research Commission (SUPARCO)/Pakistan.
- Swedish Space Corporation (SSC)/Sweden.
- United States Geological Survey (USGS)/USA.

DOCUMENT CONTROL

Document	Title and Issue	Date	Status
CCSDS 644.0-B-1	Recommendation for Space Data System Standards: The Data Description Language EAST Specification (CCSD0010), Issue 1	May 1997	Original Issue: superseded.
CCSDS 644.0-B-2	Recommendation for Space Data System Standards: The Data Description Language EAST Specification (CCSD0010), Issue 2	November 2000	Current Issue: – extends EAST ability to handle repeated data items where repetition is terminated by a marker.
CCSDS 644.0-P-2.1.1	The Data Description Language EAST Specification (CCSD0010), Draft Recommended Standard, Issue 2.1.1	September 2008	Current draft update: – adds requirement to specify EAST version

CONTENTS

Section Page

1 INTRODUCTION 1-1

1.1 PURPOSE AND SCOPE 1-1
1.2 APPLICABILITY 1-1
1.3 RATIONALE 1-1
1.4 DOCUMENT STRUCTURE 1-2
1.5 DEFINITIONS 1-2

1.5.1 TERMS 1-2
1.5.2 NOMENCLATURE 1-2
1.5.3 CONVENTIONS 1-3

1.6 REFERENCES 1-5

2 OVERVIEW 2-1

2.1 DESIGN AIMS 2-1
2.2 STRUCTURE OF AN EAST DESCRIPTION 2-1
2.3 LANGUAGE SUMMARY 2-2

3 DEFINITION OF THE EAST LANGUAGE 3-1

3.1 LEXICAL ELEMENTS 3-1

3.1.1 SEPARATORS AND DELIMITERS 3-1
3.1.2 COMMENTS 3-1
3.1.3 IDENTIFIERS 3-2
3.1.4 NUMERIC LITERALS 3-2

3.2 LOGICAL DESCRIPTION 3-7

3.2.1 TYPE DECLARATIONS 3-8
3.2.2 SUBTYPE DECLARATIONS 3-28
3.2.3 OBJECT DECLARATIONS 3-31
3.2.4 REPRESENTATION CLAUSES 3-35

3.3 PHYSICAL DESCRIPTION 3-47

3.3.1 STORING ARRAYS 3-48
3.3.2 STORING OCTETS/BITS 3-48

CONTENTS (continued)

Section Page

3.3.3 REPRESENTATION OF SCALAR TYPES 3-50
3.3.4 RELATIONSHIP BETWEEN THE REPRESENTATION OF SCALAR TYPES AND LOGICAL TYPES 3-62
3.3.5 TEMPLATE OF A PHYSICAL DESCRIPTION PART 3-64

4 RESERVED KEYWORDS 4-1

5 CONFORMANCE 5-1

ANNEX A ACRONYMS AND GLOSSARY A-1
ANNEX B CHARACTER DEFINITION B-1
ANNEX C EAST FORMAL SYNTAX SPECIFICATION C-1
ANNEX D MAIN DIFFERENCES BETWEEN ADA AND EAST D-1
ANNEX E INFORMATIVE REFERENCES E-1
INDEX I-1

Figure

1-1 Example of Syntax Diagram 1-3
3-1 Identifier Definition Diagram 3-2
3-2 Decimal Literal Definition Diagram 3-2
3-3 Integer Decimal Literal Definition Diagram 3-3
3-4 Real Decimal Literal Definition Diagram 3-3
3-5 Integer Definition Diagram 3-3
3-6 Exponent Definition Diagram 3-4
3-7 Based Literal Definition Diagram 3-4
3-8 Integer Based Literal Definition Diagram 3-5
3-9 Real Based Literal Definition Diagram 3-5
3-10 Based Integer Definition Diagram 3-5
3-11 Integer Literal Definition Diagram 3-6
3-12 Real Literal Definition Diagram 3-6
3-13 Logical Part Structure 3-8
3-14 Enumeration Type Specification Diagram 3-9
3-15 Enumeration Literal Definition Diagram 3-9
3-16 Integer Type Specification Diagram 3-10
3-17 Real Type Specification Diagram 3-11
3-18 Array Type Specification Diagram 3-12
3-19 Index Specification Diagram 3-13
3-20 Record Type Specification Diagram 3-15
3-21 Component Declaration Diagram 3-15
3-22 Default Value Definition Diagram 3-16
3-23 Index Constraint Diagram 3-17

CONTENTS (continued)

Figure | Page

3-24 Discriminant Specification Diagram 3-18
3-25 Variant Part Specification Diagram 3-19
3-26 Discriminants in a Packet Format 3-22
3-27 Actual Discriminant Value Declaration Diagram 3-26
3-28 Type Summary 3-27
3-29 Subtype Declaration Diagram 3-28
3-30 Enumeration Constraint Diagram 3-28
3-31 Integer Constraint Diagram 3-29
3-32 Real Constraint Diagram 3-30
3-33 Variable Declaration Diagram 3-31
3-34 Constant Declaration Diagram 3-32
3-35 Length Clause Specification Diagram 3-35
3-36 Enumeration Clause Specification Diagram 3-37
3-37 Component Representation Clause Specification Diagram 3-38
3-38 Record Representation Clause Specification Diagram 3-39
3-39 First Tree Structure 3-40
3-40 Second Tree Structure 3-41
3-41 Third Tree Structure 3-42
3-42 Fourth Tree Structure 3-44
3-43 Distance Specification Diagram 3-46
3-44 Record Value Specification Diagram 3-54
3-45 Component Value Definition Diagram 3-54
3-46 Array Value Specification Diagram 3-55
3-47 ASCII Encoded Decimal Integer Format 3-60
3-48 ASCII Encoded Decimal Real Format 3-61

Example

1-1 Example of BNF 1-4
3-1 Decimal Literals 3-4
3-2 Based Literals 3-6
3-3 Enumeration Type Declarations 3-10
3-4 Integer Type Declarations 3-10
3-5 Real Type Declarations 3-11
3-6 Constrained Array Type Definitions 3-13
3-7 Unconstrained Array Type Definitions 3-14
3-8 Record Type Definitions 3-18
3-9 Record Type Definition with Discriminant 3-20
3-10 Record Type Definition with Discriminant 3-20
3-11 Logical Description of the Packet Format 3-24

CONTENTS (continued)

Example Page

3-12 Calculated Size Array ... 3-25
3-13 Calculated Component Presence Condition ... 3-26
3-14 Character Declarations ... 3-29
3-15 Subtype Declarations ... 3-30
3-16 Variable Declaration ... 3-31
3-17 Constant Declaration ... 3-32
3-18 Number Declarations ... 3-33
3-19 Marker Declaration ... 3-34
3-20 EOF Marker Declaration ... 3-34
3-21 Marker Declaration in Record Definition ... 3-35
3-22 Length Clause Declarations ... 3-36
3-23 Explicit Description of Unused Space ... 3-36
3-24 Enumeration Clause Declarations ... 3-37
3-25 Type Definitions ... 3-40
3-26 Complete Record Representation Clause Declaration ... 3-41
3-27 Incomplete Record Representation Clause Declaration ... 3-42
3-28 Complete Record Representation Clause Declaration ... 3-43
3-29 Complete Record Representation Clause Declaration ... 3-45
3-30 Record Representation Clause Using WORD_32_BITS ... 3-46
3-31 Actual Array Storage Method ... 3-48
3-32 Octet Storage Possibilities ... 3-49
3-33 Actual Bit Order ... 3-50
3-34 Bit Ordering ... 3-52
3-35 Bit Ordering for the Above 16-Bit Signed Integer ... 3-55
3-36 Actual Binary Representation of the Above 16-Bit Signed Integer ... 3-55
3-37 Bit Ordering for the Above 16-Bit Unsigned Integer ... 3-56
3-38 Actual Binary Representation of the Above 16-Bit Unsigned Integer ... 3-56
3-39 Bit Ordering for the Above 32-Bit Real ... 3-57
3-40 Actual Binary Representation of a 32-Bit Real ... 3-57
3-41 ASCII Enumeration Type Logical Declaration ... 3-59
3-42 ASCII Enumeration Type Physical Description ... 3-60
3-43 ASCII Integer Type Logical Declaration ... 3-62
3-44 ASCII Integer Type Physical Description ... 3-62
3-45 ASCII Real Type Logical Declaration ... 3-62
3-46 ASCII Real Type Physical Description ... 3-62

1 INTRODUCTION

1.1 PURPOSE AND SCOPE

The purpose of this document is to establish a common Recommendation for the specification of a standard language for describing and expressing data in order to interchange them in a more uniform and automated fashion within and among Agencies participating in the Consultative Committee for Space Data Systems (CCSDS).

This Recommendation defines the Enhanced Ada SubseT (EAST) language used to create descriptions of data, called Data Description Records (DDRs). Such DDRs ensure a complete and exact understanding of the data and allow it to be interpreted in an automated fashion. This means that a software tool is able to analyze a DDR and interpret the format of the associated data. This allows the software to extract values from the data on any host machine (i.e., on a different machine from the one that produced the data).

A first look at reference [E4], which is a tutorial for the EAST language, may aid the user in understanding this document. Reference [E4] describes the requirements, explains how to use the EAST language to describe non-ambiguous data, and suggests practices and tools to the users.

This Recommendation is registered under the CCSDS Authority and Description Identifier (ADID): CCSD0010.

1.2 APPLICABILITY

The specifications in this document are applicable to all space-related science and engineering data exchanges where data descriptions are desired, and these descriptions need to provide an unambiguous description of the record structure of the data.

1.3 RATIONALE

The Consultative Committee for Space Data Systems has defined the Standard Formatted Data Unit (SFDU) concept for the implementation of standard data structures to be used for the interchange of data within and among space agencies.

SFDU data products may be viewed as containing application data (that is the data which is of primary interest, e.g., actual measurements) and data description information (that is the information telling how the application data are formatted).

The data description information shall be provided in a form that is understandable by the agencies involved in the data interchange. That is the reason why the CCSDS must provide some recommendations for the definition of standard description languages. EAST is one of the recommended languages.

1.4 DOCUMENT STRUCTURE

The Recommendation is structured as follows:

– Section 2 provides an overview of the EAST language.

– Section 3 specifies the EAST language and defines its usage in Data Descriptions.

– Section 4 lists the EAST reserved keywords.

– Annex A contains acronyms and the glossary of terms used in this document.

– Annex B defines the character set to be used in an EAST data description, as well as a predefined type called CHARACTER.

– Annex C provides the EAST formal specification using a simple variant of the Backus-Naur-Form (BNF).

– Annex D lists the main differences between the Ada programming language and EAST.

– Annex E lists the informative references.

1.5 DEFINITIONS

1.5.1 TERMS

The terms used throughout this document are listed in annex A. They are also explained in the text when they are first used.

1.5.2 NOMENCLATURE

The following conventions apply throughout this Recommendation:

a) the words ‘shall’ and ‘must’ imply a binding and verifiable specification;

b) the word ‘should’ implies an optional, but desirable, specification;

c) the word ‘may’ implies an optional specification;

d) the words ‘is’, ‘are’, and ‘will’ imply statements of fact.

1.5.3 CONVENTIONS

This document uses syntax diagrams to illustrate the syntax of the EAST constructs. Components of a construct are called elements. The following conventions are used:

a) Elements that are presented in bold characters in a circle are reserved keywords, delimiters, or literals.

b) The item named on the left of the **::=** symbol is the item being defined.

c) The diagram on the right of the **::=** symbol is the corresponding definition.

d) A vertical branch represents a choice.

e) A repetition is indicated by a loop-back covering the object to be repeated.

f) If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, an '*Integer* Identifier' is an Identifier; i.e., the definition of the category Identifier applies, but the reader has additional semantic information (it is an integer).

The following example (figure 1-1) presents a diagram specifying the declaration of Item A. Item A is defined as first a keyword ('type'), then followed by an italicized Item B (already defined, and known as Item B), then followed by a keyword ('is') and a delimiter ('('). Then this structure is followed by a choice between Items B and C. The choice may be repeated any number of times, each time a delimiter (',') is inserted. The structure is ended by two delimiters (')' and ';').

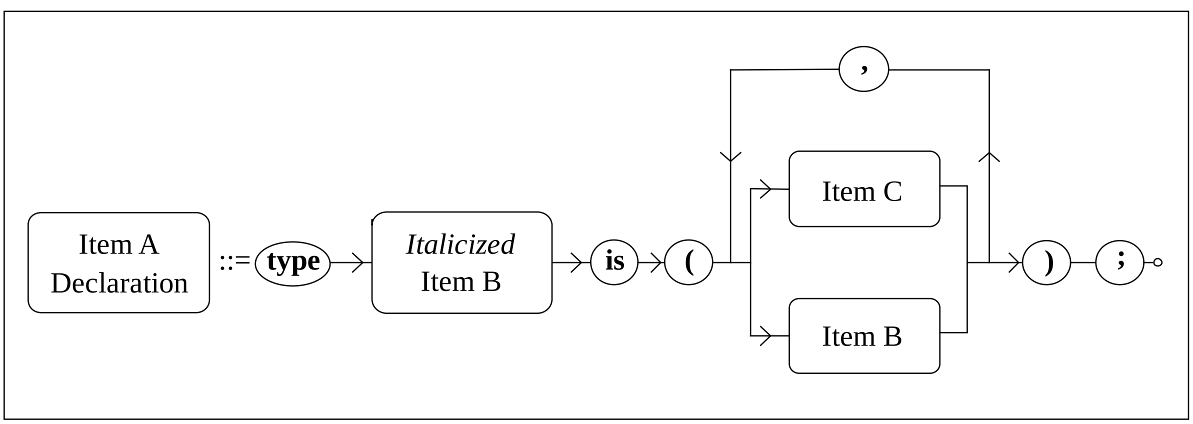


Figure 1-1: Example of Syntax Diagram

The syntax of the language is described using a simple variant of Backus-Naur-Form with the following conventions:

a) Boldface words are used to denote reserved keywords.

b) Square brackets enclose optional items.

c) Braces enclose a repeated item. This item may appear zero or more times.

d) A vertical bar separates alternative items unless it occurs immediately after an opening brace ({): in this case it represents the character ‘vertical bar’.

e) If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. This facility used for the BNF intends to assimilate every element like <*italicized_part*_name> to the previously defined element <name>.

The following example presents the definition of Item A using a simple variant of BNF. Item A is defined as first a keyword (‘type’), then followed by an italicized Item B (already defined, and known as Item B), then followed by a keyword (‘is’) and a delimiter (‘(’). The structure is followed by a choice. The choice may be repeated any number of times, each time a delimiter (‘,’) is inserted. The structure is ended by two delimiters (‘)’ and ‘;’). The choice is between Items B and C.

<Item A> ::= **type** <*Italicized*_Item B> **is (** <choice> { , <choice> } **) ;**
<choice> ::= <Item B> | <Item C>

Example 1-1: Example of BNF

In the case of any confusion, the syntax diagram and the associated text are always the reference for the EAST syntax, and not the BNF.

This document uses examples to illustrate the EAST. The following conventions are used in the examples:

a) bold characters denote reserved keyword or delimiters;

b) user type names or user variable names are provided using uppercase letters, although EAST is not a case-sensitive language.

1.6 REFERENCES

The following documents contain provisions which, through reference in this text, constitute provisions of this Recommended Standard. At the time of publication, the editions indicated were valid. All documents are subject to revision, and users of this Recommended Standard are encouraged to investigate the possibility of applying the most recent editions of the documents indicated below. The CCSDS Secretariat maintains a register of currently valid CCSDS Recommended Standards.

[1] *Information Processing—8-Bit Single-Byte Coded Graphic Character Sets—Part 1: Latin Alphabet No. 1*. International Standard, ISO 8859-1:1987. Geneva: ISO, 1987.

[2] *Information Processing—Universal Multiple-Octet Coded Character Set (UCS)*. International Standard, ISO/IEC 10646-1:1993.

2 OVERVIEW

2.1 DESIGN AIMS

EAST was designed with three overriding concerns: data description capabilities, human readability, and computer interpretability.

The need for data description languages that supply complete and non-ambiguous information about the format and the nature of the described data is well established.

Any user must be able to understand descriptions of data, with a minimal effort. Error-prone notations have been avoided, and the syntax of the EAST language avoids the use of cryptic forms in favor of more English-like constructs.

EAST is a formal language and not a natural language: it is a machine compilable (or interpretable) language. The formal nature of EAST allows the control of data descriptions and the interpretation of data in an automated fashion.

2.2 STRUCTURE OF AN EAST DESCRIPTION

An EAST Data Description Record (DDR) includes a syntactic, and in some way semantic, description of the data called a logical description, which is followed by a physical description. The physical description makes possible the interpretation of the actual bit patterns encountered on the medium. Each description part of a DDR consists of an EAST unit, called a package: one for the logical part and another one for the associated physical part.

The logical part of an EAST description includes:

– a logical description of all components of the exchanged data (see 3.2.1 and 3.2.2);

– their size in bits (see 3.2.4.1);

– their location within the set of the described data (see 3.2.4.3).

The physical part of an EAST description includes:

– the representation of some basic data types (enumeration, integer, and real) defined in the logical description and dependent on the machine that has generated the data (see 3.3.3);

– the array organization (first-index-first or last-index-first) used by the generating machine (see 3.3.1);

– the octet and bit organization on the medium (high-order-first or low-order-first—see 3.3.2).

A DDR created using the EAST Language has the following structure:

Data Description Record

package *name_of_the_logical_description* **is**

Logical Description (see 3.2)

end *name_of_the_logical_description* **;**

package *name_of_the_physical_description* **is**

Physical Description (see 3.3)

end *name_of_the_physical_description* **;**

The logical description always precedes the physical description. The logical and the physical packages are mandatory even if the content of the physical one can be empty (see 3.3).

The two part design of the DDR is intended to allow interchangeable physical description parts for one logical description part, provided that the length of fields in bits in the logical description are supported by field lengths of the same number of bits in the physical description part. For example, a 32 bit real number on a IEEE architecture has a physical description different from the one on a 1750 architecture, although lengths in bits of each field are equal. Note that the representations written to an exchange medium do not have to be the ones typically supported by the writing machine.

The data block associated with the DDR contains one or more complete sets of data. The DDR describes a single set only and is repetitively applied to fully interpret the data block.

2.3 LANGUAGE SUMMARY

An EAST description is composed of two units, called packages. The first one is a *logical* description and the second one is a *physical* description of the data. The logical part of an EAST description provides syntactic information and in some way semantic information, i.e., the information needed by a user to understand the data he has to deal with. The physical part of an EAST description provides a bit-level description that ensures the non-ambiguous interpretation of the data.

The syntax used in each of the two packages is based on the type and object concept. A type is a model, defined once, that is used to create many occurrences (objects) of the models.

Every data item described in an EAST description is an object. An object in the language has a type, which characterizes a set of values. The basic classes of types are *scalar types* (comprising *enumeration* and *numeric types*, describing single elements), and *composite types* (comprising *array* and *record types*, describing sequences of objects).

A type has a name: if well chosen, this name is a way to provide the meaning of the model (e.g., the type DATE may describe a CCSDS date). An object has a name also: this name is a way to provide (if any) the particularity of the occurrence (e.g., the object DATE_AT_THE_BEGINNING_OF_THE_ORBIT of the type DATE may represent a particular date). The name used to identify a type or an object can be any identifier except for an EAST reserved keyword (reserved keywords are provided in section 4).

An enumeration type defines an ordered set of distinct enumeration literals; for example, a Boolean type defines two enumeration literals (TRUE and FALSE). The enumeration type CHARACTER is predefined and given in 3.2.1.1.

Numeric types provide a means of describing whole numbers and real numbers. Whole numbers are described using integer types. Real numbers are described using floating point types, with relative bounds on the error.

Composite types allow definitions of structured objects with related components. The composite types of the EAST language are arrays and records. An array is an object with indexed components of the same type. The array type STRING is predefined and given in 3.2.1.1. A record is an object with named components of possibly different types.

A record may have special components called *discriminants*. Discriminants specify either which of alternative record structures is to be used or the dynamic size of an internal array (depending on the values of the discriminants).

The concept of type is refined by the concept of *subtype*, whereby a user can constrain the set of allowed values of a type. Subtypes can be used to define subranges of scalar types and arrays with a limited set of index values.

Representation clauses are used to specify the mapping between logical types and their physical representations. For example, the user specifies that objects of a given type are represented with a given number of bits, or the components of a record are represented using a given storage layout.

NOTES

1 EAST is a subset of the Ada programming language (reference [E3]). EAST contains therefore most of the declarative features of Ada, but no algorithmic features.

2 The declarative part of Ada normally defines the logical entities and sometimes some of their physical characteristics. EAST extends the descriptive power of the Ada language (using conventions in the physical packages). It is able to describe not only the logical aspects of a data item, but also all its physical aspects.

3 DEFINITION OF THE EAST LANGUAGE

An EAST Data Description is a text composed of lexical elements, each composed of ASCII characters: the 128 first characters of the Latin Alphabet No. 1 (see reference [1] and/or annex B). The rules of composition are given in 3.1. They are applicable to the whole EAST DDR.

3.1 LEXICAL ELEMENTS

A **lexical element** is either a delimiter, an identifier (which may be a reserved word), a numeric literal, a character string, a string literal, or a comment. The rules of composition are given in this section.

3.1.1 SEPARATORS AND DELIMITERS

In some cases an explicit separator is required to separate adjacent lexical elements (namely, when without separation, interpretation as a single lexical element is possible). A **separator** is any of a space character, a control character, or the end of a line.

– A space character is a separator except within a comment, a string literal, or a space character literal.

– Control characters other than horizontal tabulation are always separators. Horizontal tabulation is a separator except within a comment.

– The end of a line is always a separator. What defines the end of a line is specified in annex B.

A **delimiter** is either one of the following special characters:

& ’ ( ) * + , - . / : ; < = > |

or one of the following compound delimiters, each composed of two adjacent special characters:

=> .. ** := /= >= <= << >> <>

Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter, or as a character of a comment, string literal, character literal, or numeric literal.

The remaining forms of lexical elements are described in 3.1.2, 3.1.3 and 3.1.4.

3.1.2 COMMENTS

A comment starts with two adjacent hyphens and extends up to the end of the line. A comment can appear on any line of a description.

3.1.3 IDENTIFIERS

Identifiers are used as names and also as reserved words. See figure 3-1 for the lexical definition of an identifier:

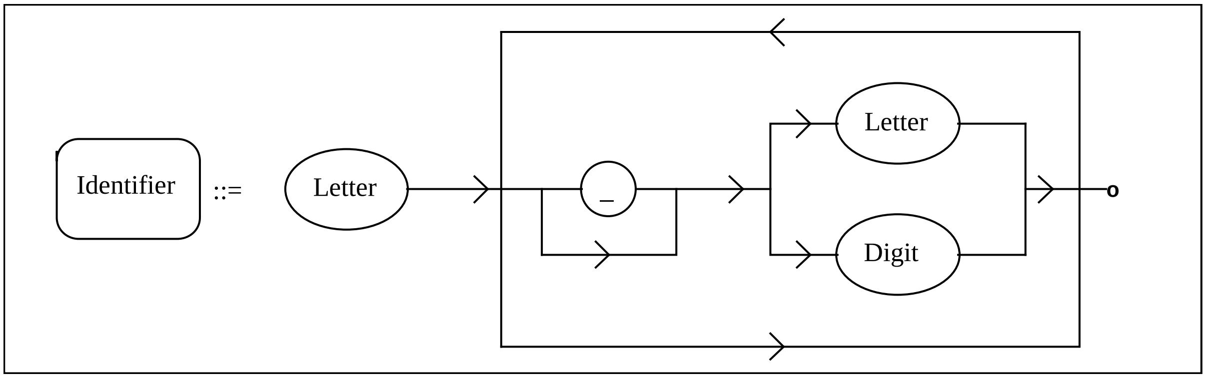


Figure 3-1: Identifier Definition Diagram

All characters of an identifier are significant, including any underline character inserted between a letter or a digit and an adjacent letter or digit. Identifiers differing in the use of corresponding upper and lower case letters are considered to be the same.

3.1.4 NUMERIC LITERALS

A numeric literal is either a decimal literal or a based literal. A decimal literal is a numeric literal expressed in the conventional decimal notation (that is, the base is implicitly ten). A based literal is a numeric literal expressed in a form that specifies the base explicitly. The base can only be either two, eight, or sixteen.

In another way, a numeric literal is either an integer literal (decimal or based) or a real literal (decimal or based). See figure 3-2.

a) decimal literals

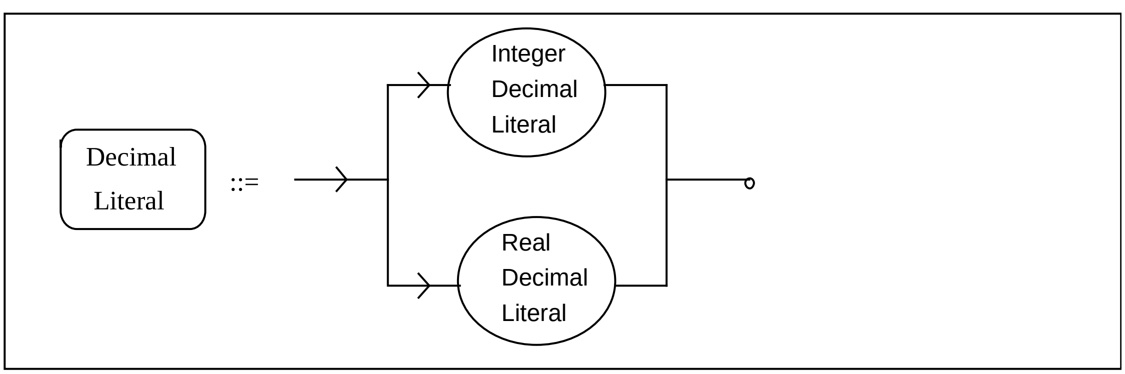


Figure 3-2: Decimal Literal Definition Diagram

where Integer Decimal Literal and Real Decimal Literal are defined as in figures 3-3 and 3-4:

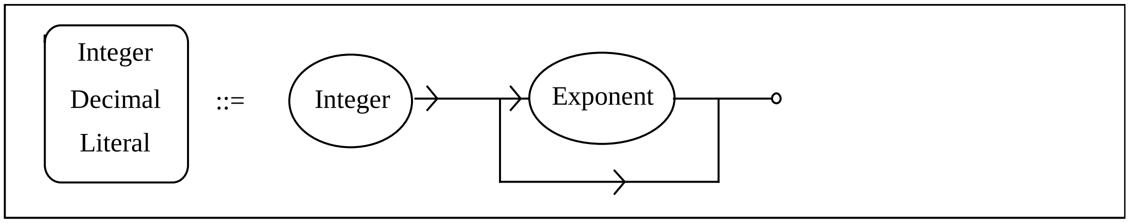


Figure 3-3: Integer Decimal Literal Definition Diagram

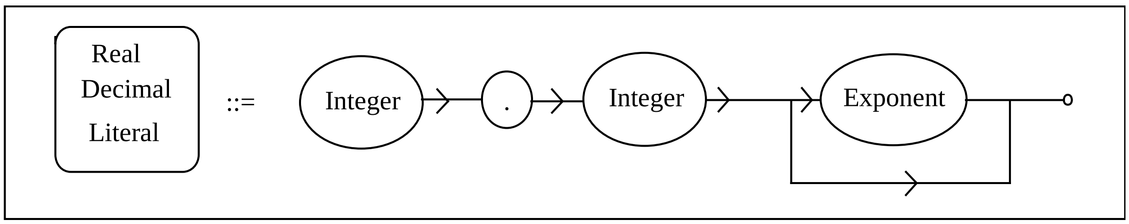


Figure 3-4: Real Decimal Literal Definition Diagram

where Integer and Exponent are defined as in figures 3-5 and 3-6:

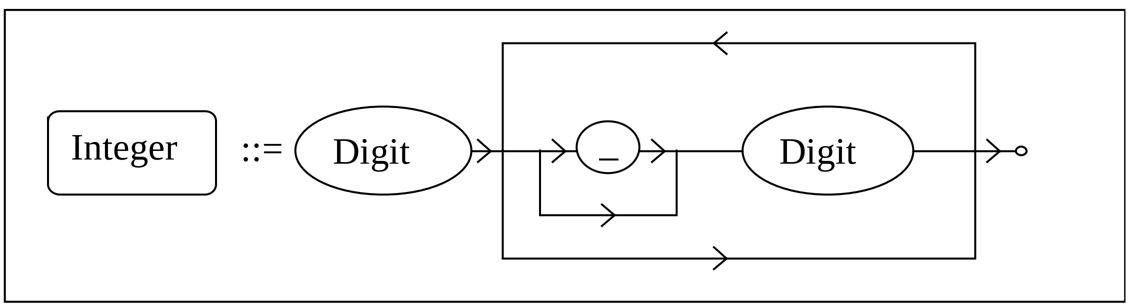


Figure 3-5: Integer Definition Diagram

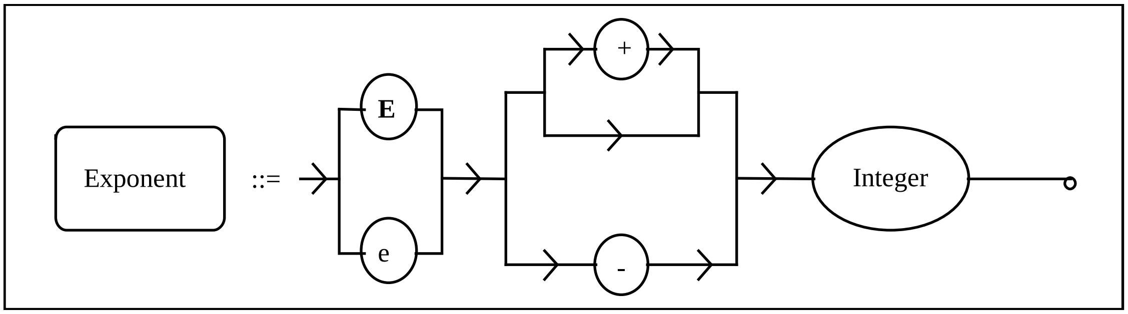


Figure 3-6: Exponent Definition Diagram

An underline character inserted between adjacent digits of a decimal literal does not affect the value of this decimal literal. The letter E of the exponent, if any, can be written either in lowercase or in uppercase, with the same meaning. Leading zeros are allowed. No space is allowed in a decimal literal.

