International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.104

Amendment 1

(10/2012)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

Formal description techniques (FDT) – Specification and Description Language (SDL)

Specification and Description Language: Data and action language in SDL-2010

Amendment 1: Replacement Annex C on language binding

Recommendation ITU-T Z.104 (2011) – Amendment 1

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	**Z.100–Z.109**
Application of formal description techniques	Z.110–Z.119
Message Sequence Chart (MSC)	Z.120–Z.129
User Requirements Notation (URN)	Z.150–Z.159
Testing and Test Control Notation (TTCN)	Z.160–Z.179
PROGRAMMING LANGUAGES	
CHILL: The ITU-T high level language	Z.200–Z.209
MAN-MACHINE LANGUAGE	
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.349
Data-oriented human-machine interfaces	Z.350–Z.359
Human-machine interfaces for the management of telecommunications networks	Z.360–Z.379
QUALITY	
Quality of telecommunication software	Z.400–Z.409
Quality aspects of protocol-related Recommendations	Z.450–Z.459
METHODS	
Methods for validation and testing	Z.500–Z.519
MIDDLEWARE	
Processing environment architectures	Z.600–Z.609

For further details, please refer to the list of ITU-T Recommendations.

Recommendation ITU-T Z.104

Specification and Description Language: Data and action language in SDL-2010

Amendment 1

Replacement Annex C on language binding

Summary

Amendment 1 updates Recommendation ITU-T Z.104 to incorporate binding to C language syntax as an alternative syntax for expressions and statements.

Amendment 1 replaces Annex C of ITU-T Z.104 (2011).

History

Edition	Recommendation	Approval	Study Group
1.0	ITU-T Z.104	2004-10-07	17
2.0	ITU-T Z.104	2011-12-22	17
2.1	ITU-T Z.104 (2011) Amd. 1	2012-10-14	17

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at http://www.itu.int/ITU-T/ipr/.

Table of Contents

Recommendation ITU-T Z.104

Specification and Description Language: Data and action language in SDL-2010

Amendment 1

Replacement Annex C on language binding

Replace ITU-T Z.104 (2011) Annex C with Annex C as defined in this amendment and add the Bibliography specified at the end.

Annex C

Language binding

(This annex forms an integral part of this Recommendation.)

This annex defines the use of the syntax of an alternative language within the syntax rules <variable definition>, <data definition>, <non terminating statement>, <terminating statement> and <expression>, which are taken as the points of syntax variation. By default an SDL-2010 specification is bound to the native syntax as defined in the main body of this Recommendation or other Recommendations for SDL-2010.

Each diagram or other definition that is allowed a <package use clause> is bound to a particular concrete syntax as defined in clause 7.2. The only allowed language <data binding> constructs are to the default binding **`package`** `Predefined`, which is the language binding for the native SDL-2010 concrete syntax, or to a package defined in this annex.

Though the syntax within the points of syntax variation looks like another language (such as C, C++, Java or some other language) the semantics are defined by SDL-2010. The binding to the SDL-2010 abstract grammar is permitted to invoke complex transformations to achieve the mapping, and constraints are allowed to exclude constructions permitted by the language concrete syntax that are not able to be reasonably mapped. A model that includes constructs that do not map to SDL-2010 does not conform to the SDL-2010 language. A tool that handles constructs that do not map to SDL-2010 abstract grammar and semantics shall provide an indication if such constructs are used.

Different languages not only vary in their syntax rules, but also have some differences in the lexical elements of the language. Therefore, within diagrams bound to the syntax of an alternative language for certain rules, it is allowed for the lexical rules of SDL-2010 to be extended to include lexemes of the alternative language. This annex also defines these variations.

To be concise, throughout this annex the phrase "SDL-2010 diagram" is used to mean a "diagram bound to the native SDL-2010 syntax for the points of syntax variation", and the phrase "SDL-2010 diagrams" to mean more than one such diagram.

C.1 C Language binding

This clause allows diagrams to be bound to a subset of [b-ISO/IEC 9899] by binding each such diagram to the **`package`** `C_Predefined` defined at the end of this clause.

To be concise, throughout this clause the phrase "C diagram" is used to mean a "diagram bound to a subset of [b-ISO/IEC 9899] for the points of syntax variation", and the phrase "C diagrams" to mean more than one such diagram.

C.1.1 Extensions to lexical rules

Lexical rules define lexical units. Lexical units are terminal symbols of the *Concrete grammar*.