12	0	1E6	123_456	-- integer literals
12.0	0.0	0.456	3.14159_26	-- real literals
1.3E-12	1.0E+6			-- real literals with exponent

Example 3-1: Decimal Literals

b) based literals

See figure 3-7.

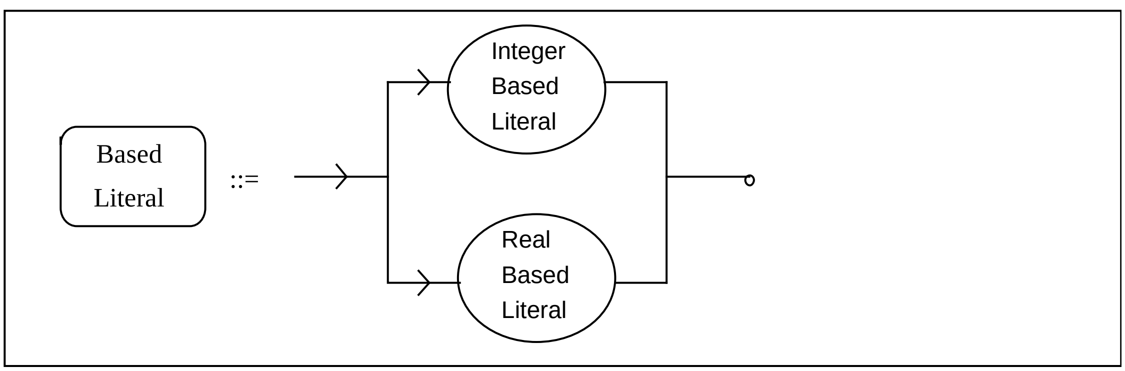


Figure 3-7: Based Literal Definition Diagram

where Integer Based Literal and Real Based Literal are defined as in figure 3-8 and figure 3-9:

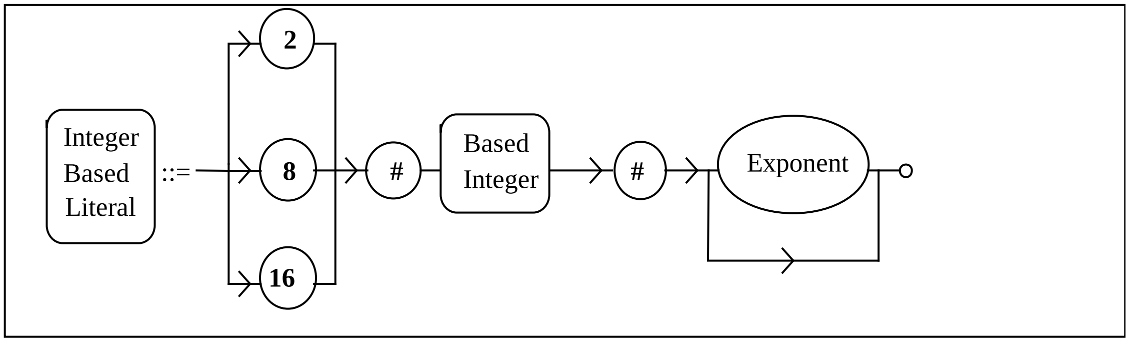


Figure 3-8: Integer Based Literal Definition Diagram

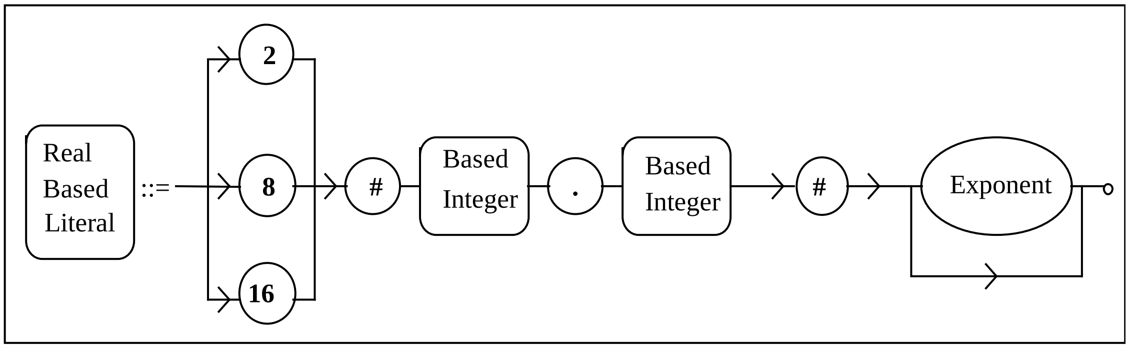


Figure 3-9: Real Based Literal Definition Diagram

where Based Integer is defined as in figure 3-10:

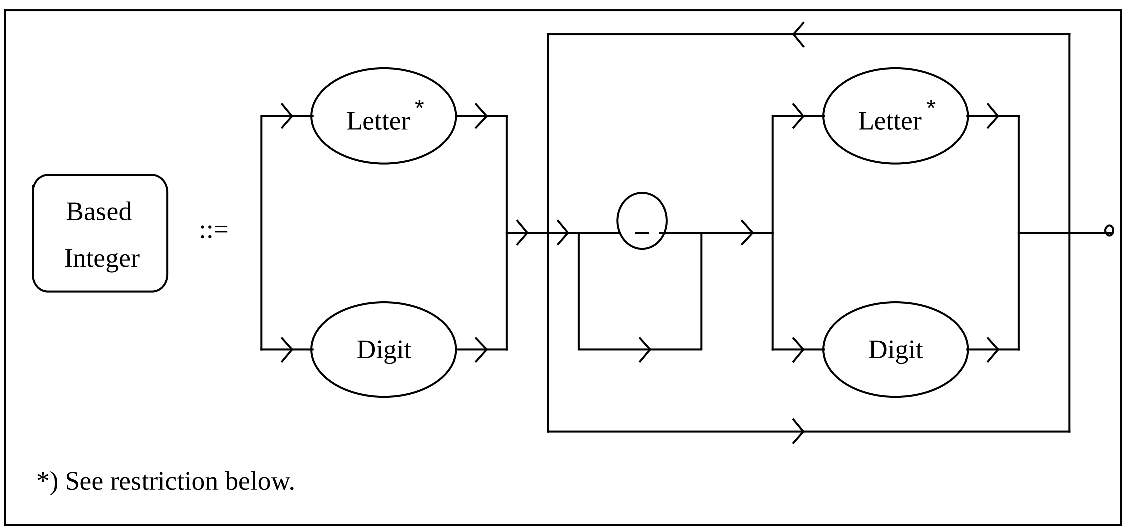


Figure 3-10: Based Integer Definition Diagram

The only letters allowed as extended digits are the letters A through F representing ten through fifteen. Letters are allowed for a based integer only if the base of the literal of which it is a part is 16. A letter in a based literal can be written either in lowercase or in uppercase, with the same meaning. No space is allowed in a based literal.

```
2#1111_1111#          16#FF#        016#0FF#    -- integer literals of value 255
   16#E#E1      2#1110_0000#                    -- integer literals of value 224

 16#F.FF#E+2          2#1.1111_1111_111#E11     -- real literals of value 4095.0
```

Example 3-2: Based Literals

c) integer literals

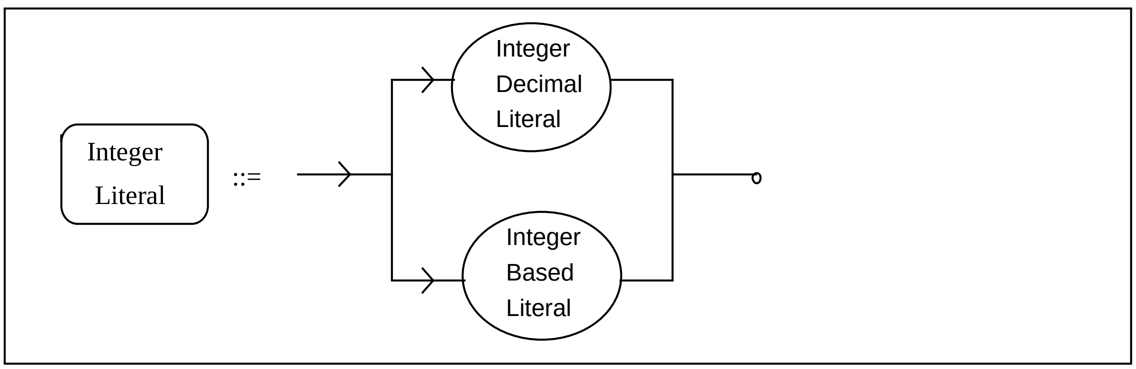


Figure 3-11: Integer Literal Definition Diagram

c) real literals

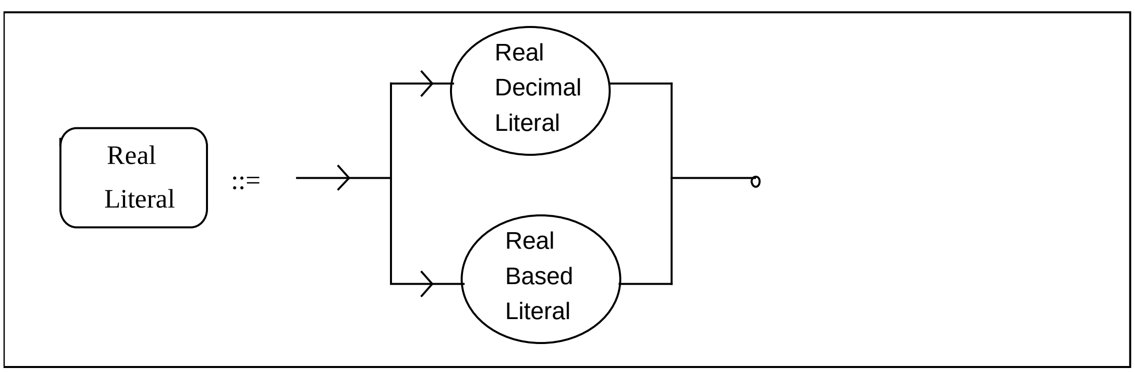


Figure 3-12: Real Literal Definition Diagram

3.2 LOGICAL DESCRIPTION

The logical part of an EAST DDR is composed of:

– the logical description of the models of data (using type and subtype declarations for the syntactic definition of the data, and using representation clauses for the specification of their size in bits and their location within the set of data);

– the declaration of the data occurrences, i.e., the declaration of the described data items (using object declarations).

The logical part of the Data Description Record consists of a package. This unit is introduced by the keyword **package**, followed by the package name, and ends with '**end** package name**;**'. The package name is an identifier (see 3.1.3).

The logical description package identification must be followed by the mention of the version of the EAST recommendation to which the description is supposed to conform.

As the notion of EAST recommendation version was not present in the first two EAST recommendation issues, the absence of the mention in a description should be interpreted as a reference to these two first versions (fully compatible).

A description that conforms to a particular version of the EAST recommendation must remain correct with regard to the following versions of EAST.

If an EAST description is generated using a tool, it is recommended that the tool indicate its own version using a comment.

Types are models, and objects are instances (or occurrences) of these models. Type declarations describe therefore the structure of the data elements which may occur in the described data, while **the actual data occurrences are represented by the declaration of variables and constants**.

A type (except predefined type), a subtype or a constant (except predefined constant) must be declared in the package before being used.

The declaration of variables must occur in the latter section of the logical description. Constants may be declared in the type declaration section or in the section for the declaration of variables: in the first section, they contribute to data models definition, while they represent data occurrences in the second section.

The described data is a concatenation of elements in the order of the corresponding variables. The types used in the declaration of variables must have been previously declared in the package.

Figure 3-13 summarizes the content of the logical part of a DDR.

package *logical_package_name* **is**

EAST version and -- tool version (optional)

Section for the Declaration of Types: Definition of the Data Models

- type declarations and representation clauses (see 3.2.1 and 3.2.4)
- subtype and constant declarations (see 3.2.2 and 3.2.3.2)

Section for the Declaration of Variables: Definition of the Data Occurrences

- variable and constant declarations (see 3.2.3.1 and 3.2.3.2)
- actual values of discriminant (see 3.2.1.6)

end *logical_package_name* ;

Figure 3-13: Logical Part Structure

The version declaration should respect the following format:

```
east_version : constant STRING := "3.0";

-- tool version : OASIS 5.0 (optional comment)
```

3.2.1 TYPE DECLARATIONS

The type is characterized by a set of permissible values. Several classes of types exist: scalar types (enumeration types, integer types, and real types), array types, and record types. Some types are EAST predefined types (see 3.2.1.1); the other types are user defined types and must be declared according to a specific syntax (see 3.2.1.2, 3.2.1.3, 3.2.1.4, 3.2.1.5 and 3.2.1.6).

3.2.1.1 Predefined Types

There are three predefined types provided by the EAST language: CHARACTER, STRING and EOF. Predefined means that no previous declaration has to be made explicitly by the user to use one of these types.

The predefined type CHARACTER is an enumeration type (see next subsection for the enumeration definition syntax rules), whose values are the 256 characters of the 8-bit coded Latin Alphabet No. 1 character set (see annex B and reference [1]).

The values of the predefined type STRING are one-dimensional arrays of the predefined type CHARACTER, indexed by values in increments of one of any positive integer type.

The number of characters must be specified every time the type is used.

As an example STRING(1 .. 10) designates a 10 character string, while STRING(10 .. 22) designates a 13 character string.

The predefined type EOF is exclusively used to declare a fictive end delimiter called EOF Marker (see 3.2.3.2.2).

3.2.1.2 Enumeration Type

An enumeration type is defined as a set of enumeration literals. An enumeration literal is an identifier or a character literal for one of the possible values of the type. Figure 3-14 illustrates the syntax of an enumeration type specification. Each enumeration literal yields a different enumeration value.

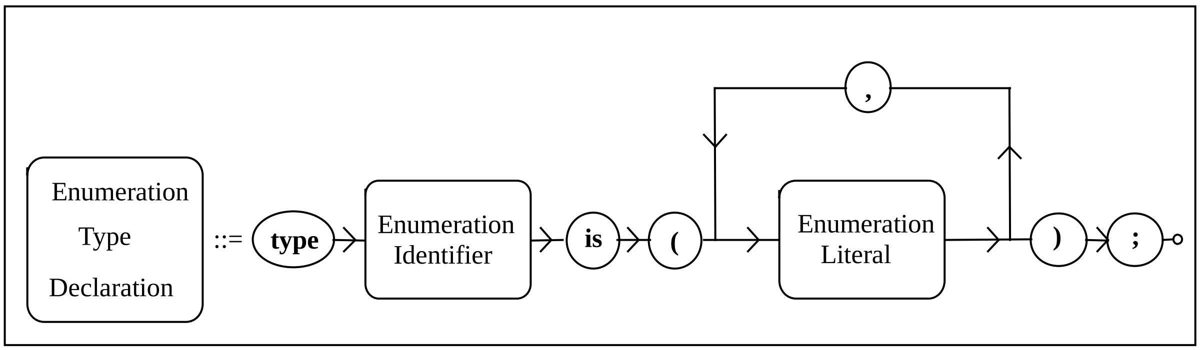


Figure 3-14: Enumeration Type Specification Diagram

where the enumeration literal is defined as in figure 3-15:

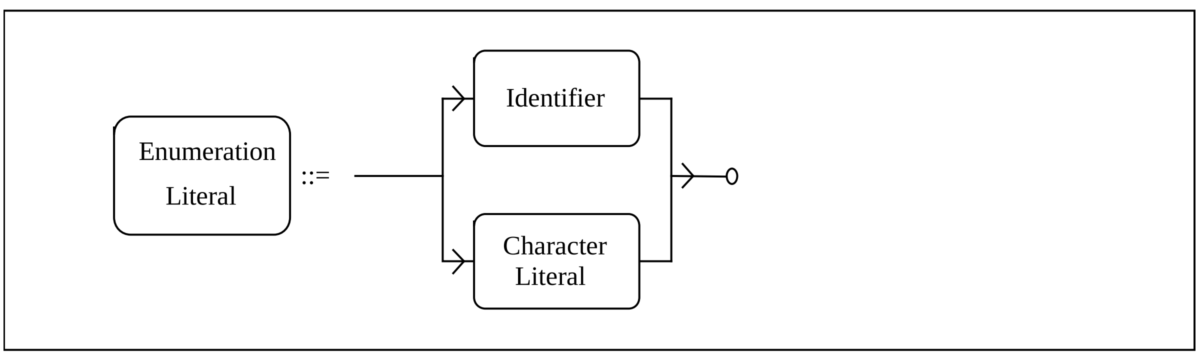


Figure 3-15: Enumeration Literal Definition Diagram

The following example presents some enumeration type definitions.

type DAY **is** (MON, TUE, WED, THU, FRI, SAT, SUN)**;**
type STATE **is** (OFF , ON);
type ROMAN_DIGIT **is** (‘I’ , ‘V’ , ‘X’ , ‘L’ , ‘C’ , ‘D’ , ‘M’);

Example 3-3: Enumeration Type Declarations

3.2.1.3 Integer Type

An integer type is defined as a set of integer values specified by a range. Each bound of the range is an integer constant identifier (see 3.2.3.2) or an integer literal (see 3.1.4). Note that both bounds need not have the same integer type and that negative bounds are allowed. The range L .. R specifies the value from L to R inclusive if the relation L <= R is true. A *null* range is a range for which the relation R < L is true; no value belongs to a null range.

Figure 3-16 illustrates the syntax of an integer type specification.

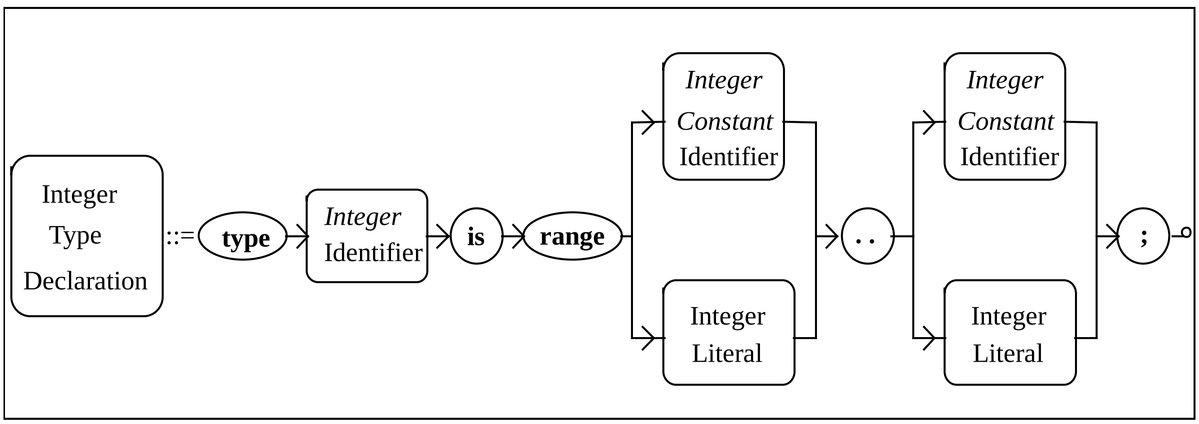


Figure 3-16: Integer Type Specification Diagram

The following example presents an integer type, defined using integer literals (-10 and 10) and an integer type, defined using a constant identifier (MAX).

type SMALL_INTEGER **is range** -10 **..** 10**;**
type NUMBER **is range** 0 **..** MAX**;**
-- where MAX could be defined as: MAX := 100;

Example 3-4: Integer Type Declarations

3.2.1.4 Real Type

Real types provide approximations to real numbers, with relative bounds on errors. The error bound is specified as a relative precision by giving the required minimum number of significant decimal digits. The range bounds are optional. When they are specified, they are either real constant identifier (see 3.2.3.2) or real literal (see 3.1.4).

Figure 3-17 illustrates the syntax of a real type specification.

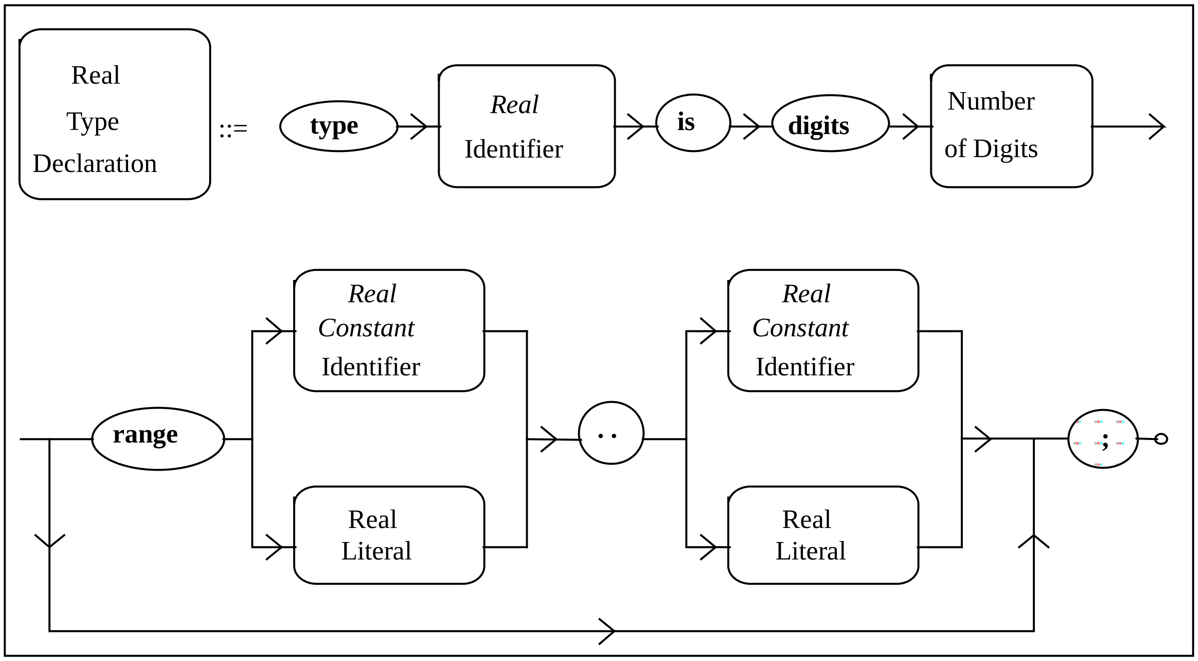


Figure 3-17: Real Type Specification Diagram

The following example presents some real type definitions.

type COEFFICIENT **is digits** 10 **range** 0.1 **..** 1.0**;**
type REAL **is digits** 15**;**

Example 3-5: Real Type Declarations

NOTE – The range is optional in a real type declaration. If the real type declaration specifies no range, then the range is supposed to be the largest range that can be implemented within the specified number of bits (see 3.2.4.1) accommodating the number of significant digits. When unspecified, the range will depend on the convention used to represent the binary values of the real type (see 3.3.3.1).

3.2.1.5 Array Type

An array type is a composite type consisting of components that have the same type. The name for a component of an array uses one or more index values belonging to specified discrete types.

A discrete type is either an enumeration type or an integer type.

An array type is characterized by:

– an ordered list of indices;

– the type of each index;

– the lower and upper bound for each index;

– the type of the components.

The order of indices is significant. The index type and component type declarations must precede the array type declaration that makes use of them, except if one of these types is a predefined type of the EAST language.

A one-dimensional array has a distinct component for each possible index value. A multi-dimensional array has a distinct component for each possible sequence of index values that can be formed by selecting one value for each index position within the list of indices (in the given order).

The possible values for a given index are all the values between the lower and upper bounds, inclusive; this range of values is called the index range. Figure 3-18 illustrates the syntax of an array type specification.

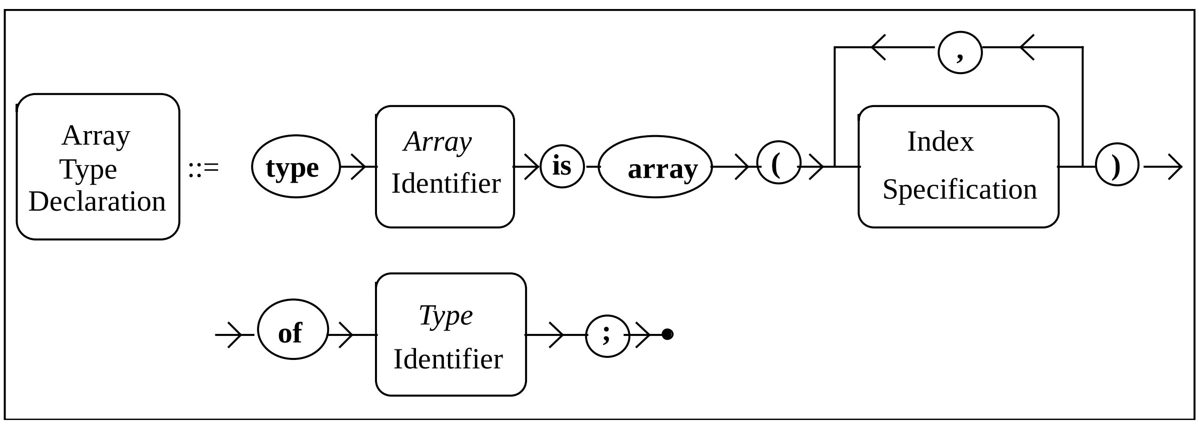


Figure 3-18: Array Type Specification Diagram

An array type can be constrained (i.e., have a fixed number of elements) or unconstrained (i.e., have an undetermined number of elements), depending on the specification of the indices. In multi-dimensional array types, the indices are either all determined or all undetermined.

An index is specified as follows in figure 3-19:

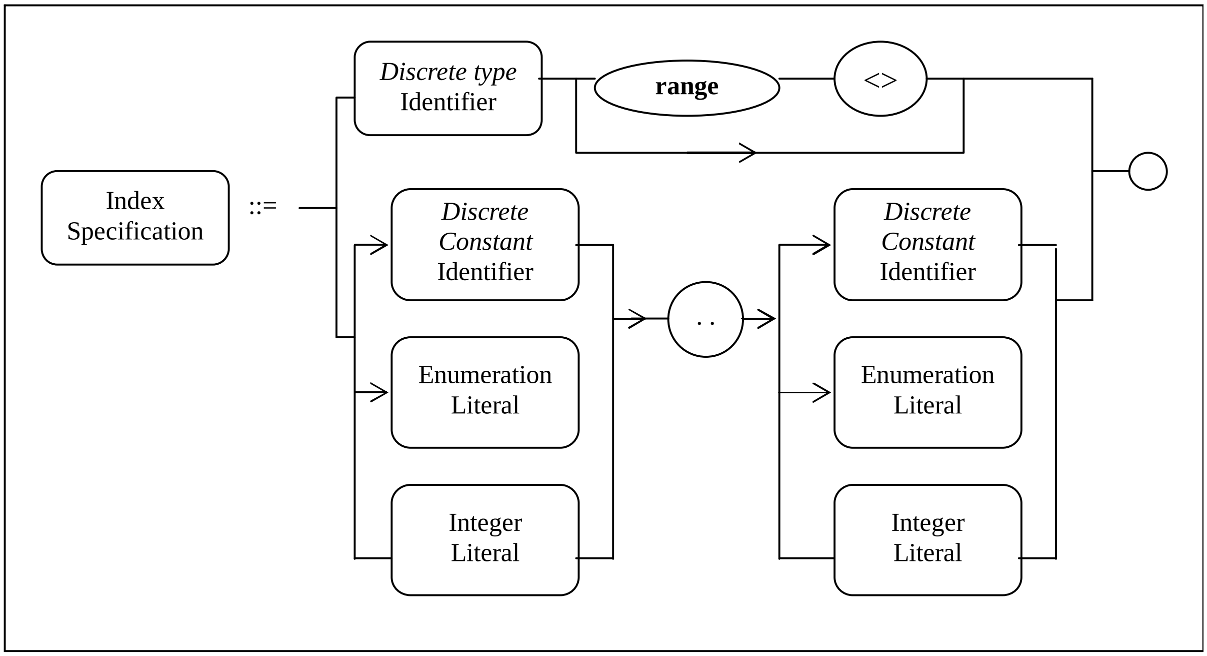


Figure 3-19: Index Specification Diagram

In the '..' notation, the first identifier or literal specifies the lower bound, while the second one specifies the upper bound.

The 'range <>' expression denotes an undetermined number of elements.

The following example defines array types, for which the number of elements is known: 100 characters in a line, and 7 states in a schedule.

type LINE **is array**(1 .. 100) **of** CHARACTER;
-- *CHARACTER is an EAST predefined type*
type SCHEDULE **is array**(DAY) **of** STATE;
-- *DAY is an enumeration type defined in 3.2.1.2 as:*
-- *type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);*

Example 3-6: Constrained Array Type Definitions

The following example defines array types, for which the number of elements is not known: because of the definition of the integer type NUMBER, VECTOR may contain at a maximum MAX reals, and at a minimum 0 real.

```
type VECTOR is array(NUMBER range <>) of REAL;
type MATRIX is array(NUMBER range <>, NUMBER range <>) of REAL;
-- NUMBER is an integer type defined in 3.2.1.3 as:
-- type NUMBER is range 0 .. MAX;
-- REAL is a real type defined in 3.2.1.4 as:
-- type REAL is digits 15;
```

Example 3-7: Unconstrained Array Type Definitions

The actual number of elements must be specified every time an unconstrained array type is used, while the number of elements must not be specified when a constrained array type is used (because this number is already fixed by the type definition).

As an example, MATRIX(1 .. 512, 1 .. 512) designates a matrix which contains 512*512 elements.

If the lower bound of an index range is greater than the upper bound (i.e., if the index range is zero), then the corresponding array row/column has no component.

NOTE – Ways of storing arrays and, therefore, which array index varies first are discussed in 3.3.1.

3.2.1.6 Record Type

A record type is a composite type consisting of a sequence of named components. EAST forbids identical component names in a record. This sequence contains the declaration of each component of the record type. Each declaration indicates the type of the component. Each component type must have been previously defined.

The identifiers of all components of a record type must be distinct. Figure 3-20 illustrates the syntax of a record type specification:

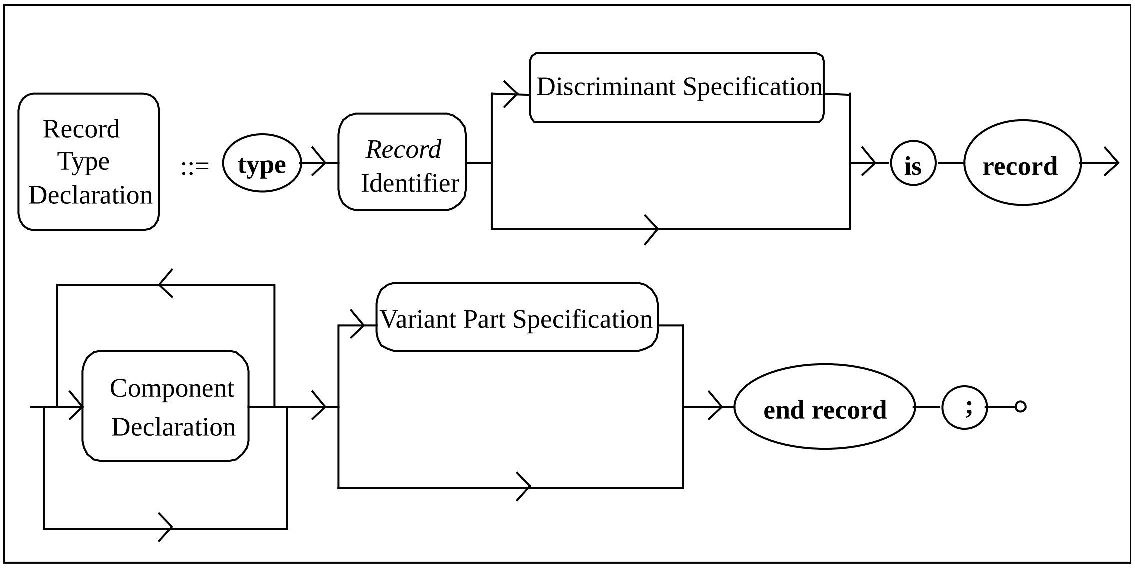


Figure 3-20: Record Type Specification Diagram

where a component declaration is specified as in figure 3-21:

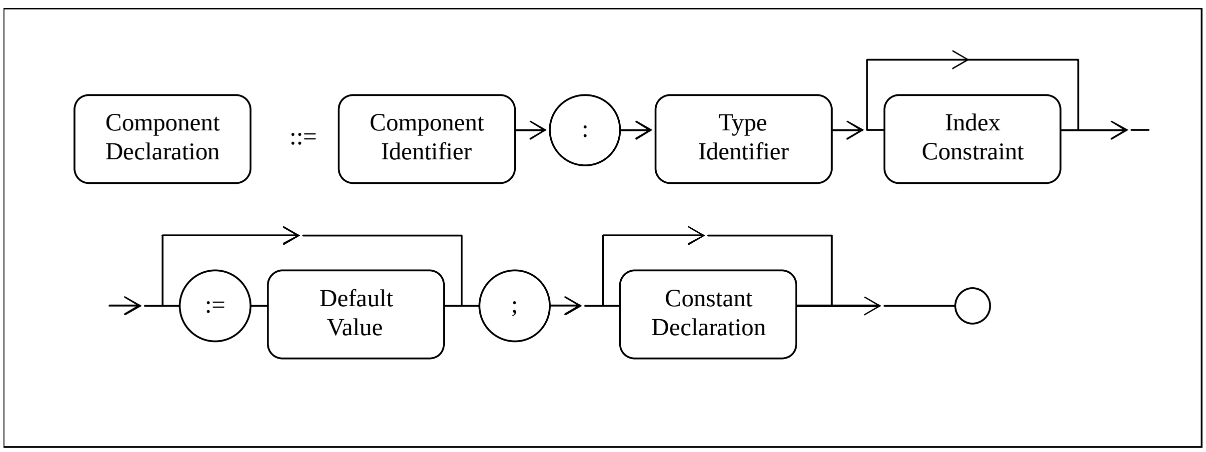


Figure 3-21: Component Declaration Diagram

The optional default value is the one to be given automatically if no other value is given by an application that could generate such data; it is to be used by a generic software layer.

The constant declaration refers to the Marker in 3.2.3.2.2.

Figure 3-22 illustrates default value definitions.

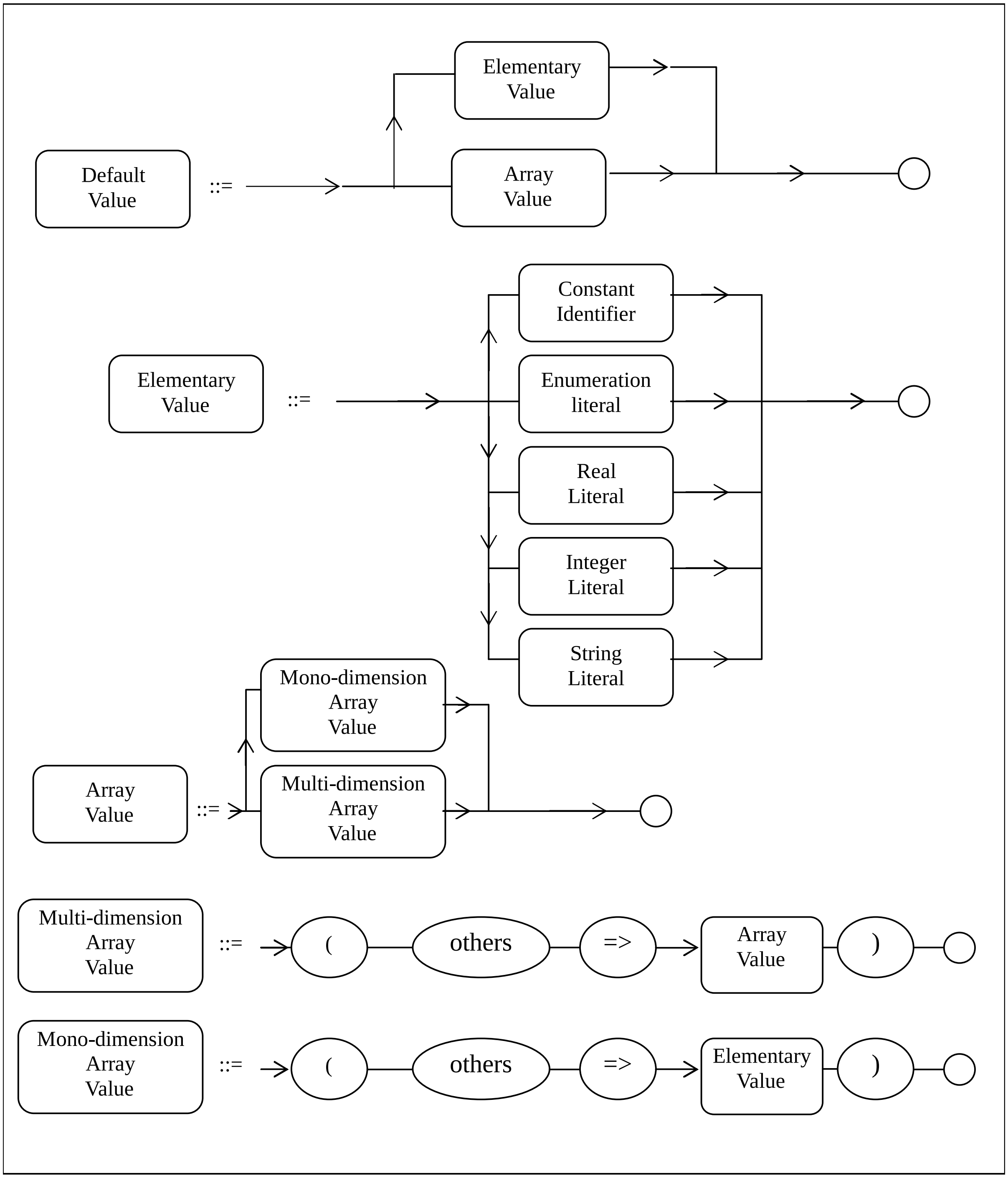


Figure 3-22: Default Value Definition Diagram

When a constant declaration is present, it means that the component is repeated in the data an unknown number of times until the marker it represents (as defined in 3.2.3.2.2.2) is encountered.

Constant declaration in a record definition makes the record size unknown with the consequences explained in 3.2.4.3 (rules 1 and 2).

An index constraint shall be present for an array component if the array type identifier corresponds to an unconstrained array type. In this case, the constraint is specified as in figure 3-23:

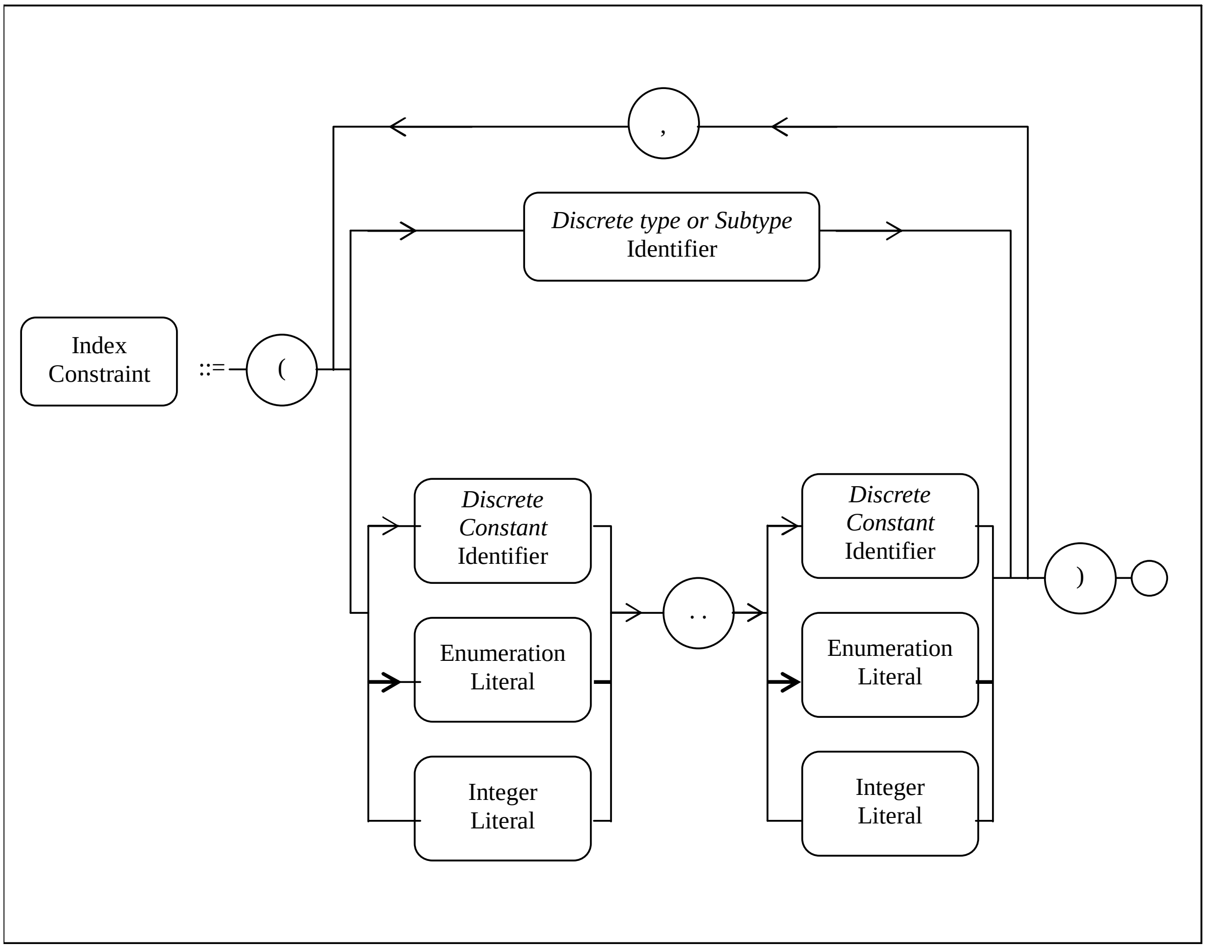


Figure 3-23: Index Constraint Diagram

The following example presents two record type definitions that consist only of simple component declarations:

```
type COMPLEX is record
        REAL_PART:  REAL;
        IMAGINARY_PART:  REAL;
end record;
-- REAL is a real type defined in 3.2.1.4 as:
-- type REAL is digits 15;
type MEASUREMENT_BLOCK is record
        TODAY:  DAY := MON;
        TEMPERATURE:  SMALL_INTEGER := 0;
        VOLUME:  SMALL_INTEGER := 0;
        FIRST_SEQUENCE_OF_MEASUREMENTS:  VECTOR(1 .. 100) := (others => 1);
        SECOND_SEQUENCE_OF_MEASUREMENTS:  VECTOR(1 ..10) := (others => 1);
end record;
-- DAY is an enumeration type defined in 3.2.1.2 as:
-- type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
-- SMALL_INTEGER is an integer type defined in 3.2.1.3 as:
-- type SMALL_INTEGER is range -10 .. 10;
-- VECTOR is an array type defined in 3.2.1.5 as:
-- type VECTOR is array (NUMBER range <>) of REAL;
```

Example 3-8: Record Type Definitions

Some records may contain components of which the size or even the existence depends on the value of another component, called a *discriminant*. The type of a discriminant must be discrete. Figure 3-24 illustrates the syntax of a discriminant specification.

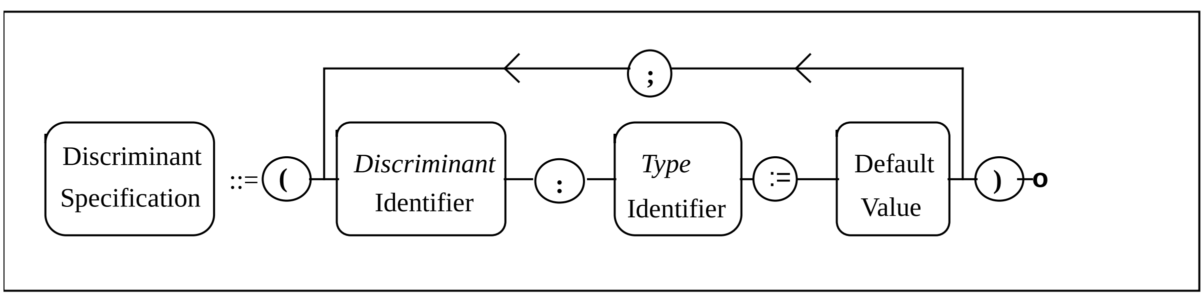


Figure 3-24: Discriminant Specification Diagram

Figure 3-25 illustrates the syntax of a variant part, introduced by the presence of a discriminant.

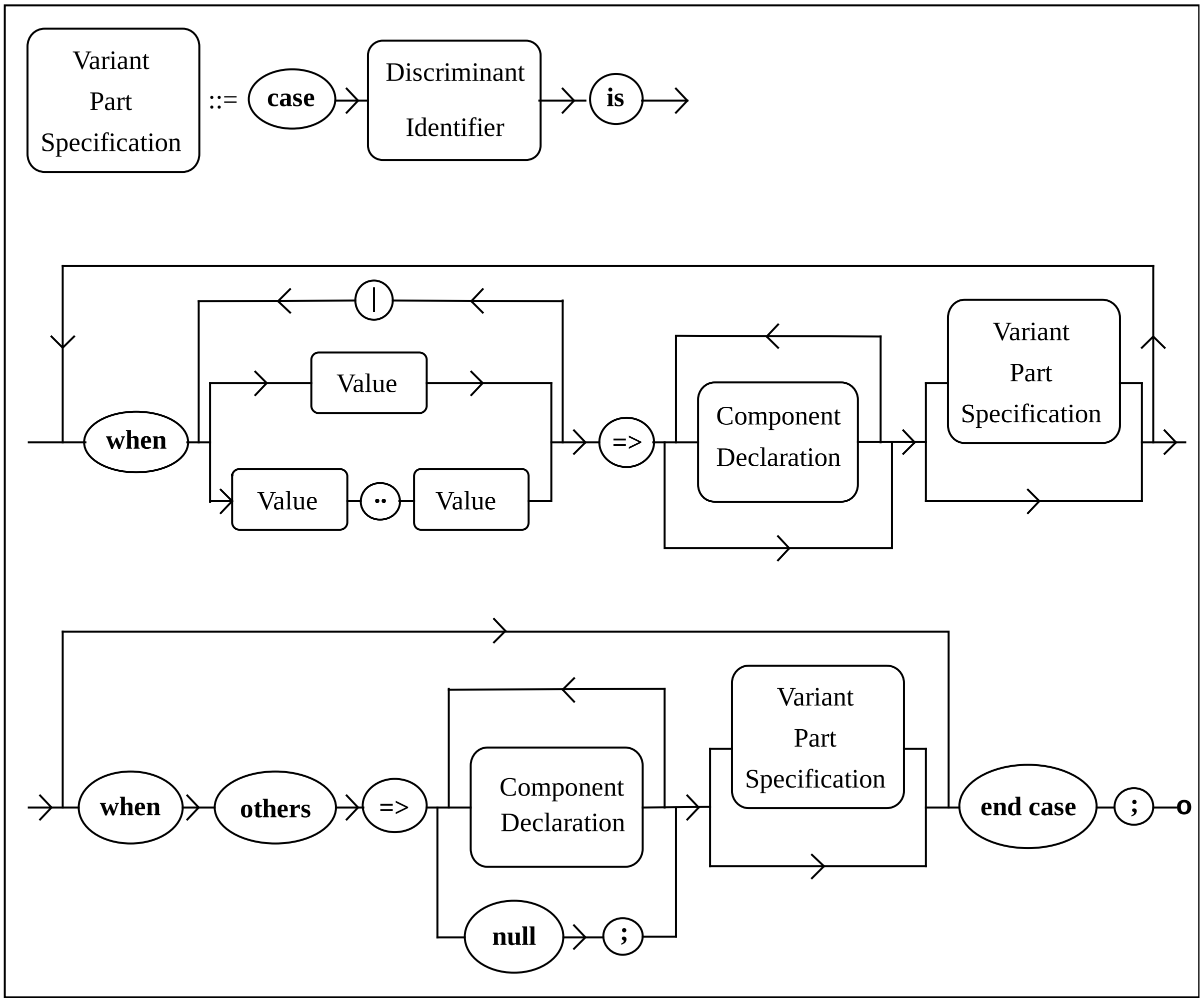


Figure 3-25: Variant Part Specification Diagram

The ‘when others’ clause is mandatory only if all the possible values of the discriminant are not explicitly named before, in the variant part specification.

The following example presents a discriminant that conditions the existence of other components:

```
type ACTIVITY(TODAY:  DAY := MON) is record
        case TODAY is
                when SAT | SUN =>
                        SLEEPING:  DURATION_IN_HOURS;
                        PLAYING_TENNIS:  DURATION_IN_HOURS;
                        SWIMMING:  DURATION_IN_HOURS;
                when MON =>
                        RESTING_AFTER_WEEK_END:  DURATION_IN_HOURS;
                when others =>
                        WORKING:  DURATION_IN_HOURS;
        end case;
end record;
-- DAY is an enumeration type defined in 3.2.1.2 as:
-- type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
-- DURATION_IN_HOURS is an integer type defined as:
-- type DURATION_IN_HOURS is range 0 .. 24;
```

Example 3-9: Record Type Definition with Discriminant

In this example, TODAY is a discriminant for the type ACTIVITY: other components of the record might change depending on the value of TODAY.

The keyword *case* introduces the variant part, which consists of alternative lists of components. The keyword *when*, followed by one or more values (separated by a vertical bar) of the type of the discriminant of the variant part, introduces a list of components that are present for the specified value(s) of the discriminant. The keyword *others* represents all the possible values of the type of the discriminant that have not been taken into account explicitly before (in this example, others is equivalent to TUE | WED | THU | FRI).

The following example presents a discriminant that conditions a size:

```
type SQUARE(LENGTH:  NUMBER := 10) is record
        MAT:  MATRIX(1 .. LENGTH, 1 .. LENGTH);
end record;
-- NUMBER is an integer type defined in 3.2.1.3 as:
-- type NUMBER is range 0 .. MAX;
-- MATRIX is an array type defined in 3.2.1.5 as:
-- type MATRIX is array (NUMBER range <> , NUMBER range <>) of REAL;
```

Example 3-10: Record Type Definition with Discriminant

In the previous example, LENGTH is a discriminant for the type SQUARE: the value of LENGTH determines the size of the matrix. If LENGTH is less than 1 (i.e., LENGTH is equal to 0), then the matrix has no element. If LENGTH is, for example, equal to 5, then the matrix has 25 elements.

The EAST syntax requires a default value for each discriminant (if any) in a record type declaration. A default value does not preclude any possible value for the discriminant of corresponding record objects. In the case of the type ‘SQUARE’, the default value could have been any allowed value for the integer type ‘NUMBER’, i.e., in the range 0 .. MAX.

Some records may contain components of which the size or the existence depend on the value of a data item that is not part of the record: this data item is considered to be a discriminant for the record, except that the occurrence of this discriminant is not in the record itself. Such a discriminant is called a *virtual discriminant*.

The syntax of a virtual discriminant is the same as a ‘classic’ discriminant (see figure 3-24). The only difference is that the discriminant identifier begins in this case with ‘VIRTUAL_’ and does not represent any data item occurrence.

Figure 3-26 presents an example of virtual discriminant use. It describes a packet format.