<lexical unit> ::=
<name>
| <integer name>
| <real name>
| <character string>
| <hex string>
| <bit string>
| <note>
| <comment body>
| <composite special>
| <special>
| <semicolon>
| <other character>
| <quoted operation name>
| <c string literal>
| <keyword>

The rule <lexical unit> is extended to include <c string literal>.

NOTE – No qualifier is allowed for an identifier in the C language syntax, therefore identifiers and names have the same syntax. The universal character name (of the form **\u** hex-quad or **\U** hex-quad hex-quad) are not allowed, therefore names in the C language have the same syntax as names in SDL-2010.

C.1.1.1 C keywords

The following SDL-2010 lowercase <name> items are also keywords in C diagrams:

auto	**case**	**char**
const	**do**	**double**
enum	**extern**	**float**
for	**goto**	**inline**
int	**long**	**register**
restrict	**short**	**signed**
sizeof	**static**	**switch**
typedef	**union**	**unsigned**
void	**volatile**	**while**

These are keywords only in C diagrams. In SDL-2010 diagrams each of these is treated as a <name>. In accordance with C and to limit the impact on SDL-2010 of C lexical rules, corresponding uppercase keywords do not exist. An SDL-2010 <keyword> item as defined in [ITU-T Z.101] cannot be used as a <name> in C type or variable definitions.

C.1.1.2 C integer constants

<integer name> ::=
<decimal digit>+
| <c integer constant>

The lexical rule <integer name> is extended to include <c integer constant>. The form <decimal digit>+ applies to SDL-2010 diagrams, and the form <c integer constant> applies to C diagrams.

<c integer constant> ::=
<c decimal constant>
| <c octal constant>
| <c hexadecimal constant>

NOTE 1 – Clause 6.4.4.1 of [b-ISO/IEC 9899] but excluding the <c integer suffix>.

<c decimal constant> ::=
<c nonzero digit>
| <c decimal constant> <c digit>

NOTE 2 – Clause 6.4.4.1 of [b-ISO/IEC 9899].

<c octal constant> ::=
0
| <c octal constant> <c octal digit>

NOTE 3 – Clause 6.4.4.1 of [b-ISO/IEC 9899].

<c hexadecimal constant> ::=
<c hexadecimal prefix> <c hexadecimal digit>
| <c hexadecimal constant> <c hexadecimal digit>

NOTE 4 – Clause 6.4.4.1 of [b-ISO/IEC 9899].

<c hexadecimal prefix> ::=
0x

NOTE 5 – Clause 6.4.4.1 of [b-ISO/IEC 9899] without the **0X** form.

<c nonzero digit> ::=
1 | **2** | **3** | **4** | **5**
| **6** | **7** | **8** | **9**

NOTE 6 – Clause 6.4.4.1 of [b-ISO/IEC 9899].

<c octal digit> ::=
0 | **1** | **2** | **3** | **4** | **5** | **6** | **7**

NOTE 7 – Clause 6.4.4.1 of [b-ISO/IEC 9899].

<c hexadecimal digit> ::=
0 | **1** | **2** | **3** | **4**
| **5** | **6** | **7** | **8** | **9**
| **a** | **b** | **c** | **d** | **e** | **f**
| **A** | **B** | **C** | **D** | **E** | **F**

NOTE 8 – Clause 6.4.4.1 of [b-ISO/IEC 9899].

A <c integer constant> has an unbounded integer value determined according to the rules given in [b-ISO/IEC 9899]. The *Name* of the <c integer constant> is the *Name* of the <decimal digit> string starting with a non-zero digit that has this value. The *Qualifier* for the *Identifier* of a <c integer constant> is the *Qualifier for* `<<package Predefined type Integer>>`.

C.1.1.3 C composite specials

<composite special> ::=
<result sign>
| <range sign>
| <composite begin sign>
| <composite end sign>
| <concatenation sign>
| <history dash sign>
| <greater than or equals sign>
| <implies sign>
| <is assigned sign>
| <less than or equals sign>
| <not equals sign>
| <qualifier begin sign>
| <qualifier end sign>
| <c increment operator>

| <c decrement operator>
| <c equality sign>
| <c inequality sign>
| <c logical and sign>
| <c logical or sign>
| <c multiplication assignment sign>
| <c division assignment sign>
| <c remainder assignment sign>
| <c addition assignment sign>
| <c subtraction assignment sign>
| <c shift left assignment sign>
| <c shift right assignment sign>
| <c bitwise and assignment sign>
| <c bitwise excl or assignment sign>
| <c bitwise incl or assignment sign>

The rule <composite special> is extended to include the additional composite symbols used in C except those symbols only used for pre-processing and alternatives for: square brackets, curly brackets and the number sign.