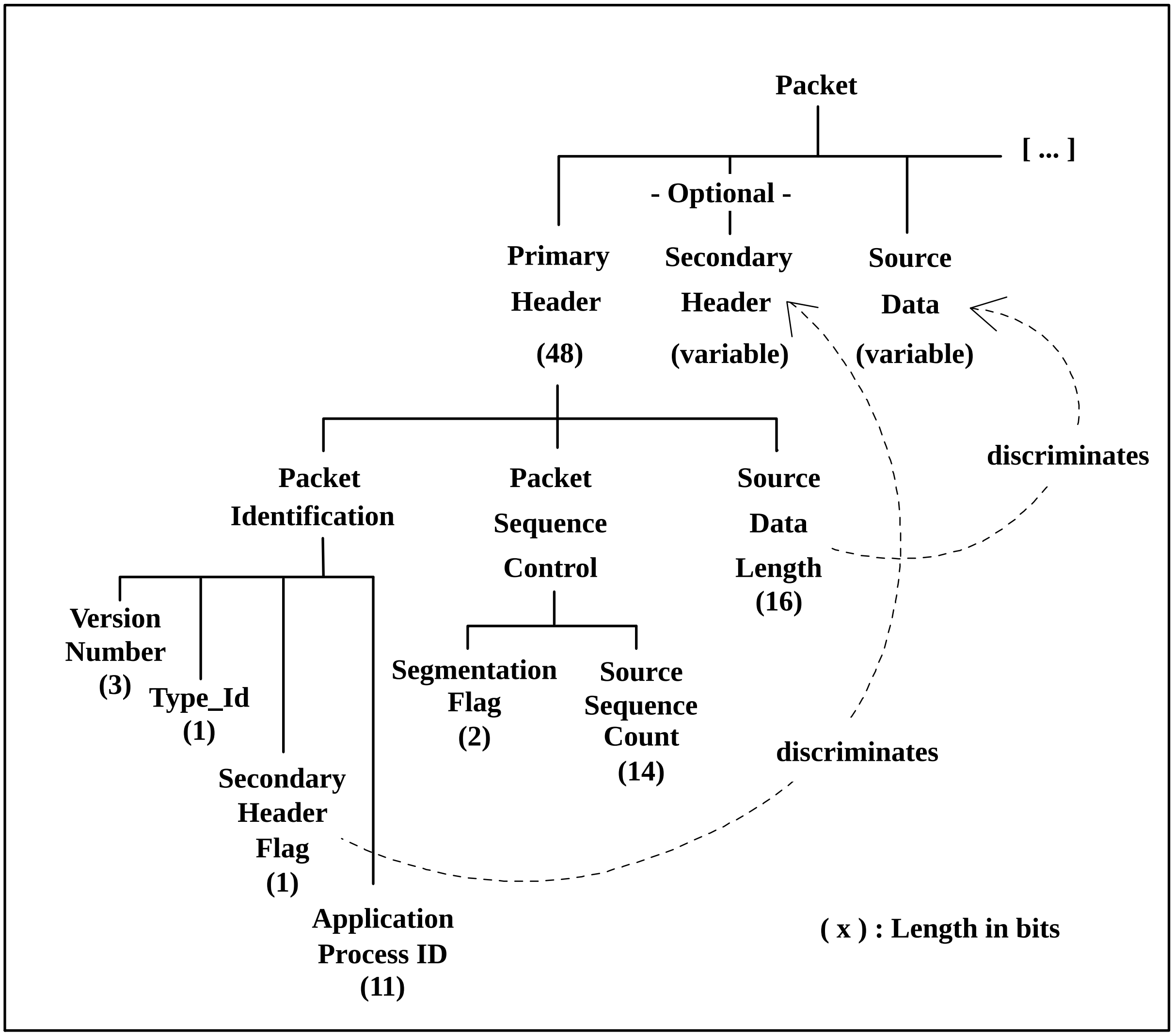


Figure 3-26: Discriminants in a Packet Format

This tree structure can be described using EAST type definitions as follows:

```
-- basic data types used in the first branch
type VERSION is (VERSION_1, VERSION_2);

type PACKET_TYPE is (TELEMETRY , TELECOMMAND);

type PRESENCE_FLAG is (ABSENT , PRESENT);

type PROCESS_IDENTIFICATION is (WORKING , IDLE);

-- structuring type for the Packet Identification
type PACKET_IDENTIFICATION_TYPE is record
        VERSION_NUMBER:  VERSION;
        TYPE_ID:  PACKET_TYPE;
        SECONDARY_HEADER_FLAG:  PRESENCE_FLAG;
        APPLICATION_PROCESS_ID:  PROCESS_IDENTIFICATION;
end record;

-- basic data types used in the second branch
type STATUS is (CONTINUATION_SEGMENT,
                  FIRST_SEGMENT, LAST_SEGMENT, UNSEGMENTED_PACKET);

type COUNTER is range 0 .. 16383;

-- structuring type for the Packet Sequence Control
type PACKET_SEQUENCE_CONTROL_TYPE is record
        SEGMENTATION_FLAG:  STATUS;
        SOURCE_SEQUENCE_COUNT:  COUNTER;
end record;

-- basic data types used in the other branches
type NUMBER is range 0 .. 65535;

type OCTET is range 0 .. 255;

                                        .../...
```

```
.../...

-- structuring types
type DATA_ARRAY is array (NUMBER range <>) of OCTET;
type SECONDARY_HEADER_TYPE is array (1 .. 4) of OCTET;

type PRIMARY_HEADER_TYPE is record
  PACKET_IDENTIFICATION:  PACKET_IDENTIFICATION_TYPE;
  PACKET_SEQUENCE_CONTROL:  PACKET_SEQUENCE_CONTROL_TYPE;
  SOURCE_DATA_LENGTH:  NUMBER;
end record;

type PACKET_FORMAT_TYPE(
       VIRTUAL_SECONDARY_HEADER_FLAG:  PRESENCE_FLAG := PRESENT;
       -- point to the secondary header flag located in the first branch
       VIRTUAL_SOURCE_DATA_LENGTH:  NUMBER := 256)
       -- point to the source data length located in the third branch
is record
       PRIMARY_HEADER:  PRIMARY_HEADER_TYPE;
       case VIRTUAL_SECONDARY_HEADER_FLAG is
         when ABSENT =>
SOURCE_DATA_0:  DATA_ARRAY (1 .. VIRTUAL_SOURCE_DATA_LENGTH);
         when PRESENT =>
                SECONDARY_HEADER:  SECONDARY_HEADER_TYPE;
SOURCE_DATA_1:  DATA_ARRAY (1 .. VIRTUAL_SOURCE_DATA_LENGTH);
       end case;
end record;

FLAG : PRESENCE_FLAG;
LENGTH : NUMBER;
PACKET : PACKET_FORMAT_TYPE;
-- Actual values of discriminants
PACKET.VIRTUAL_SECONDARY_HEADER_FLAG : virtual PRESENCE_FLAG := FLAG;
PACKET.VIRTUAL_SOURCE_DATA_LENGTH : virtual NUMBER := LENGTH;
```

Example 3-11: Logical Description of the Packet Format

The two virtual discriminants ‘VIRTUAL_SECONDARY_HEADER_FLAG’ and ‘VIRTUAL_SOURCE_DATA_LENGTH’ do not really exist in the exchanged data block. They serve as a link between other data:

– VIRTUAL_SECONDARY_HEADER_FLAG is supposed to have the value of the SECONDARY_HEADER_FLAG field in the PACKET IDENTIFICATION block; it conditions the existence of the SECONDARY_HEADER block. It serves as a link between these two fields.

– VIRTUAL_SOURCE_DATA_LENGTH is supposed to have the value of the SOURCE_DATA_LENGTH field in the PRIMARY HEADER; it conditions the size of the SOURCE DATA block. It also serves as a link.

If the size of an array is deduced from several discriminants by a calculation its virtual size declaration remains unchanged (as shown on example 3-10). The calculation to be done is described later after the object declaration section (see 3.2.3) as shown in example 3-12.

```
type A_JULIAN_DAY is range 1 .. (2**32)-1;
type A_SECOND_IN_A_DAY is range 0 .. 86399;

type A_JULIAN_DATE is record
   DAY : A_JULIAN_DAY;
   SECOND : A_SECOND_IN_A_DAY;
end record;

type A_TEMPERATURE is digit 6 range 0.0 .. 100.0;

type TEMPERATURES is array (A_JULIAN_DAY range <> ) of A_TEMPERATURE;

type DATA_RECORD (VIRTUAL_SIZE : A_JULIAN_DAY := 1) is record
      MEASUREMENTS : TEMPERATURES (1 .. VIRTUAL_SIZE);
end record;

FIRST_DATE : A_JULIAN_DATE;
LAST_DATE : A_JULIAN_DATE;
DATA : DATA_RECORD;
-- Actual values of discriminant
DATA.VIRTUAL_SIZE : virtual DAY_TYPE := LAST_DATE.DAY -
FIRST_DATE.DAY;
```

Example 3-12: Calculated Size Array

Supported operators are ‘+’, ‘-’, ‘*’, ‘/’, ‘**’ (exponent), ‘is_odd’, ‘is_even’, ‘cos’, ‘sin’, ‘tan’, ‘acos’, ‘asin’, ‘atan’, ‘log’, ‘ln’, ‘cosh’, ‘sinh’, ‘tanh’, ‘acosh’, ‘asinh’, ‘atanh’, ‘(’, ‘)’, ‘!’ (factorial).

The syntax of the virtual declaration for a calculated condition is the same as in example 3-9.

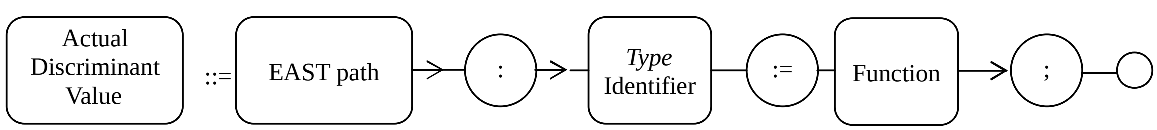
The calculation to be done is described later after the object declaration section, as shown in example 3-13.

Operators that parallel generic function calls in Ada may be used in an EAST description. These are supported by the software application.

```
type A_RESULT is range 0 .. 100;

type RESULTS (VIRTUAL_BONUS_FLAG : BOOLEAN := TRUE) is record
   RESULT_1 : A_RESULT;
   RESULT_2 : A_RESULT;
   case VIRTUAL_BONUS_FLAG is
       when TRUE => BONUS : A_RESULT;
    end case;
end record;

PREVIOUS_WEEK : A_RESULT;
THIS_WEEK : RESULTS;

-- Actual values of discriminant

THIS_WEEK.VIRTUAL_BONUS_FLAG : virtual BOOLEAN
                      := (THIS_WEEK.RESULT_2 - THIS_WEEK.RESULT_1) > PREVIOUS_WEEK;
```

Example 3-13: Calculated Component Presence Condition

Figure 3-27 illustrates the syntax of an actual discriminant value declaration.



NOTES

1 Function returns a value compliant with type identifier computed using predefined operators applied to values designated by their complete EAST path.

2 The EAST path is built using the name of each hierarchy level from the top to the designated item.

Figure 3-27: Actual Discriminant Value Declaration Diagram

3.2.1.7 Type Summary

The following diagram (figure 3-28) presents the types that can be found in the logical part of an EAST description:

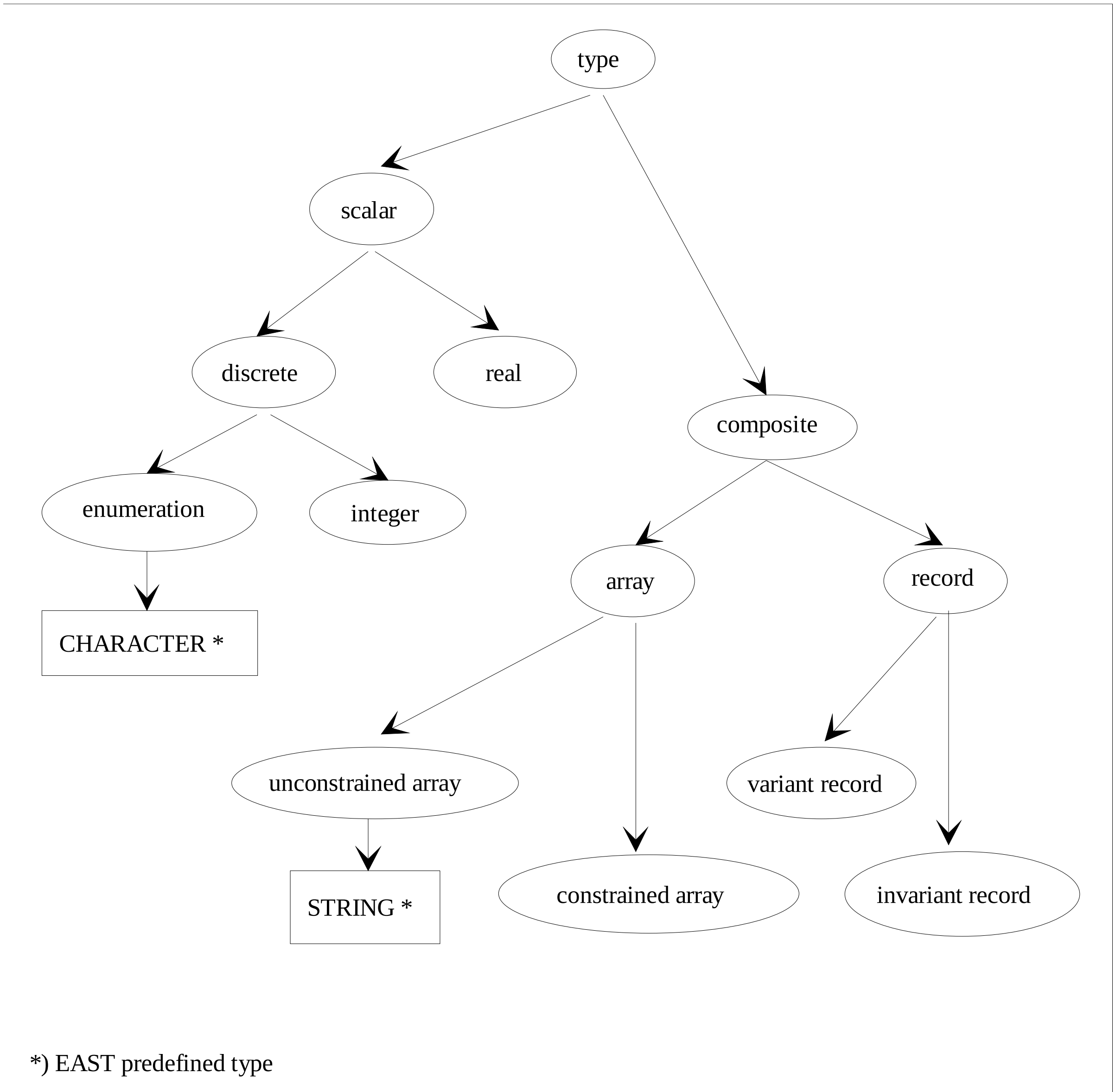


Figure 3-28: Type Summary

Scalar types have a binary coding or an ASCII coding, according to their physical description (see 3.3.3).

A variant record is a record that contains at least one discriminant. An invariant record contains no discriminant.

3.2.2 SUBTYPE DECLARATIONS

A subtype of a given type is used to restrict the set of values of the initial type. The initial type must be known at the subtype declaration time: either it is a predefined type of the EAST language or it has been previously declared.

Figure 3-29 illustrates the syntax of a subtype declaration.

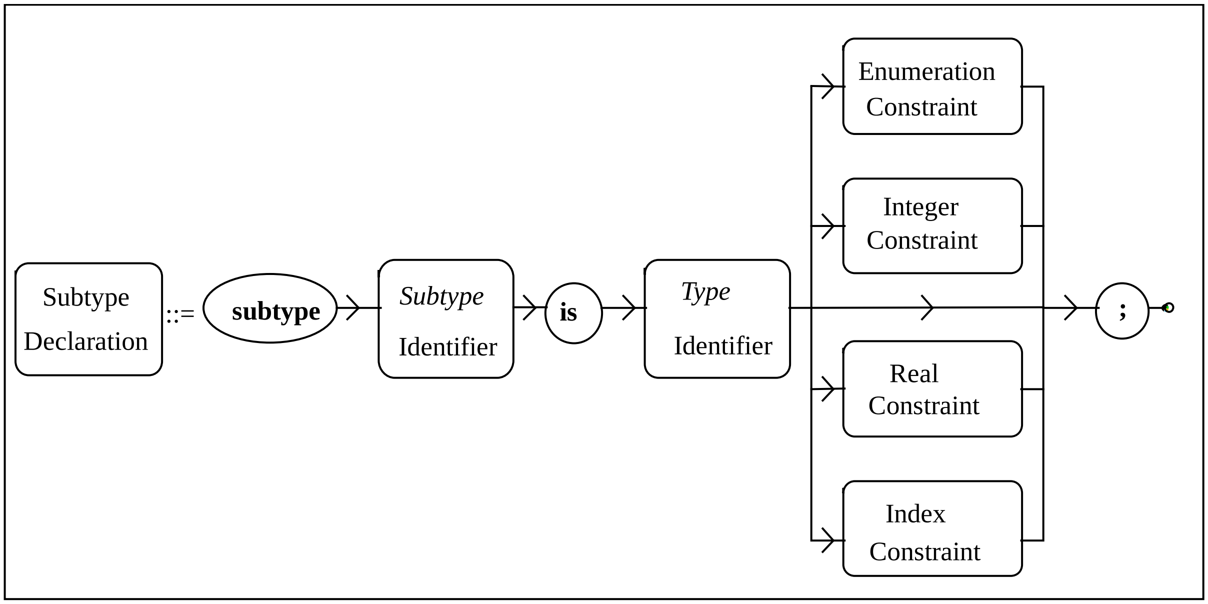


Figure 3-29: Subtype Declaration Diagram

The constraint for an enumeration subtype is defined in figure 3-30.

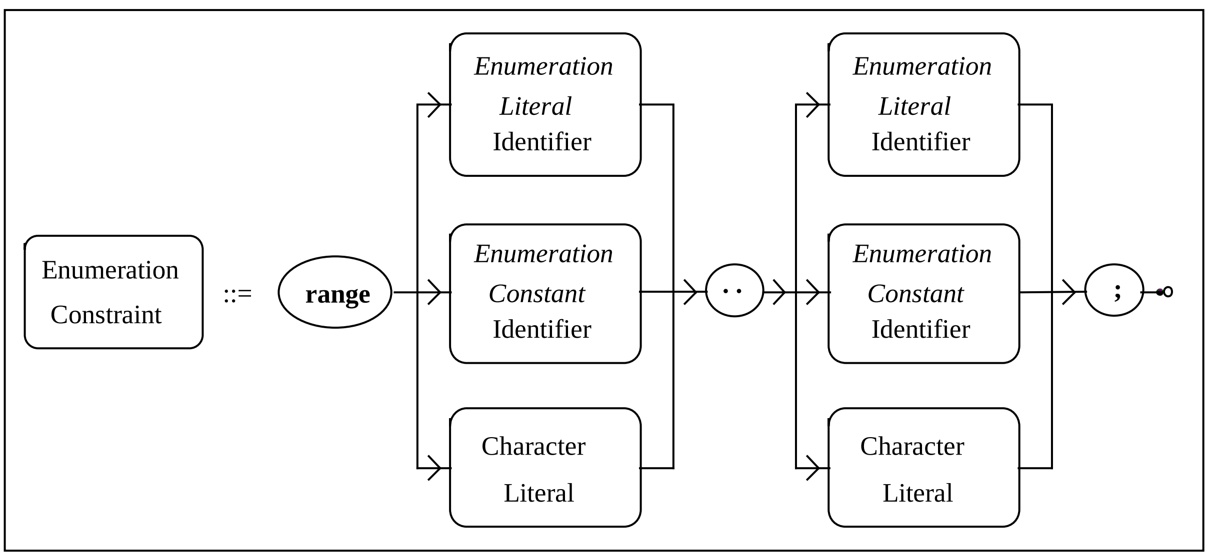


Figure 3-30: Enumeration Constraint Diagram

If a character literal used as range bound is not a printable character (as defined in annex B), its constant identifier is used (constants of the type CHARACTER are defined in annex B in a table called ASCII).

The constant identifier for a character must be prefixed by 'ASCII.', in order to avoid any confusion with other identifiers defined in the current description.

The following example defines some subtypes of CHARACTER:

subtype CAPITAL_LETTER **is** CHARACTER **range** 'A' .. 'Z';
-- the range bounds are printable

subtype LINE_FORMAT **is** CHARACTER **range** ASCII.HT .. ASCII.CR;
-- the range bounds are not printable

Example 3-14: Character Declarations

The constants of the type CHARACTER, which are specified in the ASCII table, are EAST predefined constants.

The constraint for an integer subtype is defined in figure 3-31.

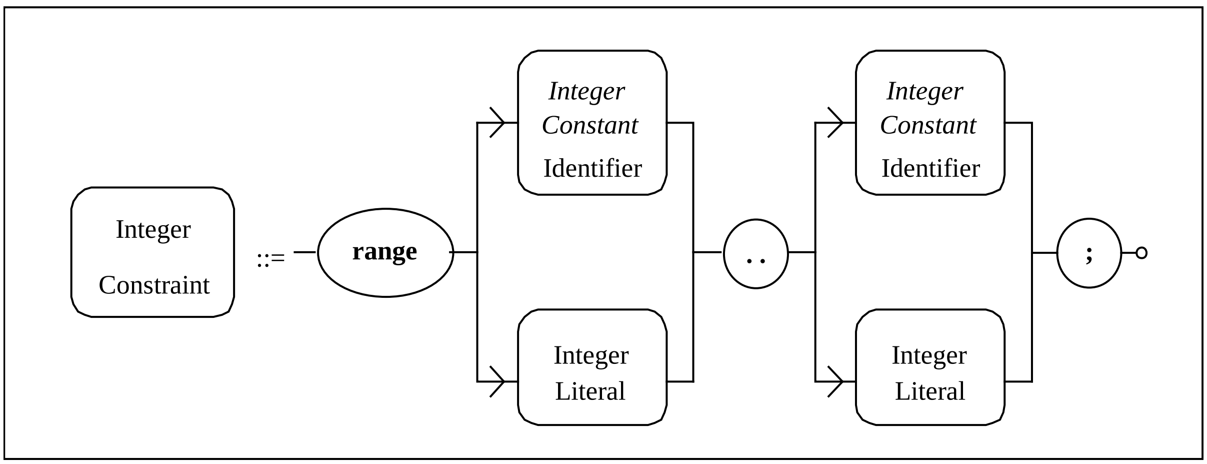


Figure 3-31: Integer Constraint Diagram

In the previous diagram, the first integer gives the lower bound and the second the upper bound of the specified range.

The constraint for a real subtype is defined in figure 3-32.

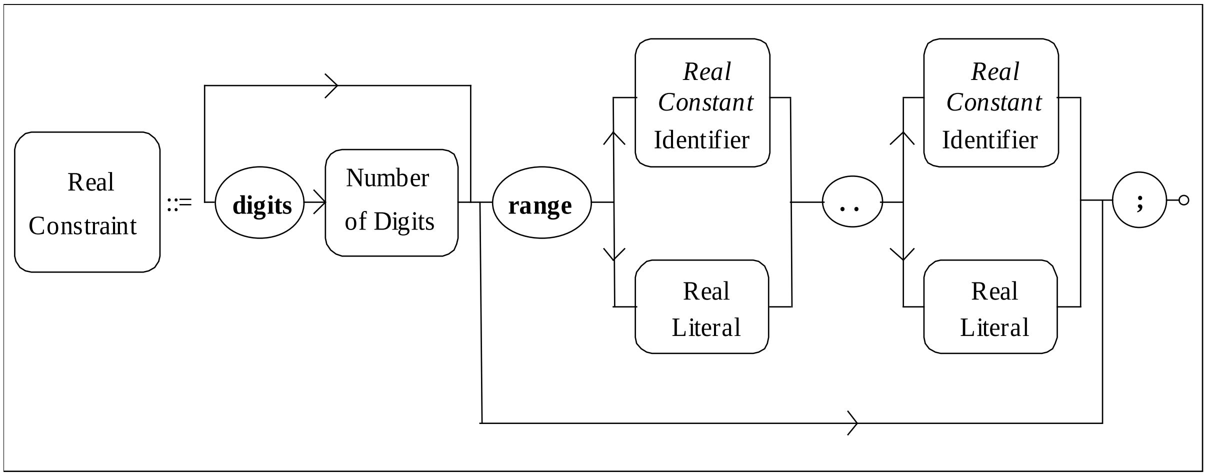


Figure 3-32: Real Constraint Diagram

In the previous diagram, the first real gives the lower bound and the second the upper bound of the specified range.

The constraint for an array subtype or for a subtype of the predefined type STRING is defined in figure 3-23 (on page 3-17). In this diagram, the discrete literal in the range specification is any integer (based or decimal integer) literal or any enumeration literal. In the same way, the discrete constant identifier in the range specification is any integer or enumeration constant (see 3.2.3.2).

The following example defines some subtypes:

subtype WEEK_END **is** DAY **range** SAT .. SUN **;**
-- *where DAY is an enumeration type defined in 3.2.1.2 as:*
-- *type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);*
subtype VERY_SMALL_INTEGER **is** SMALL_INTEGER **range** -5 .. 5**;**
-- *where SMALL_INTEGER is an integer type defined in 3.2.1.3 as:*
-- *type SMALL_INTEGER is range -10 .. 10;*
subtype MY_REAL **is** REAL **range** -9_999.999 .. 9_999.999;
-- *where REAL is a real type defined in 3.2.1.4 as:*
-- *type REAL is digits 15;*
subtype SMALL_MATRIX **is** MATRIX **(**1 **..** 10 , 1 **..** 10**);**
-- *where MATRIX is an array type defined in 3.2.1.5 as:*
-- *type MATRIX is array (NUMBER range <>, NUMBER range <>) of REAL;*
subtype NAME **is** STRING **(**1 **..** 32**);**
-- *where STRING is a predefined array type (see 3.2.1.1).*

Example 3-15: Subtype Declarations

3.2.3 OBJECT DECLARATIONS

An object is an entity that contains a value of a given type. A declared object is called a constant if the reserved word **constant** appears in the object declaration. An object that is not a constant is called a variable.

3.2.3.1 Declaration of Variables

The declaration of a variable uses types specified previously in 3.2.1. Variables correspond to the data that are to be exchanged. Figure 3-33 illustrates the syntax for the declaration of a variable.

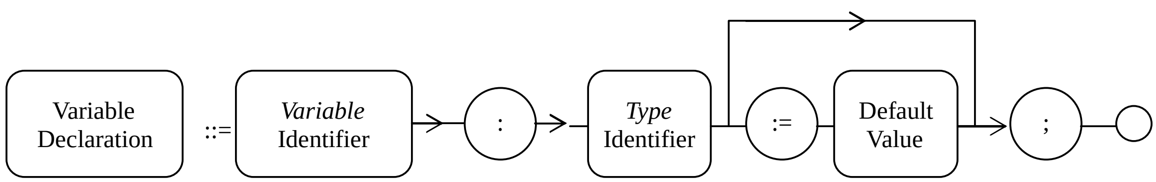


Figure 3-33: Variable Declaration Diagram

The default value (which definition is given by figure 3-22) is the one to be given automatically if no other value is given by an application generating such data; it is to be used by generic software layer.

A variable declaration consists of only one identifier (the variable identifier) followed by the identifier of the type that describes the corresponding data.

```
UPDATED_DATA:  MEASUREMENT_BLOCK ;
   -- MEASUREMENT_BLOCK is a record type defined in 3.2.1.6
INSTRUMENT_STATUS : STATE := ON;
   -- STATE is an enumeration type defined in 3.2.1.2:
   -- ON is a default value
```

Example 3-16: Variable Declaration

3.2.3.2 Declaration of Constants

The declaration of a constant must include an explicit initialization, except for the EOF Marker declaration (see 3.2.3.2.2). This declaration guarantees that the corresponding object value cannot be modified after initialization. Figure 3-34 illustrates the syntax of a constant declaration.

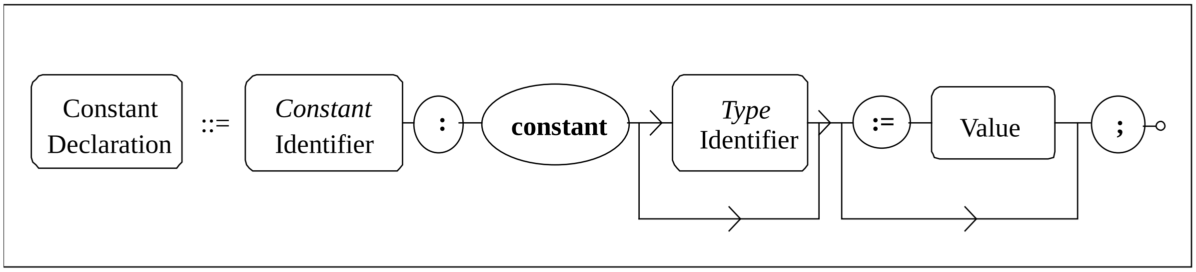


Figure 3-34: Constant Declaration Diagram

A constant declaration consists of only one identifier (the constant identifier) followed by the reserved word **constant**, an optional identifier for the constant type, and the value of the constant.

FIRST_DAY_OF_THE_WEEK: **constant** DAY **:=** MON**;**

Example 3-17: Constant Declaration

The value of a constant can be specified as a static expression, combining other constant values and operators (‘+’, ‘*’, ‘**’, ‘-’, ‘/’, ‘(’ and ‘)’).

‘+’ and ‘-’ are unary or binary operators (addition and subtraction). ‘*’, ‘/’ and ‘**’ are binary operators: ‘*’ is the multiplication operator, ‘/’ is the division operator, ‘**’ is the exponentiation operator. ‘(’ and ‘)’ are used to specify an explicit precedence for the expression evaluation.

Constants may be declared either in the section for the declaration of types or in the section for the declaration of variables (see figure 3-13). In the first case, they contribute to data model definitions while they represent, in the second case, some special data occurrences called markers.

The first definition of a variable within the logical description part delimits the two sections. Any declaration that occurs before the first variable definition belongs to the section for the declaration of types. Any declaration that occurs after the first variable definition (including the first variable declaration itself) belongs to the section for the declaration of variables.

3.2.3.2.1 Constants in the Section for the Declaration of Types

A constant that is declared in the section for the declaration of types can be used:

- in type or subtype declarations for the specification of range bounds,
- in constant declarations for the specification of the values.

In this case, the constant is either an integer constant, a real constant, or an enumeration constant, the end objective of the constant being its use as a range bound.

A number declaration is a special form of a constant declaration, where no type is specified.

```
PI:  constant := 3.14159_26536; -- a real number
MAXIMUM:  constant := 500; -- an integer number
NUMBER_OF_VALUES_OF_AN_OCTET:  constant := 2**8; -- the integer 256
```

Example 3-18: Number Declarations

3.2.3.2.2 Markers

A marker declaration is a special form of a constant declaration, where the type of the constant is mandatory. A marker is used to delimit the end of the repetition of an element. A marker indicates that the data item just above is repeated until the marker value is found.

A marker is a constant which should be unambiguously recognized. The type of a marker is therefore restricted to integer type, enumeration type, character type, or character string type.

The element of a repetition delimited by a marker can only be a variable or a component of a record type.

3.2.3.2.2.1 Markers: Constants in the Section for the Declaration of Variables

When a marker declaration occurs after the declaration of a variable, it means that this variable which is declared immediately before the constant occurs an undetermined number of times, the last instance being followed by the constant value.

RULE – The marker must follow a declaration of a variable. It cannot be the first declaration of the section.

The following example represents a set of values, the number of values being unspecified. The end of the set occurs when the character string 'END' is encountered within the data.

```
VALUE : COEFFICIENT;  -- COEFFICIENT is a real type defined in 3.2.1.4 as:
                      -- type COEFFICIENT is digits 10 range 0.0 .. 1.0;

END_OF_COEFFICIENTS : constant STRING := "END";
```

Example 3-19: Marker Declaration

The presence of the EOF marker implies that the previous element is repeated until the File Management System returns an 'end of file' indication.

The following convention is adopted: the type of the Marker is an EAST predefined type, called EOF. No explicit value is associated with this constant since this value is unknown. This is the only case of a constant declaration where the value is absent.

RULE – The EOF marker can only be used once in an EAST description. When used, the EOF marker will be the last declaration in the logical description part.

The next example presents the description of a data file that contains a header and *n* values (*n* being undetermined).

```
HEADER : HEADER_TYPE; -- any record type

VALUE : COEFFICIENT;  -- COEFFICIENT is a real type defined in 3.2.1.4 as:
                      -- type COEFFICIENT is digits 10 range 0.0 .. 1.0;

END_OF_COEFFICIENTS : constant EOF ;
```

Example 3-20: EOF Marker Declaration

3.2.3.2.2.2 Markers: Constants in Record Type Definition

When a marker declaration occurs within the declaration of the components of a record type, it means that the component which is declared immediately before the constant occurs an undetermined number of times, the last instance being followed by the constant value.

RULE – The marker must follow a declaration of a component. It cannot be the first declaration in the record.

The following example represents such a usage of a marker.

```
type CLIENT_ADDRESS is record
        ONE_CHARACTER : CHARACTER;
        END_OF_ADDRESS : constant CHARACTER := ASCII.CR; -- carriage return
end record;

type CLIENT is record
        NAME : STRING (1 .. 30);
        COMPANY : STRING (1..30);
        ADDRESS : CLIENT_ADDRESS;
        END_OF_ADDRESSES : constant STRING := "-- End of addresses --";
end record;
```

Example 3-21: Marker Declaration in Record Definition

3.2.4 REPRESENTATION CLAUSES

Concerning the descriptive features, the representation clauses are one of the most significant facilities offered by EAST. The representation clauses specify the mapping between the logical types of the language and their physical representation. EAST provides the length clauses, the enumeration representation clauses, and the record representation clauses.

A representation clause immediately follows the type whose storage it describes. A representation clause is **mandatory** in a logical data description, except for variable-sized components, for which the representation cannot be known.

3.2.4.1 Length Clauses

A length clause specifies the number of bits that data of a particular type occupy in storage. Length clauses must be provided for enumeration, integer, and real types. Length clauses must also be provided for composite types every time it is possible, i.e., every time the size of the composite type (array or record) is known. In such case, this size is the size of the whole type. Figure 3-35 illustrates the syntax of a length clause declaration.

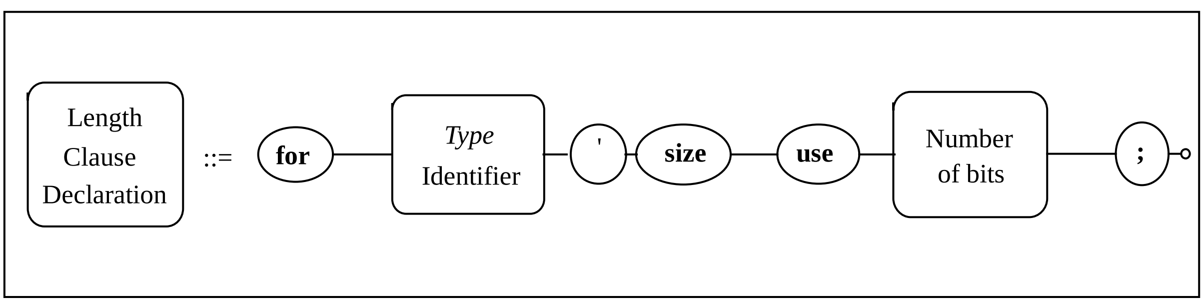


Figure 3-35: Length Clause Specification Diagram

The following example presents type declarations with their associated length clauses:

```
type VALUE is range 0 .. 500;
for VALUE'size use 16; -- bits

type COLUMN is array(1 .. 10) of VALUE;
for COLUMN'size use 160; -- 10 times 16 bits
```

Example 3-22: Length Clause Declarations

If the elements of the described array are not contiguous, the unused space between elements must be described explicitly. This results in contiguous elements containing unused space.

The following example presents an array which contains values and spare fields (for alignment purpose).

```
type VALUE is range 0 .. 500;
for VALUE'size use 16; -- bits

type OCTET is range 0 .. 255;
for OCTET'size use 8;

type SPARE is array (1 .. 2) of OCTET;
for SPARE'size use 16;

type ELEMENT is record
        A_VALUE:  VALUE;
        A_SPARE:  SPARE;
end record;
for ELEMENT'size use 32;

type COLUMN is array(1 .. 10) of ELEMENT;
for COLUMN'size use 320; -- 10 times 32 bits
```

Example 3-23: Explicit Description of Unused Space

3.2.4.2 Enumeration Representation Clauses

An enumeration representation clause specifies the bit pattern for the binary representation of the value associated with each literal of an enumeration type. An enumeration representation clause is optional.

If an enumeration representation clause is provided, each literal of the enumeration type must be provided with a unique bit pattern. The integer values (corresponding to the given bit pattern) specified for the enumeration type must satisfy the predefined ordering relation of the type; i.e., they must increase.

If no enumeration representation clause is provided for a binary enumeration type, default integer codes are presumed: the value of the first listed enumeration literal is zero; the value for each other enumeration literal is one more than for its predecessor in the list.

Figure 3-36 illustrates the syntax of an enumeration representation clause declaration:

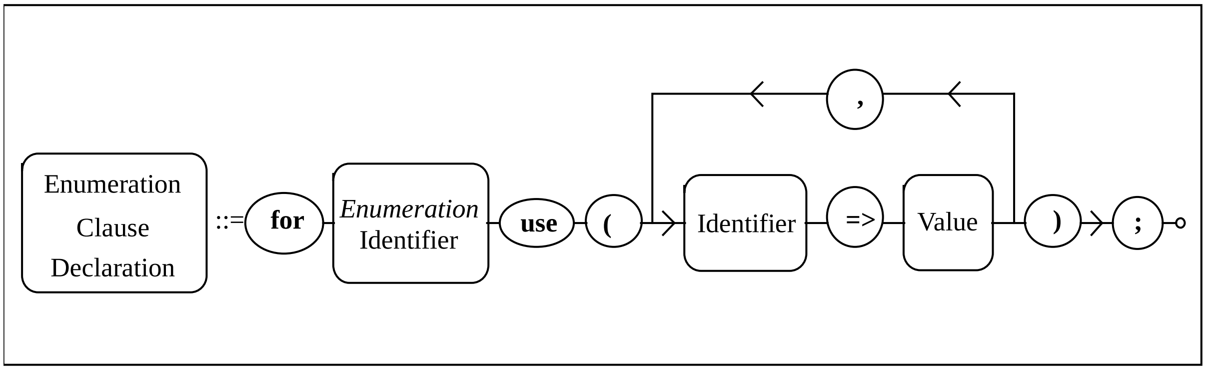


Figure 3-36: Enumeration Clause Specification Diagram

The integer value, specifying the mapping with bit pattern, can be expressed using the binary, octal, decimal or hexadecimal notation. The syntax for a binary, octal, or hexadecimal value is: base # value#.

```
type CODE is (ADD , SUB , MUL , LDA , STA , STZ);
for CODE use (ADD => 2#1#, SUB => 2#10#,
              MUL => 2#11#, LDA => 2#1000#,
              STA => 2#11000#, STZ => 2#11111#);

type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
for DAY use (  MON => 8#1#, TUE => 8#2#, WED => 8#3#,
              THU => 8#4#, FRI => 8#5#, SAT => 8#6#, SUN => 8#7#);

type STATE is (OFF , ON);
for STATE use (OFF => 0 , ON => 1);

type SYNCHRONIZATION is (NOMINAL_SYNCHRO , INVERSE_SYNCHRO);
for SYNCHRONIZATION use ( NOMINAL_SYNCHRO => 16#0C# ,
                          INVERSE_SYNCHRO => 16#F5# );
```

Example 3-24: Enumeration Clause Declarations

3.2.4.3 Record Representation Clauses

A record representation clause specifies the storage representation of records, that is, the order, position, and size of record components (including discriminants, if any).

A record representation clause occurs immediately after the record type definition and before the record length clause (if its size is known).
A component clause specifies the storage position of a component, relative to the beginning of the record. A component clause must be provided every time it is possible, i.e., every time the exact location of the component is known (e.g., it is not possible for variable-sized components).

If component clauses are given for all components, the record representation clause completely specifies the representation of the record type.

If some component clauses are missing, the order of these components is specified as in the record type definition.

The order of component clauses in a record representation clause is not significant.

A representation clause is mandatory for a discriminant, except for virtual discriminants which cannot have a representation clause.

There is an overlap between distinct variants. The EAST syntax requires that a variant part is declared after the fixed part of a record. If the variant part has a constant length, fixed components are allowed to be physically located after the alternative components of the variant: the actual location of the fixed components is specified using a record representation clause.

Figure 3-37 illustrates the syntax of a component representation. The expression after the keyword **at** indicates a relative distance to the start of the structure. This distance is expressed in words, the length of a word being either 16 bits or 32 bits (see page 3-46 for the declaration of the length). If distance is equal to 0, the range is specified relatively to the beginning (i.e., location 0) of the record. The expressions after the keyword **range** are the positions in bits relatively to the distance.

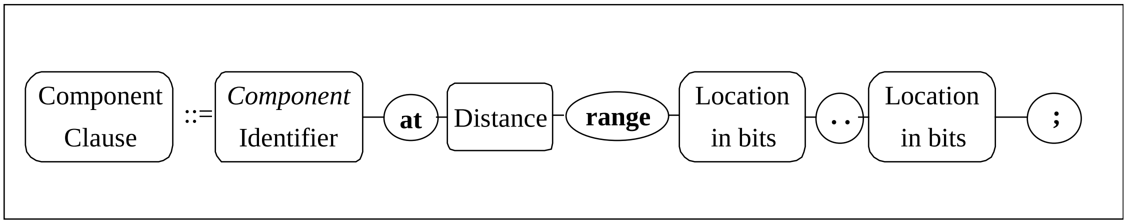


Figure 3-37: Component Representation Clause Specification Diagram

Figure 3-38 illustrates the syntax of a record representation clause.

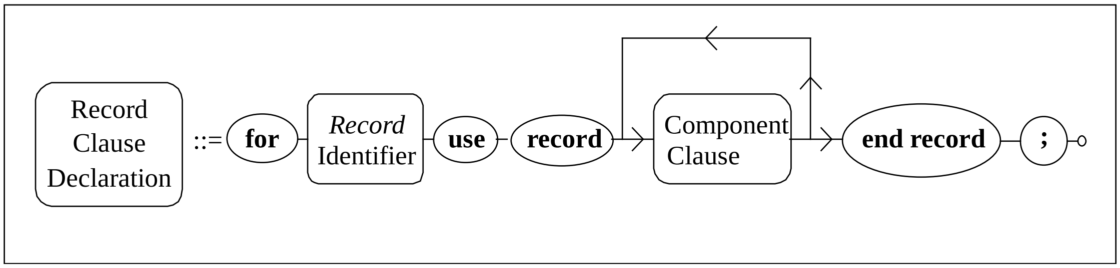


Figure 3-38: Record Representation Clause Specification Diagram

The next four examples illustrate the use of record representation clauses, in different cases:

– First case: everything is known (the size and the location of every component);

– Second case: the number of elements of a component is not known at definition time, and the size and the location of this variable component are therefore not known;

– Third case: the global size of the record is known, but there are two alternatives for the choice of the components;

– Fourth case: the record contains alternatives for the choice of the components, followed by a fixed (i.e., known) component.

Assuming the following definitions of the basic data types used in the four examples:

```
-- enumeration type definition
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
for DAY'size use 8;

-- integer type definitions
type MONTH is range 1 .. 12;
for MONTH'size use 8;
type YEAR is range 1900 .. 2100;
for YEAR'size use 16;
type NUMBER is range 1 .. 10;
for NUMBER'size use 8;
type ALPHA is range 1 .. 10;
for ALPHA'size use 8;
type BETA is range 1 .. 10;
for BETA'size use 8;
type GAMMA is range 1 .. 10;
for GAMMA'size use 8;
type DELTA is range 1 .. 10;
for DELTA'size use 8;

-- real type definition
type VALUE is digits 5;
for VALUE'size use 32;

-- array type definition
type VECTOR is array(NUMBER range <>) of VALUE;
```

Example 3-25: Type Definitions

The following example (figure 3-39) presents the case of a complete record representation clause. The record representation clause is provided because the size and the location of every component of the data structure are known.

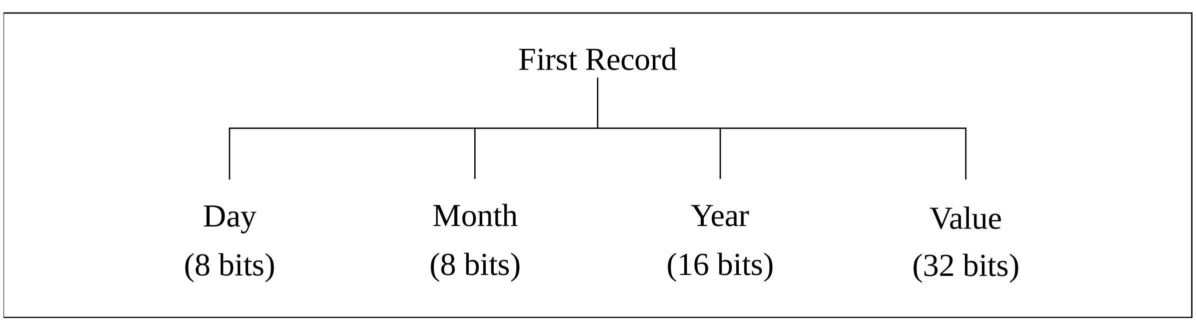


Figure 3-39: First Tree Structure

This tree structure is described using the following declaration:

```
type FIRST_RECORD is record
        THE_DAY_OF_MONTH:  DAY;
        THE_MONTH:  MONTH;
        THE_YEAR:  YEAR;
        THE_MEASUREMENT:  VALUE;
end record;
for FIRST_RECORD use record
        THE_DAY_OF_MONTH at 0 range 0 .. 7;
        THE_MONTH at 0 range 8 .. 15;
        THE_YEAR at 0 range 16 .. 31;
        THE_MEASUREMENT at 0 range 32 .. 63;
end record;
for FIRST_RECORD'size use 64; -- 64 bits
```

Example 3-26: Complete Record Representation Clause Declaration

The following example (figure 3-40) presents the case of an incomplete record representation clause. A fortiori no representation clause could be found after a computed size array or a computed structure record.

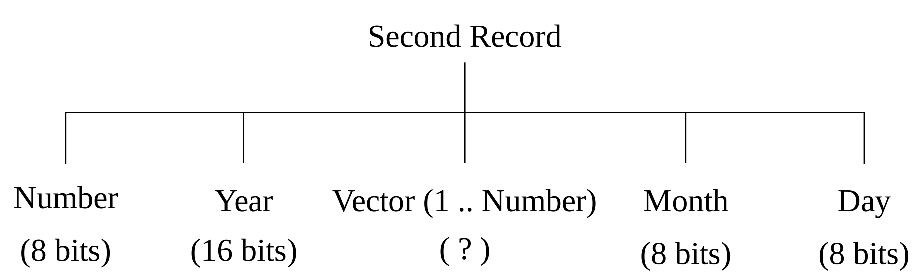


Figure 3-40: Second Tree Structure

The number of measurements is not known at definition time. The size of the vector of measurements is therefore not provided. The tree structure is described using the following declarations:

```
type SECOND_RECORD(THE_NUMBER:  NUMBER := 1) is record
        THE_YEAR:  YEAR;
        THE_MEASUREMENT:  VECTOR(1 .. THE_NUMBER);
        THE_MONTH:  MONTH;
        THE_DAY_OF_MONTH:  DAY;
end record;
for SECOND_RECORD use record
        THE_NUMBER at 0 range 0 .. 7;
        THE_YEAR at 0 range 8 .. 23;
        -- no component clause for THE_MEASUREMENT,
        -- for THE_MONTH nor for THE_DAY_OF_MONTH
end record;
-- no length clause for SECOND_RECORD type
```

Example 3-27: Incomplete Record Representation Clause Declaration

In this example, the length of ‘THE_MEASUREMENT’ depends on the value of the discriminant ‘THE_NUMBER’. No representation clause can be given for it. Nevertheless the size is determined by the expression ‘THE_NUMBER times 32’, 32 being the size of the basic element VALUE. The component ‘THE_MEASUREMENT’ begins at bit 24. The length of ‘THE_MONTH’ is known but its location is not known at definition time. No representation clause can be given for it. The component ‘THE_MONTH’ begins after the end of ‘THE_MEASUREMENT’. In the same way, the length of ‘the_day_of_month’ is known, but its location is not known at definition time. No representation clause can be given for it. The component ‘THE_DAY_OF_MONTH’ begins after the end of ‘THE_MONTH’.

The following example (figure 3-41) gives the case of a complete record representation clause, where some components overlap:

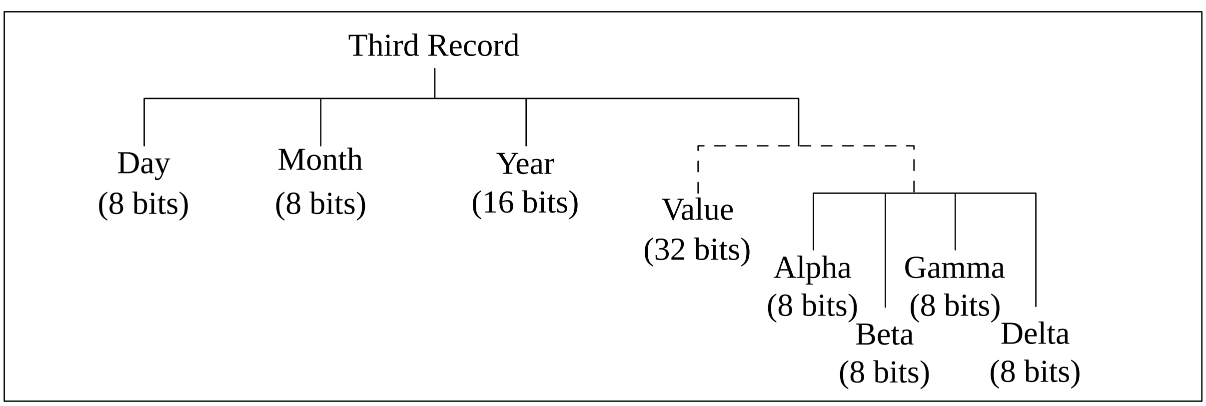


Figure 3-41: Third Tree Structure

The size of the record is known at definition time: all the alternatives have the same length (32 bits if THE_DAY_OF_MONTH is equal MON, and 4*8 bits if THE_DAY_OF_MONTH is equal something else). The location of every component is known.

```
type THIRD_RECORD(THE_DAY_OF_MONTH:  DAY := MON) is record
        THE_MONTH:  MONTH;
        THE_YEAR:  YEAR;
        case THE_DAY_OF_MONTH is
          when MON =>
                  THE_MEASUREMENT:  VALUE; -- 32 bits
          when others =>
                  THE_ALPHA_VALUE:  ALPHA; -- 8 bits
                  THE_BETA_VALUE:  BETA; -- 8 bits
                  THE_GAMMA_VALUE:  GAMMA; -- 8 bits
                  THE_DELTA_VALUE:  DELTA; -- 8 bits
        end case;
end record;
for THIRD_RECORD use record
        THE_DAY_OF_MONTH at 0 range 0 .. 7;
        THE_MONTH at 0 range 8 .. 15;
        THE_YEAR at 0 range 16 .. 31;
        THE_MEASUREMENT at 0 range 32 .. 63;
        THE_ALPHA_VALUE at 0 range 32 .. 39;
        THE_BETA_VALUE at 0 range 40 .. 47;
        THE_GAMMA_VALUE at 0 range 48 .. 55;
        THE_DELTA_VALUE at 0 range 56 .. 63;
end record;
for THIRD_RECORD'size use 64; -- 64 bits
```

Example 3-28: Complete Record Representation Clause Declaration

NOTE – The components ‘THE_MEASUREMENT’ and ‘THE_ALPHA_VALUE’ cannot appear in the same record, so their storage locations can overlap.

The following example (figure 3-42)presents the case of a complete representation clause, where components and associated representation clauses are not declared in the same order:

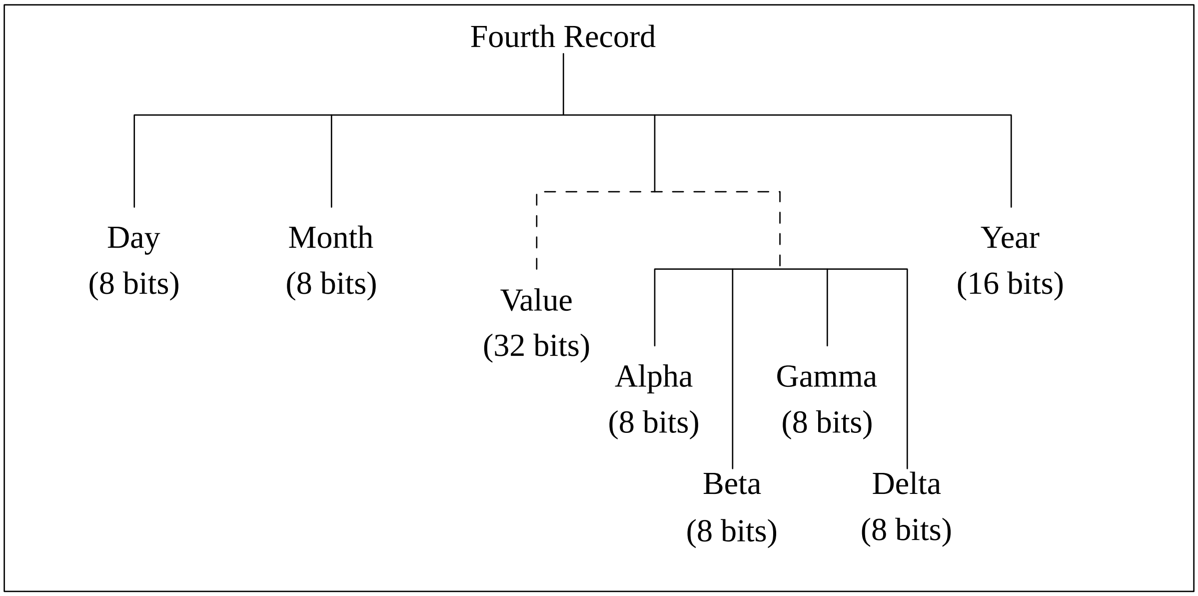


Figure 3-42: Fourth Tree Structure

The size of the record is known at definition time. The variant part has a constant length (32 bits). A fixed component is located after the variant part.

```
type FOURTH_RECORD (THE_DAY_OF_MONTH:  DAY := MON) is record
        THE_MONTH:  MONTH;
        THE_YEAR:  YEAR;
        case THE_DAY_OF_MONTH is
          when MON =>
                THE_MEASUREMENT:  VALUE; -- 32 bits
          when others =>
                THE_ALPHA_VALUE:  ALPHA; -- 8 bits
                THE_BETA_VALUE:  BETA; -- 8 bits
                THE_GAMMA_VALUE:  GAMMA; -- 8 bits
                THE_DELTA_VALUE:  DELTA; -- 8 bits
        end case;
end record;
for FOURTH_RECORD use record
        THE_DAY_OF_MONTH at 0 range 0 .. 7;
        THE_MONTH at 0 range 8 .. 15;
        THE_MEASUREMENT at 0 range 16 .. 47;
        THE_ALPHA_VALUE at 0 range 16 .. 23;
        THE_BETA_VALUE at 0 range 24 .. 31;
        THE_GAMMA_VALUE at 0 range 32 .. 39;
        THE_DELTA_VALUE at 0 range 40 .. 47;
        THE_YEAR at 0 range 48 .. 63;
end record;
for FOURTH_RECORD'size use 64; -- 64 bits
```

Example 3-29: Complete Record Representation Clause Declaration

The data item of the type YEAR is declared before the variant part in the record type declaration, but after the variant part in the record representation clause declaration.

The four previous examples are an illustration of the following rules:

1 The reasons for not providing a component representation clause are: the component has a variable size or it follows a component that has no component representation clause.

2 When no representation clause can be given for a component, its location is supposed to be contiguous to the previous component.

3 A fixed component is allowed after the variant part if that part has a constant length, i.e., if the location of the fixed component can be stated using a component representation clause.

The storage location of a component, relative to the start of the record, has been expressed until now in bits in the examples (the distance has been set to 0). For large structures, the values of expressions given after the reserved word **range** can be huge.

The EAST syntax also allows one to express the relative position of a component in distance to which a number of bits is added. For that purpose, EAST allows two units for the distance: WORD_16_BITS and WORD_32_BITS, representing respectively a 16-bit word and a 32-bit word.

WORD_16_BITS and WORD_32_BITS are two EAST predefined identifiers.

Distances are expressed in multiples of the selected unit as follows:

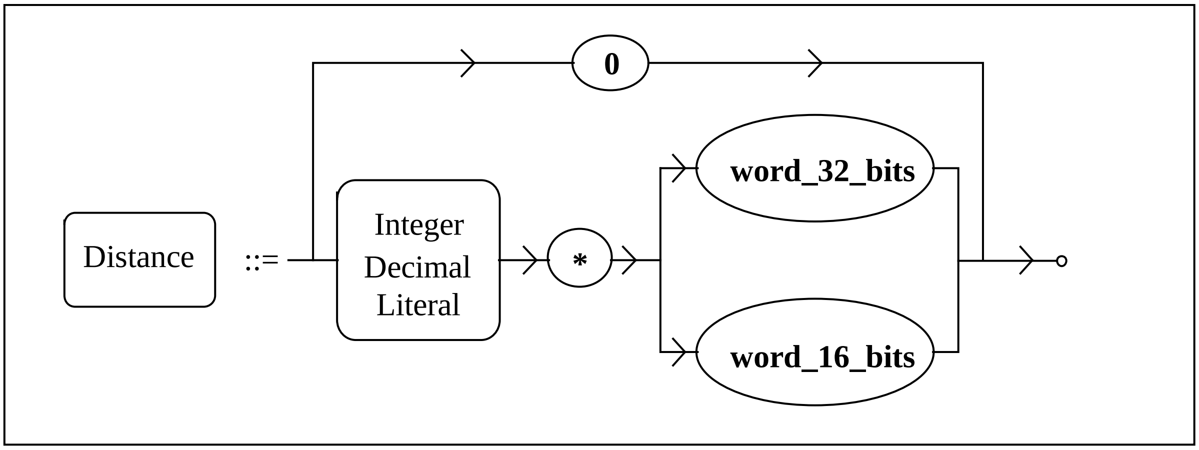


Figure 3-43: Distance Specification Diagram

NOTE – The integer decimal literal is the value of the distance expressed in the selected unit, either word_32_bits or word_16_bits.

See below for the previous record representation clause written using the constant WORD_32_BITS:

```
for THIRD_RECORD use record
        THE_DAY_OF_MONTH at 0 * WORD_32_BITS range 0 .. 7;
        THE_MONTH at 0 * WORD_32_BITS range 8 .. 15;
        THE_YEAR at 0 * WORD_32_BITS range 16 .. 31;
        THE_MEASUREMENT at 1 * WORD_32_BITS range 0 .. 31;
        THE_ALPHA_VALUE at 1 * WORD_32_BITS range 0 .. 7;
        THE_BETA_VALUE at 1 * WORD_32_BITS range 8 .. 15;
        THE_GAMMA_VALUE at 1 * WORD_32_BITS range 16 .. 23;
        THE_DELTA_VALUE at 1 * WORD_32_BITS range 24 .. 31;
end record;
```

Example 3-30: Record Representation Clause Using WORD_32_BITS

3.3 PHYSICAL DESCRIPTION

The physical description part adds implementation information to the logical part. While the logical part of the DDR describes the meaning of the exchanged data, the physical part describes how the data are physically implemented on the medium.

The machine-dependent characteristics include:

– the representation of numerics;

– the way of storing arrays on the medium;

– the way of storing octets on the medium.

This physical part of the Data Description Record consists of a package. See below the content of the physical part of a DDR.

package *physical_package_name* **is**
way of storing arrays (see 3.3.1)
way of storing octets (see 3.3.2)
actual scalar type representations (see 3.3.3)
association of basic type names with their actual representations (see 3.3.4)
end *physical_package_name* ;

The name of the physical package is an identifier (see 3.1.3) and must be different from the name of the package giving the associated logical description.

The physical description part has to be considered to be the instance of a template. Thus, the syntax used throughout this section is not justified or formally defined. An extended example of the template is provided in 3.3.5. The next subsections (3.3.1 to 3.3.4) explain the content of the template. Each time a declaration of the template must be used as it is, it is called ‘fixed part of the physical description’ as opposed to the declarations that change from a description to another one.

Every part of the template is optional (see 3.3.5). There is no required ordering between the different parts of the template.

3.3.1 STORING ARRAYS

An array object on a medium consists of a sequence of components. For a multi-dimensional array, i.e., an array with more than one index range, there are different methods to organize the sequence: either the first index range varies first or the last index range varies first. The first described method of storing arrays is called *first_index_first*, and the second one is called *last_index_first*.

The method for storing arrays on the medium is described in the physical description by using an enumeration type. See below the corresponding declaration:

```
type ARRAY_STORAGE_METHOD is (  FIRST_INDEX_FIRST,
                                LAST_INDEX_FIRST);
```

Fixed Part 3-1 of the Physical Description: Array Storage Method

Using this declaration, it is necessary to declare the actual method for storing arrays, for example:

```
ARRAY_STORAGE:  constant ARRAY_STORAGE_METHOD :=
                                FIRST_INDEX_FIRST;
```

Example 3-31: Actual Array Storage Method

This declaration is applicable to the whole description.

By default, the array storage is FIRST_INDEX_FIRST.

3.3.2 STORING OCTETS/BITS

The method used to store octets/bits determines the location of the Most Significant Bit (MSB) and the Least Significant Bit (LSB) of a data element.

A machine is said to be big-endian or little-endian depending on whether the MSB is in the lowest or highest addressed octet of the data element.

For a big-endian representation of a multi-octet data element, the MSB is in the first transmitted octet, i.e., in the first octet on the medium, while it is in the last transmitted octet, i.e., in the last octet on the medium, for a little-endian representation of a multi-octet data element.

The big-endian representation for a data element can be viewed as storing the bits from most to LSB order, and then keeping this same order when output to some medium.

The little-endian representation for a data element can be viewed as storing the bits from least to MSB order, but then re-ordering the bits (from most to least significant) within each octet when output to some medium.

This machine-dependent characteristic is very important for a correct interpretation of the data. Its definition is given for multi-octet data elements, but is still applicable for every data element, whatever its length and its position (on octet boundary or not) within the data set.

The following example presents the transmission of data elements for both kinds of machines.

Logically we have:

A 2 bits		B 3 bits			C 16 bits									D n bits			
A_1	A_2	B_1	B_2	B_3	C_1	C_2	C_3	C_4	....	C_{13}	C_{14}	C_{15}	C_{16}	D_1	D_2	....	D_n

When writing onto a medium, the machine writes the bits of the current octet first so that the contained data element bits are ordered from MSB to LSB while maintaining their relative bit positions to one another.

Therefore, for a big-endian machine where the bits are stored MSB first, the bit values in memory appear as follows:

A_1	A_2	B_1	B_2	B_3	C_1	C_2	C_3	C_4	............C_{11}.....	C_{13}	C_{14}	C_{15}	C_{16}	D_1	D_2	...	D_n
2^1	2^0	2^2	2^1	2^0	2^{15}	2^{14}	2^{13}	2^{12}	.............2^5......	2^3	2^2	2^1	2^0	2^{n-1}	2^{n-2}	...	2^0

The bits are transmitted towards the medium octet by octet in the following order:

A_1 A_2 B_1 B_2 B_3 C_1 C_2 C_3 then C_4 C_5 C_6 ... C_{11} then C_{12} C_{13} ... D_1 D_2 D_3 and so forth.

For a little-endian machine where the bits are stored LSB first, the bit values in memory appear as follows:

A_1	A_2	B_1	B_2	B_3	C_1	C_2	C_3	C_4	.............C_{11}...	C_{13}	C_{14}	C_{15}	C_{16}	D_1	D_2	...	D_n
2^0	2^1	2^0	2^1	2^2	2^0	2^1	2^2	2^3	.............2^{10}...	2^{12}	2^{13}	2^{14}	2^{15}	2^0	2^1	...	2^{n-1}

The bits are transmitted towards the medium octet by octet in the following order:

C_3 C_2 C_1 B_3 B_2 B_1 A_2 A_1 then C_{11} C_{10} C_9 ... C_4 then D_3 D_2 D_1 C_{16} ... C_{12} and so forth.

Example 3-32: Octet Storage Possibilities

See below the corresponding declaration:

```
type BIT_ORDER is ( HIGH_ORDER_FIRST,    -- big-endian representation
                    LOW_ORDER_FIRST);    -- little-endian representation
```

Fixed Part 3-2 of the Physical Description: Bit Order

Using this declaration, it is necessary to declare the actual way of storing octets, for example:

```
OCTET_STORAGE: constant BIT_ORDER := HIGH_ORDER_FIRST;
```

Example 3-33: Actual Bit Order

This declaration is applicable to the whole description.

The description of the way of storing octets (using the type BIT_ORDER) is sufficient to fully describe the organization on the medium (even at a bit level).

By default, the octet storage is HIGH_ORDER_FIRST.

3.3.3 REPRESENTATION OF SCALAR TYPES

Scalar types can be either binary encoded or ASCII encoded.

3.3.3.1 Binary Representation of Scalar Types

The way to determine the value of a numeric (integer or real), i.e., how to interpret its bit pattern on the medium, depends on its binary representation.

The binary representation of a numeric indicates its bit pattern on the medium. It includes the physical characteristics that may differ depending on the machine that has generated the numeric.

No binary representation is provided for enumeration types, because they are mapped on integers, for which the location of the bits from the MSB to the LSB are deduced from another physical information item, called bit order (see 3.3.2). If necessary, negative values are represented in a two's complement form.

If a negative value is present in the enumeration list, then the sign bit is present in any data occurrence of the enumeration type. If the sign bit is set, the two's complement shall be used to decode the integer value.

If all enumeration values are positive integers, then there is no sign bit and any data occurrence of the enumeration must be considered to be an unsigned integer.

The binary representation of an **integer** includes the following characteristics:

– the sign convention, which indicates the complementation, if any;

– the bit ordering, which indicates the location of MSB to the LSB, the sign position, if any, being the MSB.

The binary representation of a **real** includes the following characteristics:

– the sign position;

– the sign convention, if any;

– the location of the exponent;

– the bias, which is a constant chosen to make the sum of exponent value and bias which is a non-negative number;

– the exponent base, which is the integer (two, ten or sixteen) raised to the exponent power in determining the value of the represented number;

– the location of the mantissa.

It must additionally include the identifier of the convention of the generating machine, ‘convention of the generating machine’ being the method to reconstitute the real values from the previously defined characteristics. An Authority and Description Identifier (ADID) is associated with every registered convention. See reference [E5] for the list of conventions and related ADIDs.

The conventions adopted in this document for the data representation on a medium are the following:

– In multi-octet elements, the first octet is drawn in the leftmost position and is called ‘Octet Zero’. Successive octets are assigned successively larger numbers.

– Within an octet or binary field (not a multiple of octets), the first bit is drawn in the leftmost position and is called ‘Bit Zero’.

The following rule is applicable for a field representing an integer, an exponent or a mantissa of a real: the bits of the field are not necessarily provided in the right order (MSB to LSB) on the medium. The aim is to reconstitute the proper bit ordering (MSB to LSB). To achieve that, the initial field might be divided into an ordered sequence of subfields for which the bit ordering is respected in each of them. The order of the subfields provides the order of bits from the MSB to the LSB of the whole field.

Bit number	0	1	2	3	4	5	6	7	8	9
Significance	2^6	2^5	2^4	2^3	2^2	2^9	2^8	2^7	2^1	2^0

The bit ordering for this field from the MSB to LSB is: 5-6-7-0-1-2-3-4-8-9. This can be summarized using the previous rule in 3 subfields according to the bit numbers in the following order: (5 , 7) - (0 , 4) - (8 , 9).

Example 3-34: Bit Ordering

Using the previous conventions and rules, the binary representation of numerics is described in the corresponding physical description part. It contains:

– a fixed part declaring the types used to describe the representations (INTEGER_PHYSICAL_DESCRIPTION and REAL_PHYSICAL_DESCRIPTION), this part being always the same and present in any physical description part;

– a part declaring the actual representations used, i.e., a specific part, depending on the nature of the numerics to be described.

```
type NATURAL_NUMBER is range 0 .. 65535;

type LOCATION_OF_SUBFIELD is        -- subfields composing an integer or the
record                              -- exponent/mantissa of a real.
    BEGINNING_AT_BIT_NUMBER:  NATURAL_NUMBER;
    ENDING_AT_BIT_NUMBER:  NATURAL_NUMBER;
end record;

MAXIMUM_NUMBER_OF_SUBFIELDS:  constant := 255;
type SUBFIELD_NUMBER is range
    1 .. MAXIMUM_NUMBER_OF_SUBFIELDS;

type LOCATION_OF_FIELD is array (SUBFIELD_NUMBER range <>)
    of LOCATION_OF_SUBFIELD;
```

Fixed Part 3-3 of the Physical Description: Location of Fields for Numerics

NOTES

1 The MAXIMUM_NUMBER_OF_SUBFIELDS is set to 255. It is an arbitrary value that is big enough to cover all the identified machine architectures (i.e., the number of subfields that are necessary to locate the bits of an integer can be up to 255).

2 The upper bound of NATURAL_NUMBER is set to 65535. It is an arbitrary value that seems to be large enough in this context.

```
type SIGN_CONVENTION is (UNSIGNED, SIGN_AND_MAGNITUDE,
        ONES_COMPLEMENT, TWOS_COMPLEMENT);

type LIST_OF_RECOGNIZED_CONVENTIONS is (FCSTC000, FCSTC001,
        FCSTC0002, FCSTC0003); -- this list is not exhaustive (see reference [E5])

type INTEGER_PHYSICAL_DESCRIPTION (
        NUMBER_OF_SUBFIELDS:  SUBFIELD_NUMBER := 1) is record
        COMPLEMENT:  SIGN_CONVENTION;
        LOCATION:  LOCATION_OF_FIELD (1 .. NUMBER_OF_SUBFIELDS);
end record;

type REAL_PHYSICAL_DESCRIPTION(
        NUMBER_OF_SUBFIELDS_IN_EXPONENT:  SUBFIELD_NUMBER := 1;
        NUMBER_OF_SUBFIELDS_IN_MANTISSA:  SUBFIELD_NUMBER := 1)
is record
        CONVENTION_USED:  LIST_OF_RECOGNIZED_CONVENTIONS;
        SIGN_BIT_NUMBER:  NATURAL_NUMBER;
        COMPLEMENT:  SIGN_CONVENTION;
        EXPONENT_BASE:  NATURAL_NUMBER;
        BIAS:  NATURAL_NUMBER;
        LOCATION_OF_EXPONENT:  LOCATION_OF_FIELD (
                1 .. NUMBER_OF_SUBFIELDS_IN_EXPONENT);
        LOCATION_OF_MANTISSA:  LOCATION_OF_FIELD (
                1 .. NUMBER_OF_SUBFIELDS_IN_MANTISSA);
end record;
```

Fixed Part 3-4 of the Physical Description: Binary Description for Numerics

Each time the bits of an integer or the bits of the exponent or mantissa are not contiguously located on the medium from the MSB to the LSB (see example 3-34), several subfields are necessary to locate the bits. In these cases, BEGINNING_AT_BIT_NUMBER of the first element of the array LOCATION_OF_FIELD is supposed to be the bit number of the MSB. Bit numbers continue in sequence until ENDING_AT_BIT_NUMBER of the last element of LOCATION_OF_FIELD, which is supposed to be the bit number of the LSB.

The actual representation of the numerics is given by the declaration of constants of the previous record types (INTEGER_PHYSICAL_DESCRIPTION for the representation of integers and REAL_PHYSICAL_DESCRIPTION for the representation of reals).

The actual representation of a numeric is therefore provided by a record value (i.e., the value of the constant of the relevant record type: INTEGER_PHYSICAL_DESCRIPTION or REAL_PHYSICAL_DESCRIPTION).

Figure 3-44 illustrates the syntax of a record value.

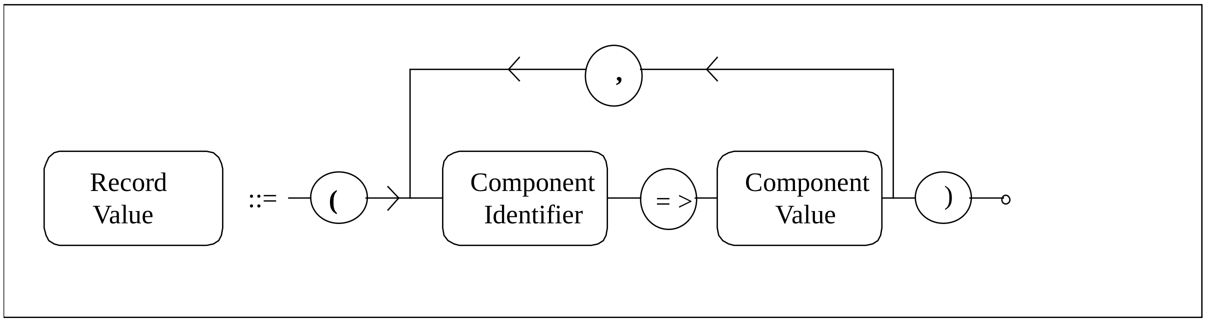


Figure 3-44: Record Value Specification Diagram

In the case of the record types used in the physical part of an EAST description, the component value is either an enumeration literal, an integer literal or an array value (see figure 3-45).

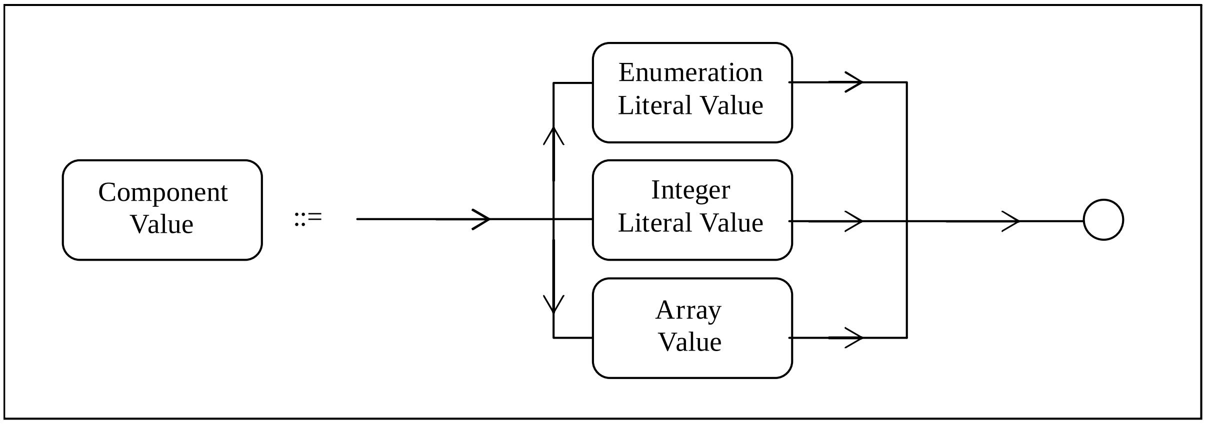


Figure 3-45: Component Value Definition Diagram

Figure 3-46 illustrates the syntax of an array value.

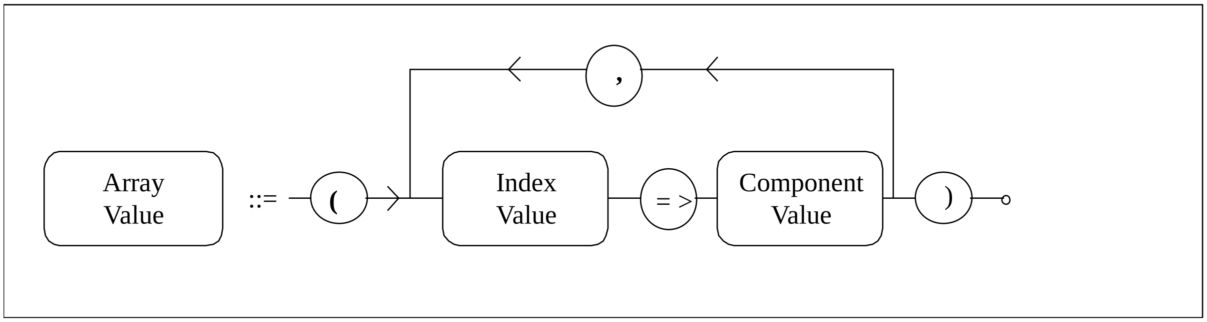
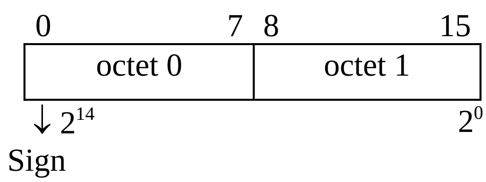


Figure 3-46: Array Value Specification Diagram

The index value is an integer literal. In the case of the array types used in the physical part of an EAST description, the component value is either an enumeration literal, an integer literal, an array value, or a record value.

The following examples present real cases of two integers and a real that must be described.

A 16-bit signed integer with the following physical representation (big-endian representation):



– The sign position is bit 0.

– The bit ordering is (0,15), which means that the MSB is bit 1 (bit 0 being the sign bit) and the LSB is bit 15.

Example 3-35: Bit Ordering for the Above 16-Bit Signed Integer

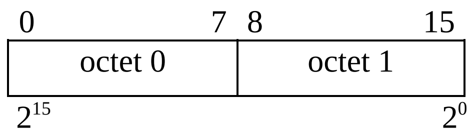
Using the types declared in the fixed part of the physical description, it is possible to declare the actual binary representation of this integer. Assuming that for negative values the two's complement is used, the actual binary representation is given by the following declaration:

```
Binary_Representation_01: constant INTEGER_PHYSICAL_DESCRIPTION :=
                (NUMBER_OF_SUBFIELDS => 1 ,
                COMPLEMENT => TWOS_COMPLEMENT,
                LOCATION => (1 => (0,15)));
```

Example 3-36: Actual Binary Representation of the Above 16-Bit Signed Integer

In this example, the binary representation indicates that the sign bit is the first bit encountered (bit 0). Then, a less significant bit is the second bit encountered (bit 1) and so on till the sixteenth bit (this bit being the LSB of the integer).

In the same way, a 16-bit unsigned integer with the following physical representation (big-endian representation):



– The bit ordering is (0,15), which means that the MSB is bit 0 and LSB is bit 15.

Example 3-37: Bit Ordering for the Above 16-Bit Unsigned Integer

Using the types declared in the fixed part of the physical description, it is possible to declare the actual binary representation of this integer. The actual binary representation is given by the following declaration:

```
Binary_Representation_02: constant INTEGER_PHYSICAL_DESCRIPTION :=
        (NUMBER_OF_SUBFIELDS => 1 ,
        COMPLEMENT => UNSIGNED,
        LOCATION => (1 => (0,15)));
```

Example 3-38: Actual Binary Representation of the Above 16-Bit Unsigned Integer

In this example, the binary representation indicates that the most significant is the first bit encountered (bit 0). Then, a less significant bit is the second bit encountered (bit 1) and so on until the sixteenth bit (this bit being the LSB of the integer).

If the range that is specified in the integer type definition (in the logical part of the EAST description) allows negative values, then there is a sign bit, and the SIGN_CONVENTION cannot be UNSIGNED. If this range specifies only positive values, then there can be a sign bit (or not) according to the SIGN_CONVENTION. If there is no sign bit, the first bit number of the first subfield really corresponds to the MSB.

A 32-bit real with the following physical representation (little-endian representation):

0 7	8 15	16 23	24 31
octet 0	octet 1	octet 2	octet 3

← Mantissa → ← Mantissa →↓← Mantissa →↓← Exponent →
Exponent Sign

- The sign position is bit 24.
- The location of the exponent includes two subfields (25,31) and (16,16), which means that the MSB of the exponent is bit 25 and the LSB is bit 16.
- The location of the mantissa includes three subfields (17,23), (8,15) and (0,7), which means that the MSB of the mantissa is bit 17 and the LSB is bit 7.

Example 3-39: Bit Ordering for the Above 32-Bit Real

Using the types declared in the fixed part of the physical description, it is possible to declare the actual binary representation of this real. Assuming that the real is generated on a PC (which uses the IEEE 754 convention, identified by FCSTC000, see reference[E5]), the actual binary representation is given by the following declaration:

```
Binary_Representation_03: constant REAL_PHYSICAL_DESCRIPTION :=
        (NUMBER_OF_SUBFIELDS_IN_EXPONENT => 2,
        NUMBER_OF_SUBFIELDS_IN_MANTISSA => 3,
        CONVENTION_USED => FCSTC000, -- IEEE 754
        SIGN_BIT_NUMBER => 24,
        COMPLEMENT => SIGN_AND_MAGNITUDE,
        EXPONENT_BASE => 2,
        BIAS => 127,
        LOCATION_OF_EXPONENT => (  1 => (25,31),
                                   2 => (16,16)),
        LOCATION_OF_MANTISSA => (  1 => (17,23),
                                   2 => (8,15),
                                   3 => (0,7)));
```

Example 3-40: Actual Binary Representation of a 32-Bit Real

In this example, the binary representation indicates that the most significant bit of the exponent is the twenty-sixth bit encountered (bit 25). Then from bit 26 through bit 31 the bits encountered are less significant, and bit 16 is the LSB of the exponent.

In the same way, the most significant bit of the mantissa is the eighteenth bit encountered (bit 17). Then from bit 18 through bit 23, and then from bit 8 through bit 15, and from bit 0 through bit 7, the bits encountered are less significant, bit 7 being the LSB of the mantissa.

NOTE – The name of the constant used to identify the binary representation (Binary_Representation_01 or Binary_Representation_02) could be any identifier (except a reserved keyword). The only restriction is that a constant identifier cannot be defined twice in the physical part.

Reference [E5] provides the way of calculating real values for the conventions, mentioned in the definition of LIST_OF_RECOGNIZED_CONVENTIONS.

3.3.3.2 ASCII Representation of Scalar Types

ASCII encoded types are sometimes used to increase the portability of the data. Enumeration types, integer types, and real types can be encoded using character strings. An ASCII encoded type is a character string type with a specific format, depending on the nature of the type (enumeration, integer, or real).

There is no difference (except the size) between the logical description of a binary type and the logical description of an ASCII encoded type. The physical description specifies the actual representation of the scalar types. By default, a type is a binary encoded type. An ASCII representation must be associated with the type name, if the type is ASCII encoded.

The ASCII representation of an **enumeration** type provides all the character strings associated with all the enumeration literals of the type. The character strings, which are the coding values of the enumeration type, have all the same length (NUMBER_OF_CHARACTERS). The set of the coding values is therefore represented by a character string list, which is also an array of characters, dimensioned by the NUMBER_OF_OCCURRENCES of the enumeration type and the NUMBER_OF_CHARACTERS of every occurrence.

The ASCII representation of an enumeration uses the following types:

```
type STRING_LIST is array( NATURAL_NUMBER range <>,
                           NATURAL_NUMBER range <>) of CHARACTER;



type ASCII_ENUMERATION_PHYSICAL_DESCRIPTION (
      NUMBER_OF_OCCURRENCES:  NATURAL_NUMBER := 0;
      NUMBER_OF_CHARACTERS:  NATURAL_NUMBER := 0) is record
      REPRESENTATION:  STRING_LIST ( 1 .. NUMBER_OF_OCCURRENCES,
                                     1 .. NUMBER_OF_CHARACTERS);

end record;
```

Fixed Part 3-5 of the Physical Description: ASCII Description for Enumeration Types

The number of characters used to encode the enumeration type must be the same for every enumeration literal of the type. This number is known at definition time.

All characters (i.e., the 256 characters of the Latin Alphabet No. 1—see reference [1] and/or annex B) are allowed and are significant, including the space character.

The physical representations of the enumeration literals are provided in the order of their declaration in the logical part.

An enumeration type is either an ASCII encoded type (in this case, its ASCII representation shall be present in the physical description part) or a binary encoded type (in this case, an enumeration representation clause can be present in the logical description part). In any case, enumeration representation clause and ASCII representation are exclusive: they must not be associated with the same enumeration type.

Using the types declared in the fixed part of the physical description, it is possible to declare the actual ASCII representation of the enumeration types.

For example, an enumeration type which has two permitted values: ‘WORKING’ and ‘IDLE’, identifying a process, can be described in the logical part as follows:

```
type PROCESS_IDENTIFICATION is (WORKING, IDLE);
for PROCESS_IDENTIFICATION'size use 56; -- bits, i.e., 7 characters
```

Example 3-41: ASCII Enumeration Type Logical Declaration

and in the physical part as follows:

```
ASCII_Rep_01:  constant ASCII_ENUMERATION_PHYSICAL_DESCRIPTION :=
        (NUMBER_OF_OCCURRENCES => 2, NUMBER_OF_CHARACTERS => 7,
        REPRESENTATION => ("WORKING" , "IDLE   "));
```

Example 3-42: ASCII Enumeration Type Physical Description

In this example, three space characters belong to the representation of the enumeration value IDLE.

An ASCII Encoded Decimal Integer is a character string representing an integer value. The format of the character string corresponding to an ASCII encoded decimal integer is described in figure 3-47:

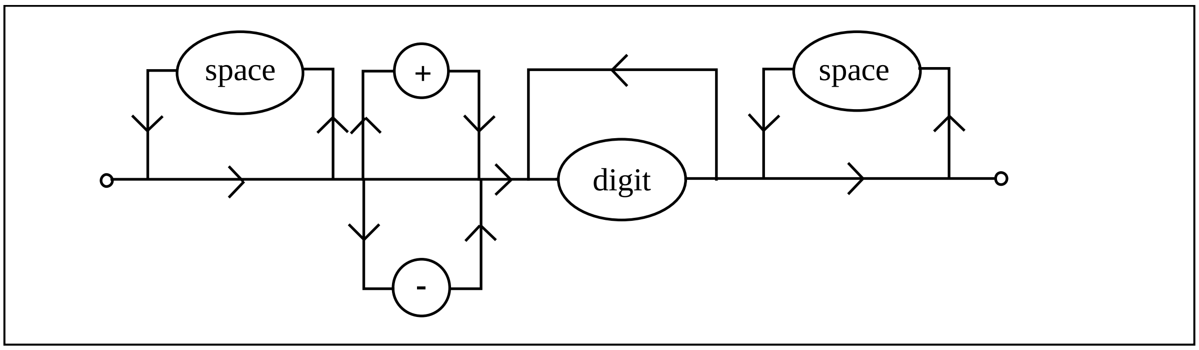


Figure 3-47: ASCII Encoded Decimal Integer Format

An ASCII Encoded Decimal Real is a string representing a real value. The format of the character string corresponding to an ASCII encoded decimal real is described in figure 3-48:

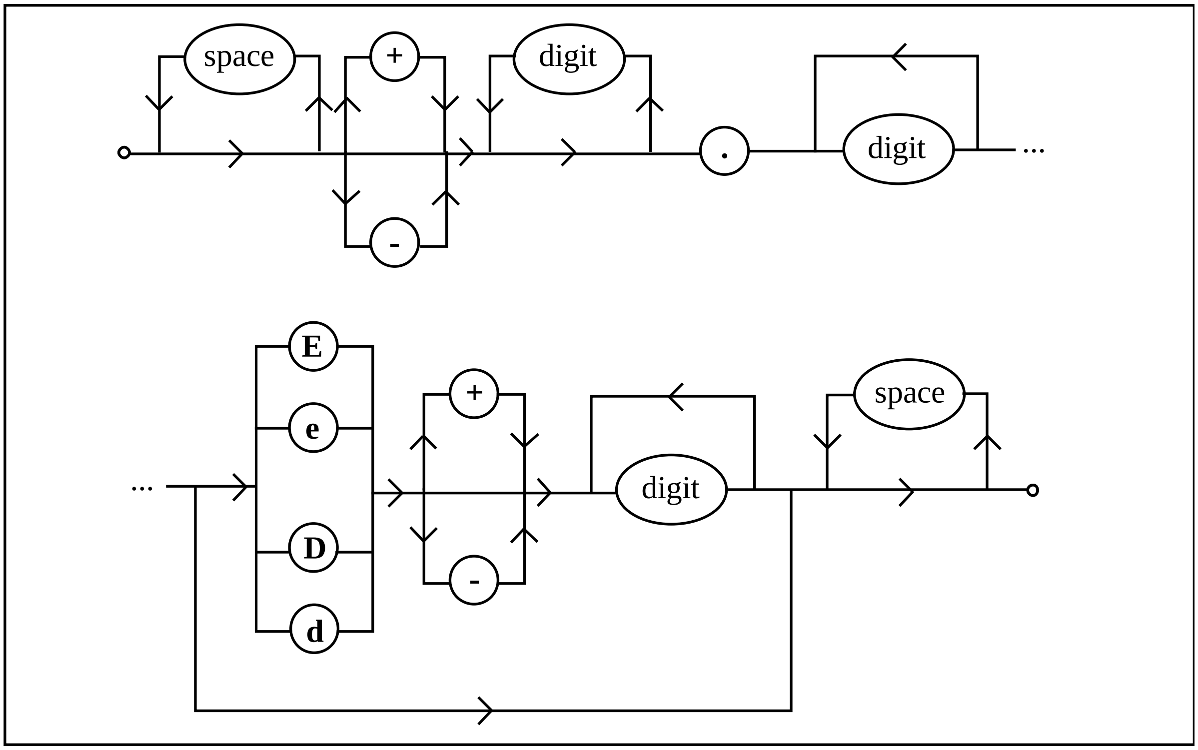


Figure 3-48: ASCII Encoded Decimal Real Format

A digit is one of the following characters: ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’.

Only the normalized ASCII encoded numbers can be described using EAST. There is no convention for the ASCII representation of infinite values (‘+INF’, ‘-INF’ or ‘+ ∞’, ‘- ∞’) and no representation for ‘NaN’ (Not a Number).

The ASCII representation of an **integer** or **real** type specifies the number of characters used for the integer or real values. The ASCII representation of an integer or real uses the following type:

```
type ASCII_NUMERIC_PHYSICAL_DESCRIPTION is record
        NUMBER_OF_CHARACTERS: NATURAL_NUMBER;
end record;
```

Fixed Part 3-6 of the Physical Description: ASCII Description for Numerics

Using the types declared in the fixed part of the physical description, it is possible to declare the actual ASCII representation of the numerics.

For example, a five-character ASCII decimal integer type can be described in the logical part as follows:

```
type COUNTER is range -1 .. 16383;
for COUNTER'size use 40; -- bits, i.e., 5 characters
```

Example 3-43: ASCII Integer Type Logical Declaration

and in the physical part as follows:

```
ASCII_Rep_02: constant ASCII_NUMERIC_PHYSICAL_DESCRIPTION :=
        (NUMBER_OF_CHARACTERS => 5);
```

Example 3-44: ASCII Integer Type Physical Description

For example, an 11-character ASCII decimal real type can be described in the logical part as follows:

```
type KILOMETERS is digits 5;
for KILOMETERS'size use 88; -- bits
```

Example 3-45: ASCII Real Type Logical Declaration

and in the physical part as follows:

```
ASCII_Rep_03: constant ASCII_NUMERIC_PHYSICAL_DESCRIPTION :=
        (NUMBER_OF_CHARACTERS => 11);
```

Example 3-46: ASCII Real Type Physical Description

NOTE – The name of the constant used to identify the ASCII representation (ASCII_Rep_01 or ASCII_Rep_02 or ASCII_Rep_03) could be any identifier (except a reserved keyword). The only restriction is that a constant identifier cannot be defined twice in the physical part.

3.3.4 RELATIONSHIP BETWEEN THE REPRESENTATION OF SCALAR TYPES AND LOGICAL TYPES

As seen in 3.3.3, a binary or ASCII representation is provided for some basic types (enumeration, integer, or real types) defined in the logical part of the DDR. The association of a type name with the corresponding representation name also has to be provided in this physical description part. See below how this association is implemented in EAST:

– an enumeration type gives all the basic type names, which are previously defined in the logical description part and which need a physical representation, by prefixing them with ‘USER_TYPE_’:

```
type BASIC_TYPE_NAMES is (USER_TYPE_xxx , USER_TYPE_yyy ,
                    USER_TYPE_zzz, USER_TYPE_ttt);
```

– the different representations are declared as seen in 3.3.3.1 and 3.3.3.2:

```
Binary_Representation_01:  constant INTEGER_PHYSICAL_DESCRIPTION
                    := "value"; -- integer type
Binary_Representation_02:  constant REAL_PHYSICAL_DESCRIPTION
                    := "value"; -- real type
ASCII_Representation_01:  constant
                    ASCII_NUMERIC_PHYSICAL_DESCRIPTION
                    := "value"; -- integer or real type
ASCII_Representation_02:  constant
                    ASCII_ENUMERATION_PHYSICAL_DESCRIPTION
                    := "value"; -- enumeration type
... and so forth ...
```

– finally, the relation between the type names and their binary representations is specified as follows:

```
type RELATION(choice:  BASIC_TYPE_NAMES) is record
          case choice is
                    when USER_TYPE_xxx =>
                    PHYS_xxx:  INTEGER_PHYSICAL_DESCRIPTION
                              := Binary_Representation_01;
                    when USER_TYPE_yyy =>
                    PHYS_yyy:  REAL_PHYSICAL_DESCRIPTION
                              := Binary_Representation_02;
                    when USER_TYPE_zzz =>
                    PHYS_zzz:  ASCII_NUMERIC_PHYSICAL_DESCRIPTION
                              := ASCII_Representation_01;
                    when USER_TYPE_ttt =>
                    PHYS_ttt:  ASCII_ENUMERATION_PHYSICAL_DESCRIPTION
                              := ASCII_Representation_02;
                    and so forth ...
          end case;
end record;
```

3.3.5 TEMPLATE OF A PHYSICAL DESCRIPTION PART

This subsection gives an extended template for the physical description part definition. The italicized part corresponds to the variable part of the description, i.e., what changes from a physical part to another physical part.

```
package physical_package_name is

type ARRAY_STORAGE_METHOD is (   FIRST_INDEX_FIRST,
                                 LAST_INDEX_FIRST);
ARRAY_STORAGE:  constant ARRAY_STORAGE_METHOD :=
                                 FIRST_INDEX_FIRST;

type BIT_ORDER is (          HIGH_ORDER_FIRST,       -- big-endian representation
                 LOW_ORDER_FIRST);        -- little-endian representation
OCTET_STORAGE:  constant BIT_ORDER := HIGH_ORDER_FIRST;

type LOCATION_OF_SUBFIELD is          -- subfields composing an integer or the
record                                          -- exponent/mantissa of a real.
        BEGINNING_AT_BIT_NUMBER:  NATURAL_NUMBER;
        ENDING_AT_BIT_NUMBER:  NATURAL_NUMBER;
end record;

MAXIMUM_NUMBER_OF_SUBFIELDS:  constant := 255;
type SUBFIELD_NUMBER is range
        1 .. MAXIMUM_NUMBER_OF_SUBFIELDS;

type LOCATION_OF_FIELD is array (SUBFIELD_NUMBER range <>)
        of LOCATION_OF_SUBFIELD;

type SIGN_CONVENTION is (UNSIGNED, SIGN_AND_MAGNITUDE,
        ONES_COMPLEMENT, TWOS_COMPLEMENT);

type LIST_OF_RECOGNIZED_CONVENTIONS is (FCSTC000);

type INTEGER_PHYSICAL_DESCRIPTION (
        NUMBER_OF_SUBFIELDS:  SUBFIELD_NUMBER := 1) is record
        COMPLEMENT:  SIGN_CONVENTION;
        LOCATION:  LOCATION_OF_FIELD (1 .. NUMBER_OF_SUBFIELDS);
end record;
```

```
type REAL_PHYSICAL_DESCRIPTION(
        NUMBER_OF_SUBFIELDS_IN_EXPONENT:  SUBFIELD_NUMBER := 1;
        NUMBER_OF_SUBFIELDS_IN_MANTISSA:  SUBFIELD_NUMBER := 1)
is record
        CONVENTION_USED:  LIST_OF_RECOGNIZED_CONVENTIONS;
        SIGN_BIT_NUMBER:  NATURAL_NUMBER;
        COMPLEMENT:  SIGN_CONVENTION;
        EXPONENT_BASE:  NATURAL_NUMBER;
        BIAS:  NATURAL_NUMBER;
        LOCATION_OF_EXPONENT:  LOCATION_OF_FIELD (
                  1 .. NUMBER_OF_SUBFIELDS_IN_EXPONENT);
        LOCATION_OF_MANTISSA:  LOCATION_OF_FIELD (
                  1 .. NUMBER_OF_SUBFIELDS_IN_MANTISSA);
end record;

type STRING_LIST is array(        NATURAL_NUMBER range <>,
                  NATURAL_NUMBER range <>) of CHARACTER;
type ASCII_ENUMERATION_PHYSICAL_DESCRIPTION (
        NUMBER_OF_OCCURRENCES:  NATURAL_NUMBER := 0;
        NUMBER_OF_CHARACTERS:  NATURAL_NUMBER := 0) is record
        REPRESENTATION:  STRING_LIST (      1 .. NUMBER_OF_OCCURRENCES,
                                            1 .. NUMBER_OF_CHARACTERS);
end record;

type ASCII_NUMERIC_PHYSICAL_DESCRIPTION is record
        NUMBER_OF_CHARACTERS:  NATURAL_NUMBER;
end record;

Binary_Representation_01: constant INTEGER_PHYSICAL_DESCRIPTION :=
                  (NUMBER_OF_SUBFIELDS => 1 ,
                  COMPLEMENT => TWOS_COMPLEMENT,
                  LOCATION => (1 => (0,15)));
 Binary_Representation_02:constant REAL_PHYSICAL_DESCRIPTION :=
                  (NUMBER_OF_SUBFIELDS_IN_EXPONENT => 1,
                  NUMBER_OF_SUBFIELDS_IN_MANTISSA => 1,
                  CONVENTION_USED => FCSTC000,
                  SIGN_BIT_NUMBER => 0,
                  COMPLEMENT => SIGN_AND_MAGNITUDE,
                  EXPONENT_BASE => 2,
                  BIAS => 127,
                  LOCATION_OF_EXPONENT => (  1 => (1,8),
                  LOCATION_OF_MANTISSA => (  1 => (9,31)));
ASCII_Rep_01:  constant ASCII_ENUMERATION_PHYSICAL_DESCRIPTION :=
        (NUMBER_OF_OCCURRENCES => 2, NUMBER_OF_CHARACTERS => 7,
        REPRESENTATION => ("WORKING" , "IDLE"));
```

```
ASCII_Rep_02:  constant ASCII_NUMERIC_PHYSICAL_DESCRIPTION :=
        (NUMBER_OF_CHARACTERS => 5);

type BASIC_TYPE_NAMES is (USER_TYPE_xxx , USER_TYPE_yyy ,
                            USER_TYPE_zzz , USER_TYPE_ttt);

type RELATION(choice:  BASIC_TYPE_NAMES) is record
        case choice is
                when USER_TYPE_xxx =>
                PHYS_xxx:  INTEGER_PHYSICAL_DESCRIPTION
                        := Binary_Representation_01;
                when USER_TYPE_yyy =>
                PHYS_yyy:  REAL_PHYSICAL_DESCRIPTION
                        := Binary_Representation_02;
                when USER_TYPE_zzz =>
                PHYS_zzz:  ASCII_ENUMERATION_PHYSICAL_DESCRIPTION
                        := ASCII_Rep_01;
                PHYS_ttt:  ASCII_NUMERIC_PHYSICAL_DESCRIPTION
                        := ASCII_Rep_02;
        end case;
end record;
end physical_package_name;
```

Most of the declarations are optional. Indeed only the types that are used must be declared. As an example, the type REAL_PHYSICAL_DESCRIPTION must be defined only if it is used in the physical part, i.e., if at least one real type is defined in the logical part.

The following rules apply:

1 The array storage is optional (ARRAY_STORAGE_METHOD type and ARRAY_STORAGE constant) if there is no multi-dimensional array in the logical part, or if the method is FIRST_INDEX_FIRST (default value).

2 The octet storage is optional (BIT_ORDER type and OCTET_STORAGE constant) if the method is HIGH_ORDER_FIRST (default value).

3 The type REAL_PHYSICAL_DESCRIPTION is optional if there is no binary representation for real type to provide, i.e., if there is no binary real type in the logical part.

4 The type INTEGER_PHYSICAL_DESCRIPTION is optional if there is no binary representation for integer type to provide, i.e., if there is no binary integer type in the logical part or if they are all considered to be unsigned integers or two’s-complement signed integers.

5 The type ASCII_ENUMERATION_PHYSICAL_DESCRIPTION is optional if there is no ASCII representation for enumeration type to provide, i.e., if there is no ASCII enumeration type in the logical part.

6 The type ASCII_NUMERIC_PHYSICAL_DESCRIPTION is optional if there is no ASCII representation for integer or real type to provide, i.e., if there is no ASCII integer type and no ASCII real type in the logical part.

7 The types BASIC_TYPE_NAMES and RELATION are optional if there is no representation to provide.

4 RESERVED KEYWORDS

The following reserved keywords are not available for use as declared identifiers. Some of them are reserved keywords of the Ada programming language (see reference [E3]), but not of the EAST language. These words are also reserved so that in the case of an Ada application using an EAST description in accessing the data, the syntax will be compatible, although not equivalent in meaning. Differences between Ada and EAST syntax interpretations are discussed in annex D. Other words are reserved identifiers of the EAST language and not of the Ada programming language.

a) EAST and Ada Keywords

array	digits	is	package	type
at				
	end	null	range	use
case			record	
constant	for	of		when
		others	subtype	

b) Other Ada Keywords

abort	delta	if	pragma	tagged
abs	do	in	private	task
abstract			procedure	terminate
accept	else	limited	protected	then
access	elsif	loop		
aliased	entry		raise	until
all	exception	mod	rem	
and	exit		renames	while
		new	requeue	with
begin	function	not	return	
body			reverse	xor
	generic	or		
declare	goto	out	select	
delay			separate	

c) Pure EAST reserved identifiers

virtual_...	word_32_bits	word_16_bits	east_version	virtual

NOTE – Any identifier beginning with ‘virtual_’ is reserved for virtual component identifier only.

5 CONFORMANCE

Data conforming to a Recommendation may be said to be in conformance at some identified level. Identifying conformance levels provides a standard way to classify the required capabilities of generating and receiving systems.

The Recommendation for Data Description Language EAST Specification recognizes only one conformance level, and that is the entire specification. Therefore recipient systems which are said to be in conformance to this Recommendation shall recognize the entire specification.

ANNEX A

ACRONYMS AND GLOSSARY

(This annex **is** part of the Recommendation)

This annex defines key acronyms and the glossary of terms which are used throughout this Recommendation to describe the Data Description Language EAST.

A1 ACRONYMS

ADID	Authority and Description IDentifier
ASCII	American Standard Code for Information Interchange
BNF	Backus-Naur-Form
DDR	Data Description Record
EAST	Enhanced Ada SubseT
ISO	International Standards Organization
LSB	Least Significant Bit
LSO	Least Significant Octet
MSB	Most Significant Bit
MSO	Most Significant Octet

A2 GLOSSARY OF TERMS

ADID: in the context of EAST, an ADID is an identifier of the EAST Recommendation within the CCSDS organization. See reference [E2].

Array type: an array type is a composite type whose components are all of the same type. Components are selected by indexing.

Based literal: a based literal is a numeric literal expressed in a form that specifies the base explicitly.

Character literal: a character literal is formed by enclosing a graphic character between two apostrophe characters.

Character type: a character type is an enumeration type that represents a character set.

Composite type: a composite type is a collection of components of the same or different types.

Constant: a constant is a keyword that indicates that the identifier it qualifies has a unique and specified value.

Constrained array: a constrained array is an array with a constant number of elements.

Discrete type: a discrete type is either an integer type or an enumeration type. Discrete types may be used, for example, in case statements and as array indexes.

Discriminant: a discriminant is a component of a record type whose value influences the structure of this record.

Elementary type: an elementary type does not have components.

Enumeration representation clause: an enumeration representation clause specifies the bit pattern for each literal of the corresponding enumeration type.

Enumeration type: an enumeration type is defined by the list of its values, called enumeration literals, which may be identifiers or character literals. All values for a given enumeration type are different.

Length clause: a length clause specifies the amount of storage in bits associated with a type.

Literal: a literal is a value represented by its value itself instead of an identifier. A literal can be specialized as a numeric literal, an enumeration literal, a character literal, or a string literal.

Marker: a marker is a constant value provided by a data description. This value will be found in the data as an end-delimiter of a repetition.

Numeric literal: a numeric literal is the value of a number, expressed by means of characters.

Object: an object is either a constant or a variable. An object contains a value.

Predefined type: a predefined type is a type provided by EAST, that is, a type that can be used in any EAST description without being previously declared.

Record representation clause: a record representation clause specifies the storage representation of the record type on the medium, that is, the order, position and size of record components (including discriminants, if any).

Record type: a record type is a composite type consisting of zero or more named components, possibly of different types.

Representation clause: representation clauses specify the mapping between types of the language and their physical representation.

Scalar type: scalar types are discrete types and real types.

String literal: a string literal is formed by a sequence of graphic characters (possibly none) enclosed between two quotation marks used as string brackets.

Subtype: a subtype is a type together with a constraint, which constrains the values of the type to satisfy a certain condition. The values of a subtype are a subset of the values of its type.

Type: a type is a named set of characteristics. This name can be used to define sets of values.

Unconstrained array: an unconstrained array is an array with a variable number of elements.

Variable: a variable is an identifier that represents a data item occurrence.

Variant part: a variant part of a record specifies alternative record components, dependent on the discriminant of the record. Each value of the discriminant establishes a particular alternative of the variant part.

Virtual Discriminant: a virtual discriminant is a discriminant that is not included in the composite type that it discriminates.

ANNEX B

CHARACTER DEFINITION

(This annex **is** part of the Recommendation)

This annex contains the definition of the character set used for the EAST predefined type CHARACTER.

This character set is a subset of the 16-bit Basic Multilingual Plane (BMP) of the ISO 10646 coded character set (reference [2]). This subset is defined as the first 256 characters (row00) of the BMP, which corresponds to the ISO 8859-1, which is an 8-bit single-byte coded graphic character set, also known as 'Latin Alphabet No. 1' (reference [1]). The corresponding codes are shown in the following tables. *(The code for each character (Char) is given in decimal (Dec), and hexadecimal (Hex).)*

The whole of the ISO 8859-1 character set shown in the following tables is permitted in the data that conforms to this Recommendation, although for interpretation purposes the characters shaded in the following tables are ignored and should not be displayed or printed.

The use of an ISO 8859-1 encoding to represent the natural language also permits the incorporation of tables and figures that can be drawn with the characters listed below. For these figures or tables to be presented identically to any receiver, the interpretation of the control characters (Vertical Tab, Horizontal Tab, Form Feed, Line Feed *(also known as New Line)* and Carriage Return) must be standardized. The following rules apply:

a) A new line (positioning the next displayable character to the leftmost displayable position and one line down) for presentation purposes is understood to occur upon encountering the following conditions:

   1) a Carriage Return, when it is not followed by a Line Feed;

   2) a Carriage Return/Line Feed pair, regardless of what follows;

   3) a Line Feed, when it is not followed by a Carriage Return;

   4) a Line Feed/Carriage Return pair, regardless of what follows.

b) A Horizontal Tab character positions the next displayable character onto the next character position that is a multiple of 8 (i.e., character positions 8, 16, 24, 32 etc., where the leftmost displayable character position is 0).

c) A Form Feed character positions the next displayable character to the leftmost displayable position and down to the beginning of the next page. The definition of a page is as defined by the local device (e.g., a new screen for a visual display unit (VDU) or a new piece of paper for a printer).

d) If the characteristics of the display device conflict with those of the data, for example, line lengths may be greater than those permitted by the device, then some adjustment to the layout of the data, as determined by the device, will occur. *(Note also that some devices may process or react to codes which this Recommendation specifies as being ignored for presentation purposes.)*

NOTE – If the alignment of the displayed characters is significant to the understanding of the information, then a fixed space font should be used for presentation.

Some of the defined characters need some explanations: Space (SP), No_Break_Space (NBSP), Soft_Hyphen (SHY) and Res.

a) A space might be interpreted as a graphic character, or a control character or both. As a graphic character, its representation consists of no symbol, but it takes up display space.

b) A No_Break_Space is a graphic character for which the representation consists of no symbol. It shall be used when no break (new line) is allowed.

c) A Soft_Hyphen is a graphic character with the following representation: '-'. It occurs when a word is broken up because of a new line.

d) *Res* represents a reserved control character. It is anyway ignored in EAST.

The language identified by the **ADID = CCSD0010** is EAST. The character set to be used in EAST descriptions is a subset of the ISO 8859-1 (reference [1]). This subset is defined as **the first 128 characters** of the 8-bit single byte coded graphic character set (from the decimal code 0 up to the decimal code 127 in the following tables).

NOTE – The character set used in EAST descriptions should not be confused with the character set of the predefined type CHARACTER that describes occurrences of data.

Char	Dec	Hex
NUL	0	00
SOH	1	01
STX	2	02
ETX	3	03
EOT	4	04
ENQ	5	05
ACK	6	06
BEL	7	07
BS	8	08
HT	9	09
LF	10	0A
VT	11	0B
FF	12	0C
CR	13	0D
SO	14	0E
SI	15	0F
DLE	16	10
DC1	17	11
DC2	18	12
DC3	19	13
DC4	20	14
NAK	21	15
SYN	22	16
ETB	23	17
CAN	24	18
EM	25	19
SUB	26	1A
ESC	27	1B
FS	28	1C
GS	29	1D
RS	30	1E
US	31	1F

Char	Dec	Hex
space	32	20
!	33	21
“	34	22
#	35	23
$	36	24
%	37	25
&	38	26
‘	39	27
(	40	28
)	41	29
*	42	2A
+	43	2B
,	44	2C
-	45	2D
.	46	2E
/	47	2F
0	48	30
1	49	31
2	50	32
3	51	33
4	52	34
5	53	35
6	54	36
7	55	37
8	56	38
9	57	39
:	58	3A
;	59	3B
<	60	3C
=	61	3D
>	62	3E
?	63	3F

Char	Dec	Hex
@	64	40
A	65	41
B	66	42
C	67	43
D	68	44
E	69	45
F	70	46
G	71	47
H	72	48
I	73	49
J	74	4A
K	75	4B
L	76	4C
M	77	4D
N	78	4E
O	79	4F
P	80	50
Q	81	51
R	82	52
S	83	53
T	84	54
U	85	55
V	86	56
W	87	57
X	88	58
Y	89	59
Z	90	5A
[	91	5B
\	92	5C
]	93	5D
^	94	5E
_	95	5F

Char	Dec	Hex
`	96	60
a	97	61
b	98	62
c	99	63
d	100	64
e	101	65
f	102	66
g	103	67
h	104	68
i	105	69
j	106	6A
k	107	6B
l	108	6C
m	109	6D
n	110	6E
o	111	6F
p	112	70
q	113	71
r	114	72
s	115	73
t	116	74
u	117	75
v	118	76
w	119	77
x	120	78
y	121	79
z	122	7A
{	123	7B
		124	7C
}	125	7D
~	126	7E
DEL	127	7F

Char	Dec	Hex
Res	128	80
Res	129	81
Res	130	82
Res	131	83
IND	132	84
NEL	133	85
SSA	134	86
ESA	135	87
HTS	136	88
HTJ	137	89
VTS	138	8A
PLD	139	8B
PLU	140	8C
RI	141	8D
SS2	142	8E
SS3	143	8F
DCS	144	90
PU1	145	91
PU2	146	92
STS	147	93
CCH	148	94
MW	149	95
SPA	150	96
EPA	151	97
Res	152	98
Res	153	99
Res	154	9A
CSI	155	9B
ST	156	9C
OSC	157	9D
PM	158	9E
APC	159	9F

Char	Dec	Hex
nbsp	160	A0
¡	161	A1
¢	162	A2
£	163	A3
¤	164	A4
¥	165	A5
¦	166	A6
§	167	A7
¨	168	A8
©	169	A9
ª	170	AA
«	171	AB
¬	172	AC
shy	173	AD
®	174	AE
¯	175	AF
°	176	B0
±	177	B1
²	178	B2
³	179	B3
´	180	B4
µ	181	B5
¶	182	B6
·	183	B7
¸	184	B8
¹	185	B9
º	186	BA
»	187	BB
¼	188	BC
½	189	BD
¾	190	BE
¿	191	BF

Char	Dec	Hex
À	192	C0
Á	193	C1
Â	194	C2
Ã	195	C3
Ä	196	C4
Å	197	C5
Æ	198	C6
Ç	199	C7
È	200	C8
É	201	C9
Ê	202	CA
Ë	203	CB
Ì	204	CC
Í	205	CD
Î	206	CE
Ï	207	CF
Ð	208	D0
Ñ	209	D1
Ò	210	D2
Ó	211	D3
Ô	212	D4
Õ	213	D5
Ö	214	D6
×	215	D7
Ø	216	D8
Ù	217	D9
Ú	218	DA
Û	219	DB
Ü	220	DC
Ý	221	DD
Þ	222	DE
ß	223	DF

Char	Dec	Hex
à	224	E0
á	225	E1
â	226	E2
ã	227	E3
ä	228	E4
å	229	E5
æ	230	E6
ç	231	E7
è	232	E8
é	233	E9
ê	234	EA
ë	235	EB
ì	236	EC
í	237	ED
î	238	EE
ï	239	EF
ð	240	F0
ñ	241	F1
ò	242	F2
ó	243	F3
ô	244	F4
õ	245	F5
ö	246	F6
÷	247	F7
ø	248	F8
ù	249	F9
ú	250	FA
û	251	FB
ü	252	FC
ý	253	FD
þ	254	FE
ÿ	255	FF

The following tables assign a name (according to the ISO standard) to each printable character of the set.

Hex	Name
09	Horizontal Tab
0A	Line Feed
0C	Form Feed
0D	Carriage Return
20	Space
21	Exclamation Mark
22	Quotation Mark
23	Number Sign
24	Dollar Sign
25	Percent Sign
26	Ampersand
27	Apostrophe
28	Left Parenthesis
29	Right Parenthesis
2A	Asterisk
2B	Plus Sign
2C	Comma
2D	Hyphen, Minus Sign
2E	Full Stop
2F	Solidus
30	Digit Zero
31	Digit One
32	Digit Two
33	Digit Three
34	Digit Four
35	Digit Five
36	Digit Six
37	Digit Seven
38	Digit Eight
39	Digit Nine
3A	Colon
3B	Semicolon
3C	Less Than Sign

Hex	Name
3D	Equals Sign
3E	Greater Than Sign
3F	Question Mark
40	Commercial AT
41	Capital Letter A
42	Capital Letter B
43	Capital Letter C
44	Capital Letter D
45	Capital Letter E
46	Capital Letter F
47	Capital Letter G
48	Capital Letter H
49	Capital Letter I
4A	Capital Letter J
4B	Capital Letter K
4C	Capital Letter L
4D	Capital Letter M
4E	Capital Letter N
4F	Capital Letter O
50	Capital Letter P
51	Capital Letter Q
52	Capital Letter R
53	Capital Letter S
54	Capital Letter T
55	Capital Letter U
56	Capital Letter V
57	Capital Letter W
58	Capital Letter X
59	Capital Letter Y
5A	Capital Letter Z
5B	Left Square bracket
5C	Reverse Solidus
5D	Right Square Bracket

Hex	Name
5E	Circumflex Accent
5F	Low Line
60	Grave Accent
61	Small Letter a
62	Small Letter b
63	Small Letter c
64	Small Letter d
65	Small Letter e
66	Small Letter f
67	Small Letter g
68	Small Letter h
69	Small Letter i
6A	Small Letter j
6B	Small Letter k
6C	Small Letter l
6D	Small Letter m
6E	Small Letter n
6F	Small Letter o
70	Small Letter p
71	Small Letter q
72	Small Letter r
73	Small Letter s
74	Small Letter t
75	Small Letter u
76	Small Letter v
77	Small Letter w
78	Small Letter x
79	Small Letter y
7A	Small Letter z
7B	Left Curly Bracket
7C	Vertical Line
7D	Right Curly Bracket
7E	Tilde
A0	No-Break Space

Hex	Name
A1	Inverted Exclamation Mark
A2	Cent Sign
A3	Pound Sign
A4	Currency Sign
A5	Yen Sign
A6	Broken Bar
A7	Paragraph Sign, Section Sign
A8	Diaeresis
A9	Copyright Sign
AA	Feminine Ordinal Indicator
AB	Left Angle Quotation Mark
AC	Not Sign
AD	Soft Hyphen
AE	Registered Trade Mark Sign
AF	Macron
B0	Ring Above, Degree Sign
B1	Plus-Minus Sign
B2	Superscript Two
B3	Superscript Three
B4	Acute Accent
B5	Micro Sign
B6	Pilcrow Sign
B7	Middle Dot
B8	Cedilla
B9	Superscript One
BA	Masculine Ordinal Indicator
BB	Right Angle Quotation Mark
BC	Vulgar Fraction One Quarter
BD	Vulgar Fraction One Half
BE	Vulgar Fraction Three Quarters
BF	Inverted Question Mark
C0	Capital Latter A with Grave
C1	Capital Letter A with Acute Accent
C2	Capital Letter A with Circumflex

Hex	Name
C3	Capital Letter A with Tilde
C4	Capital Letter A with Diaeresis
C5	Capital Letter A with Ring Above
C6	Capital Diphthong A with E
C7	Capital Letter C with Cedilla
C8	Capital Letter E with Grave Accent
C9	Capital Letter E with Acute Accent
CA	Capital Letter E with Circumflex
CB	Capital Letter E with Diaeresis
CC	Capital Letter I with Grave Accent
CD	Capital Letter I with Acute Accent
CE	Capital Letter I with Circumflex
CF	Capital Letter I with Diaeresis
D0	Capital Icelandic Letter ETH
D1	Capital Letter N with Tilde
D2	Capital Letter O with Grave Accent
D3	Capital Letter O with Acute Accent
D4	Capital Letter O with Circumflex
D5	Capital Letter O with Tilde
D6	Capital Letter O with Diaeresis
D7	Multiplication Sign
D8	Capital Letter O with Oblique
D9	Capital Letter U with Grave Accent
DA	Capital Letter U with Acute Accent
DB	Capital Letter U with Circumflex
DC	Capital Letter U with Diaeresis
DD	Capital Letter Y with Acute Accent
DE	Capital Icelandic Letter THORN
DF	Small German Letter Sharp s
E0	Small Letter a with Grave Accent
E1	Small Letter a with Acute Accent

Hex	Name
E2	Small Letter a with Circumflex
E3	Small Letter a with Tilde
E4	Small Letter a with Diaeresis
E5	Small Letter a with Ring Above
E6	Small Diphthong a with e
E7	Small Letter c with Cedilla
E8	Small Letter e with Grave Accent
E9	Small Letter e with Acute Accent
EA	Small Letter e with Circumflex
EB	Small Letter e with Diaeresis
EC	Small Letter i with Grave Accent
ED	Small Letter i with Acute Accent
EE	Small Letter i with Circumflex
EF	Small Letter i with Diaeresis
F0	Small Icelandic Letter ETH
F1	Small Letter n with Tilde
F2	Small Letter o with Grave Accent
F3	Small Letter o with Acute Accent
F4	Small Letter o with Circumflex
F5	Small Letter o with Tilde
F6	Small Letter o with Diaeresis
F7	Division Sign
F8	Small Letter o with Oblique Stroke
F9	Small Letter u with Grave Accent
FA	Small Letter u with Acute Accent
FB	Small Letter u with Circumflex
FC	Small Letter u with Diaeresis
FD	Small Letter y with Acute Accent
FE	Small Icelandic Letter THORN
FF	Small Letter y with Diaeresis

The following tables assign an identifier to each character of the set. These identifiers are the constant names of each character.

Hex	Constant Name
00	NUL
01	SOH
02	STX
03	ETX
04	EOT
05	ENQ
06	ACK
07	BEL
08	BS
09	HT
0A	LF
0B	VT
0C	FF
0D	CR
0E	SO
0F	SI
10	DLE
11	DC1
12	DC2
13	DC3
14	DC4
15	NAK
16	SYN
17	ETB
18	CAN
19	EM
1A	SUB
1B	ESC
1C	FS or IS4
1D	GS or IS3
1E	RS or IS2
1F	US or IS1

Hex	Constant Name
20	Space
21	Exclamation
22	Quotation
23	Number_Sign
24	Dollar_Sign
25	Percent_Sign
26	Ampersand
27	Apostrophe
28	Left_Parenthesis
29	Right_Parenthesis
2A	Asterisk
2B	Plus_Sign
2C	Comma
2D	Hyphen or Minus_Sign
2E	Full_Stop
2F	Solidus
30	Digit_Zero
31	Digit_One
32	Digit_Two
33	Digit_Three
34	Digit_Four
35	Digit_Five
36	Digit_Six
37	Digit_Seven
38	Digit_Eight
39	Digit_Nine
3A	Colon
3B	Semicolon
3C	Less_Than_Sign
3D	Equals_Sign
3E	Greater_Than_Sign
3F	Question

Hex	Constant Name
40	Commercial_At
41	UC_A
42	UC_B
43	UC_C
44	UC_D
45	UC_E
46	UC_F
47	UC_G
48	UC_H
49	UC_I
4A	UC_J
4B	UC_K
4C	UC_L
4D	UC_M
4E	UC_N
4F	UC_O
50	UC_P
51	UC_Q
52	UC_R
53	UC_S
54	UC_T
55	UC_U
56	UC_V
57	UC_W
58	UC_X
59	UC_Y
5A	UC_Z
5B	Left_Square_Bracket
5C	Reverse_Solidus
5D	Right_Square_Bracket
5E	Circumflex
5F	Low_Line

Hex	Constant Name
60	Grave
61	LC_A
62	LC_B
63	LC_C
64	LC_D
65	LC_E
66	LC_F
67	LC_G
68	LC_H
69	LC_I
6A	LC_J
6B	LC_K
6C	LC_L
6D	LC_M
6E	LC_N
6F	LC_O
70	LC_P
71	LC_Q
72	LC_R
73	LC_S
74	LC_T
75	LC_U
76	LC_V
77	LC_W
78	LC_X
79	LC_Y
7A	LC_Z
7B	Left_Curly_Bracket
7C	Vertical_Line
7D	Right_Curly_Bracket
7E	Tilde
7F	DEL

Hex	Constant Name
80	Reserved_128
81	Reserved_129
82	BPH
83	NBH
84	Reserved_132
85	NEL
86	SSA
87	ESA
88	HTS
89	HTJ
8A	VTS
8B	PLD
8C	PLU
8D	RI
8E	SS2
8F	SS3
90	DCS
91	PU1
92	PU2
93	STS
94	CCH
95	MW
96	SPA
97	EPA
98	Res
99	Res
9A	Res
9B	CSI
9C	ST
9D	OSC
9E	PM
9F	APC

Hex	Constant Name
A0	No_Break_Space or NBSP
A1	Inverted_Exclamation
A2	Cent_Sign
A3	Pound_Sign
A4	Currency_Sign
A5	Yen_Sign
A6	Broken_Bar
A7	Section_Sign
A8	Diaeresis
A9	Copyright_Sign
AA	Feminine_Ordinal_Indicator
AB	Left_Angle_Quotation
AC	Not_Sign
AD	Soft_Hyphen
AE	Registered_Trade_Mark_Sign
AF	Macron
B0	Degree_Sign or Ring_Above
B1	Plus_Minus_Sign
B2	Superscript_Two
B3	Superscript_Three
B4	Acute
B5	Micro_Sign
B6	Pilcrow_Sign or Paragraph_Sign
B7	Middle_Dot
B8	Cedilla
B9	Superscript_One
BA	Masculine_Ordinal_Indicator
BB	Right_Angle_Quotation
BC	Fraction_One_Quarter
BD	Fraction_One_Half
BE	Fraction_Three_Quarters
BF	Inverted_Question

Hex	Constant Name
C0	UC_A_Grave
C1	UC_A_Acute
C2	UC_A_Circumflex
C3	UC_A_Tilde
C4	UC_A_Diaeresis
C5	UC_A_Ring
C6	UC_AE_Diphthong
C7	UC_C_Cedilla
C8	UC_E_Grave
C9	UC_E_Acute
CA	UC_E_Circumflex
CB	UC_E_Diaeresis
CC	UC_I_Grave
CD	UC_I_Acute
CE	UC_I_Circumflex
CF	UC_I_Diaeresis
D0	UC_Icelandic_Eth
D1	UC_N_Tilde
D2	UC_O_Grave
D3	UC_O_Acute
D4	UC_O_Circumflex
D5	UC_O_Tilde
D6	UC_O_Diaeresis
D7	Multiplication_Sign
D8	UC_O_Oblique_Stroke
D9	UC_U_Grave
DA	UC_U_Acute
DB	UC_U_Circumflex
DC	UC_U_Diaeresis
DD	UC_Y_Acute
DE	UC_Icelandic_Thorn
DF	LC_German_Sharp_S

Hex	Constant Name
E0	LC_A_Grave
E1	LC_A_Acute
E2	LC_A_Circumflex
E3	LC_A_Tilde
E4	LC_A_Diaeresis
E5	LC_A_Ring
E6	LC_AE_Diphthong
E7	LC_C_Cedilla
E8	LC_E_Grave
E9	LC_E_Acute
EA	LC_E_Circumflex
EB	LC_E_Diaeresis
EC	LC_I_Grave
ED	LC_I_Acute
EE	LC_I_Circumflex
EF	LC_I_Diaeresis
F0	LC_Icelandic_Eth
F1	LC_N_Tilde
F2	LC_O_Grave
F3	LC_O_Acute
F4	LC_O_Circumflex
F5	LC_O_Tilde
F6	LC_O_Diaeresis
F7	Division_Sign
F8	LC_O_Oblique_Stroke
F9	LC_U_Grave
FA	LC_U_Acute
FB	LC_U_Circumflex
FC	LC_U_Diaeresis
FD	LC_Y_Acute
FE	LC_Icelandic_Thorn
FF	LC_Y_Diaeresis

ANNEX C

EAST FORMAL SYNTAX SPECIFICATION

(This annex **is not** part of the Recommendation)

This annex describes the EAST syntax using a simple version of the Backus-Naur-Form. See below the lexical rules, common to the whole syntax specification:

C1 COMMON LEXICAL RULES

<underline> ::= _

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

<upper_case> ::= A | B |...| Z

<lower_case> ::= a | b |...| z

<letter> ::= <upper_case> | <lower_case>

<letter_or_digit> ::= <letter> | <digit>

<identifier> ::= <letter> { [<underline>] <letter_or_digit> }

<integer_literal> ::= <digit> { [<underline>] <digit> }

<exponent> ::= E [+ | -] <integer_literal>

<floating_point_literal> ::= <integer_literal>.<integer_literal> [<exponent>]

<identifier_list> ::= <identifier> { , <identifier> }

<character> ::= *nul* | ... | 0 | 1 | ... | b | ... | ~ | *del*

<character_literal> ::= ‘ <character> ’

<based_literal> ::= <base># <based_integer> [. <based_integer>] # [<exponent>]

<base> ::= 2 | 8 | 16

<based_integer> ::= <extended_digit> {[<underline>] <extended_digit>}

<extended_digit> ::= digit | A | B | C | D | E | F | a | b | c | d | e | f

<integer_based_literal> ::= <base># <based_integer> # [<exponent>]

<real_based_literal> ::= ::= <base># <based_integer> . <based_integer> # [<exponent>]

<simple_integer_based_literal > ::= <base># <based_integer> #

C2 DECLARATION OF THE LOGICAL DATA DESCRIPTION RECORD

<Logical Description> ::= **package** <identifier> **is**
{ <constant_declaration> |
<type_declaration> | <subtype_declaration> |
<representation_clause> }
{ <variable_declaration> | <constant_declaration> }
end <identifier> **;**

C2.1 SUBTYPE DECLARATION

<subtype_declaration> ::= **subtype** <*subtype*_identifier> **is** <subtype_indication> **;**

<subtype_indication> ::= <type_mark> [<constraint>]

<type_mark> ::= <predefined_type> | <*subtype*_identifier> |
<*type*_identifier>

C2.2 TYPE DECLARATION

<type_declaration> ::= **type** <*type*_identifier> [<discriminant_part>] **is**
<type_definition> **;**

<discriminant_part> ::= (<discriminant_specification>
{ **;** <discriminant_specification> })

<discriminant_specification>::= <identifier> **:** <*discrete*_type_mark> :=
<discriminant_value>

<discriminant_value> ::= <integer_literal> | <enumeration_literal>

<type_definition> ::= <integer_type_definition> | <real_type_definition> |
<array_type_definition> | <record_type_definition> |
<enumeration_type_definition>

a) constraint declaration

<constraint> ::= <range_constraint> | <floating_point_constraint> |
<index_constraint>

<range_constraint> ::= **range** <range>

<range> ::= <integer_bound> **..** <integer_bound> **;**

<integer_bound> ::= [+ | -] <integer_literal> | <*integer_constant*_identifier>

<floating_point_constraint> ::= [**digits** <*positive*_integer_literal>]
[**range** <real_bound> **..** <real_bound>] **;**

<real_bound> ::= [+ | **-**] <floating_point_literal> | <*real_i*dentifier>

<index_constraint> ::= **(** <discrete_range> {**,** <discrete_range> } **)**

b) integer type declaration

<integer_type_definition> ::= <range_constraint>

c) real type declaration

<real_type_definition> ::= <floating_point_constraint>

d) array type declaration

<array_type_definition> ::= <unconstrained_array_definition> |
<constrained_array_definition>

<unconstrained_array_definition> ::= **array (** <index_subtype_definition>
{**,** <index_subtype_definition> } **) of** <type_mark> **;**

<constrained_array_definition> ::= **array (** <discrete_range>
{**,** <discrete_range> } **) of** <type_mark> **;**

<index_subtype_definition> ::= <discrete_type_mark> **range <>**

<discrete_range> ::= <discrete_type_mark> | <constrained_index>

<discrete_type_mark> ::= <*integer*_type_mark> | <*enumeration*_type_mark>

<constrained_index> ::= <index_bound> .. <index_bound>

<index_bound> ::= [+|-] <integer_literal> | <*integer_constant*_identifier> |
<enumeration_literal>

e) record type declaration

<record_type_definition> ::= **record** <component_list> **end record ;**

<component_list> ::= <component_declaration> { <component_declaration> }|
{ <component_declaration> } <variant_part> |
null ;

<component_declaration> ::= <identifier> **:** <subtype_indication> **;**

<variant_part> ::= **case** <*discriminant*_identifier> **is**
<variant> { <variant> }
end case ;

<variant> ::= **when** <choice> { **|** <choice> } **=>**
<component_list>

<choice> ::= [-] <integer_literal> | <discrete_range> | **others** |
<enumeration_literal_specifiation>

f) enumeration type declaration

<enumeration_type_definition> ::=**(** <enumeration_literal> { **,** <enumeration_literal> } **)**

<enumeration_literal>::= <identifier> | <character_literal>

g) predefined types

<predefined_type> ::= **character** | **string | EOF**

C2.3 OBJECT DECLARATION

<constant_declaration>::= <identifier> : **constant** [<type_mark>]
[:= <constant_value>]**;**

<constant_value> ::= <integer_literal> |
<floating_point_literal> |
<enumeration_literal> |
<string_literal> |
<static_expression> ;

<static_expression> ::= { (<term>) | <term> }

<term> ::= [unary_operator] <number> [<binary_operator><number>]

<unary_operator> ::= + | -

<binary_operator> ::= + | - | * | / | **

-- See 3.2.3.2 of the present Recommendation for further constraints on operators.

<number> ::= <integer_literal> | <floating_point_literal> |
<*integer_constant*_identifier> | <*real_constant*_identifier>

<variable_declaration>::= <identifier> : <type_mark>;

C2.4 REPRESENTATION CLAUSE

<representation_clause> ::= <length_clause> |
<enumeration_representation_clause> |
<record_representation_clause>

a) length clause

<length_clause>::= **for** <*type*_identifier>**'size use** <integer_literal> **;**

b) enumeration representation clause

<enumeration_representation_clause>::= **for** <*enumeration_type*_identifier> **use**
<aggregate>**;**

<aggregate> ::=**(**<component_association> {**,** <component_association>} **)**

<component_association> ::= <enumeration_literal> **=>** <bit_pattern>

<bit_pattern> ::= <integer_literal> | <simple_integer_based_literal >

c) record representation clause

<record_representation_clause> ::= **for** <*record_type*_identifier> **use**
record
<component_clause> { <component_clause>}
end record ;

<component_clause> ::= <*component*_identifier> **at** <distance>
range <*static*_range> **;**

<distance> ::= <word_number> * **word_32_bits |**
<word_number> * **word_16_bits |** 0

<word_number> ::= <integer_literal>

C3 DECLARATION OF THE PHYSICAL DATA DESCRIPTION RECORD

<Physical Description> ::= **package** <identifier> **is**
<declaration_part>
end <identifier> **;**

<declaration_part> ::= { <constant_declaration> |
<type_declaration> | <subtype_declaration> }

<constant_declaration>::= <identifier> : **constant** <type_mark> := <constant_value> **;**

<constant_value> ::= <integer_literal> | <floating_point_literal> |
<enumeration_literal> |
<array_value> | <record_value>

<record_value> ::= **(** <record_component_value> { **,** <record_component_value> } **)**

<record_component_value> ::= <*component*_identifier> => <constant_value>

<array_value> ::= **(** <array_component_value> { **,** <array_component_value> } **)**

<array_component_value> ::= <integer_literal> => <constant_value>

Other BNF rules needed for the physical data description record, if not defined above in this section, are identical to those specified for the logical data description record.

ANNEX D

MAIN DIFFERENCES BETWEEN ADA AND EAST

(This annex **is not** part of the Recommendation)

This annex describes the main differences between EAST and the Ada programming language:

– the Ada features not retained in EAST;

– the Ada syntax elements which have another semantic in EAST;

– the EAST syntax restrictions vs. Ada.

D1 ADA FEATURES NOT RETAINED IN EAST

No algorithmic features of the Ada programming language have been retained in EAST.

In Ada, a program unit can be a procedure, a function, a package or a task. Only packages are allowed in EAST: a package structure is used to implement the logical description part; another package implements the physical description part.

Within the declarative part, some Ada **predefined types** have been **excluded** for the data description:

– The predefined type BOOLEAN, because no enumeration representation clause is provided for this type in the Ada STANDARD package. Any bit pattern may therefore be used for the implementation of a Boolean value.

– The predefined type INTEGER, because its definition depends on the implementation (size, bounds, etc.). For the same reason, other integer types such as LONG_INTEGER or SHORT_INTEGER are also forbidden.

– The subtypes of INTEGER: POSITIVE and NATURAL depend on the definition of INTEGER and so depend on the implementation.

– The predefined type FLOAT, because its definition depends on the implementation (size, number of digits, etc.). For the same reason, other floating-point types such as LONG_FLOAT or SHORT_FLOAT are also forbidden.

– The predefined type DURATION and any fixed-point type, because their size depends on the implementation. Consequence: a real type in EAST is always considered to be a floating-point type.

Access types and derived types are not considered to be useful in a Data Description Record.

In the same way, generics have not been retained in EAST.

In Ada, a pragma is used to convey information to Ada compilers. As such, pragma use is not justified in EAST.

D2 ADA SYNTAX ELEMENTS THAT HAVE A DIFFERENT MEANING IN EAST

A length clause is defined by the following declaration:

for type_identifier'**size use** static_expression **;**

In Ada, the value of the expression specifies an upper bound for the number of bits to be allocated to objects of the given type. In EAST, the expression specifies the exact number of bits that any object of the given type occupies.

In Ada, a record representation clause specifies the storage representation of records in memory, that is, the order, position, and size of record components in memory of a given machine. In EAST, the record representation clause specifies the actual storage representation on the medium.

D3 EAST SYNTAX RESTRICTIONS VS. ADA

In Ada, the base for based numeric literals can be any number between 2 and 16. In EAST the base can only be 2, 8 or 16.

In Ada the values specified in a range constraint within an integer or real type definition can be a simple_expression. In EAST the values may only be a numeric literal or an identifier (naming an appropriate numeric constant or an appropriate discriminant eventually computed later, as described in 3.2.1.6).

In Ada, a constant declaration allows a list of identifiers. EAST allows only a single identifier.

ANNEX E

INFORMATIVE REFERENCES

(This annex **is not** part of the Recommendation)

Informative references [E2]-[E5] below contain information showing how EAST descriptions can be used in the SFDU standard or presenting the Ada language, which is the basis of this Recommendation.

[E1] *Procedures Manual for the Consultative Committee for Space Data Systems*. CCSDS A00.0-Y-9. Yellow Book. Issue 9. Washington, D.C.: CCSDS, November 2003.

[E2] *Standard Formatted Data Units—Structure and Construction Rules*. Recommendation for Space Data System Standards, CCSDS 620.0-B-2. Blue Book. Issue 2. Washington, D.C.: CCSDS, May 1992.

[E3] *Information Technology—Programming Languages—Ada*. International Standard, ISO/EIC 8652:1995. Geneva: ISO, 1995.

[E4] *The Data Description Language EAST—A Tutorial*. Report Concerning Space Data System Standards, CCSDS 645.0-G-1. Green Book. Issue 1, May 1997.

[E5] *The Data Description Language EAST—List of Conventions*. Report Concerning Space Data System Standards, CCSDS 646.0-G-1. Green Book. Issue 1, May 1997.

INDEX

Array storage method, 3-48

Array type, **3-12**, 3-13, 3-14, A-1

ASCII Representation, 3-58, 3-59, 3-61, 3-62, 3-67

Based literal, 3-4, 3-5, A-1

Binary Representation, 3-50, 3-55, 3-56, 3-57

BNF, 1-2, 1-4, A-1, C-6

Character literal, A-1

Character type, A-1

Comment, 3-1

Composite type, 2-3, A-1

Constant, 3-7, 3-8, 3-10, 3-11, 3-29, 3-30, 3-31, **3-32**, 3-33, 3-34, 3-46, 3-48, 3-50, 3-51, 3-52, 3-55, 3-56, 3-57, 3-58, 3-60, 3-62, 3-64, 3-66, A-1, A-2, B-8, B-9, B-10, B-11, C-2, C-4, C-6, D-2

Constraint array type, 3-13

Decimal literal, 3-2, 3-3, 3-4

Delimiter, 1-3, 1-4, 3-1, A-2

Discrete type, A-2

Discriminant, 3-18, 3-19, 3-20, 3-21, 3-22, 3-38, 3-42, A-2, A-3, C-2, C-4

Distance, 3-38, 3-45, 3-46, C-5

Enumeration constraint, 3-28

Enumeration representation clause, **3-36**, A-2

Enumeration type, **3-9**, 3-10, 3-58, 3-59, 3-60, A-2

Identifier, 1-1, 1-3, 2-3, 3-1, **3-2**, 3-7, 3-9, 3-10, 3-11, 3-13, 3-17, 3-21, 3-29, 3-30, 3-31, 3-32, 3-47, 3-51, 3-58, 3-62, 4-1, A-1, A-2, A-3, B-8, C-1, C-2, C-3, C-4, C-5, C-6, D-2

Index, 2-1, 2-3, 3-12, 3-13, 3-14, 3-17, 3-48, 3-55, C-2, C-3

Integer constraint, 3-29

Integer type, 2-3, 3-8, **3-10**, 3-12, 3-14, 3-18, 3-20, 3-21, 3-30, 3-33, 3-40, 3-56, 3-58, 3-62, 3-66, 3-67, A-2, C-3, D-1

Length clause, **3-35**, 3-36, A-2

Logical description, 2-2, 3-24, C-2

Marker, **3-33**, 3-34, A-2

Numeric literal, A-2

Numeric type, 2-3

Object, 1-3, 2-2, 2-3, 3-7, **3-31**, 3-32, 3-48, A-2, C-4, D-2

Package, 2-1, 2-2, 3-7, 3-8, 3-47, 3-64, C-2, C-6, D-1

Physical description, 2-2, 3-48, 3-50, 3-52, 3-53, 3-59, 3-60, 3-61, 3-62, C-6

Predefined type, **3-8**, A-2

Real constraint, 3-30

Real type, **3-11**, 3-62

Record representation clause, **3-38**, 3-39, 3-41, 3-42, 3-43, 3-45, 3-46, A-2

Record type, **3-14**, 3-15, 3-18, 3-20, A-2

Representation clause, 2-3, **3-35**, 3-36, 3-38, 3-39, 3-41, 3-42, 3-43, 3-45, 3-46, A-2, C-5

Scalar type, 3-27, 3-58, A-2

Separator, 3-1

Subtype, 2-3, 3-7, 3-8, **3-28**, 3-29, 3-30, 3-33, A-3, C-2, C-3, C-6

Unconstrained array type, 3-14

Variable, 1-4, 3-8, 3-31, 3-32, 3-39, A-2, A-3, C-2, C-4

Variant, 1-2, 1-4, 3-19, 3-38, 3-44, 3-45, A-3, C-3

Virtual discriminant, 3-21, 3-22, 3-24, 3-38, A-3

WORD_16_BITS, 3-46, 4-1, C-5

WORD_32_BITS, 3-46, 4-1, C-5