<c pointer operator> ::=
<result sign>

<c shift left sign> ::=
<qualifier begin sign>

<c shift right sign> ::=
<qualifier end sign>

The rule <c pointer operator>, <c shift left sign> and <c shift right sign> lexical units are the same as <result sign>, <qualifier begin sign> and <qualifier end sign>, respectively, and the lexical unit represented is determined by context. In syntax rules defined for C diagrams these lexical units always represent <c pointer operator>, <c shift left sign> and <c shift right sign>, respectively.

<c increment operator> ::=
<plus sign> <plus sign>

<c decrement operator> ::=
<hyphen> <hyphen>

<c equality sign> ::=
<equals sign> <equals sign>

<c inequality sign> ::=
<exclamation mark> <equals sign>

<c logical and sign> ::=
<ampersand> <ampersand>

<c logical or sign> ::=
<vertical line> <vertical line>

<c multiplication assignment sign> ::=
<asterisk> <equals sign>

<c division assignment sign> ::=
<solidus> <equals sign>

<c remainder assignment sign> ::=
<percent sign> <equals sign>

<c addition assignment sign> ::=
<plus sign> <equals sign>

<c subtraction assignment sign> ::=
<hyphen> <equals sign>

<c shift left assignment sign> ::=
<less than sign> <less than sign> <equals sign>

<c shift right assignment sign> ::=
<greater than sign> <greater than sign> <equals sign>

<c bitwise and assignment sign> ::=
<ampersand> <equals sign>

<c bitwise excl or assignment sign> ::=
<circumflex accent> <equals sign>

<c bitwise incl or assignment sign> ::=
<vertical line> <equals sign>

C.1.1.4 C character constants

<c character constant> ::=
<apostrophe> <c char sequence> <apostrophe>

NOTE 1 – Clause 6.4.4.4 of [b-ISO/IEC 9899] without the prefixes for types other than **unsigned char**.

<c char sequence> ::=
<c char>

NOTE 2 – Clause 6.4.4.4 of [b-ISO/IEC 9899] restricted to a single character, because the "value of an integer character constant containing more than one character, or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined". (See clause 6.4.4.4 of [b-ISO/IEC 9899].)

<c char> ::=
<quotation mark> | <c other character> | <space> | <c escape sequence>

NOTE 3 – Clause 6.4.4.4 of [b-ISO/IEC 9899], modified to use SDL-2010 lexical items.

A <c escape sequence> (see clause C.1.1.5 below) shall map to a single byte character.

The <c character constant> represents the constant **unsigned char** value corresponding to the constant expression `to_Unsigned_char(<<package Predefined type Char>> num(c))`, where c is the character specified by the <c character constant>.

C.1.1.5 C string literals

<c string literal> ::=
<quotation mark> [<c s char sequence>] <quotation mark>

There is a clash between <quoted operation name> and <c string literal>. A lexical unit in an SDL-2010 diagram that starts with a <quotation mark> never represents a <c string literal>. A lexical unit in a C diagram that starts with a <quotation mark> always represents a <c string literal>.

A <c string literal> is pointer to the first element of an array of characters derived from the <character string> where the <c string literal> has a <c s char sequence> the same as the sequence of characters in a <character string> (after replacement of <c escape character> items in the <c s char sequence> and <apostrophe> pairs in the <character string>).

A <c string literal> represents a *Variable-access* for an implicit anonymous **unsigned char** pointer (to first **char** of an implicit **char** array) defined by

```
synonym anoncp Star_Unsigned_char = "&"(xstring[0]);
```

where

`xstring` is an implicit anonymous **char** array defined by

```
synonym xstring value inherits Cvector <char,n+1> = c_array_init;
```

where

`n` is the length of the <character string> (that is, the length in bytes of the <c s char sequence> after replacement of <c escape character> items), and

`c_array_init` is the initialization value using a call of `Make` and calls of `Modify` (as described for <c initialize> in clause C.1.3) with each element of the array initialized to the corresponding character of the string and the string terminates in a zero (a `NUL` character).

For example, if the <c string literal> is `"AB3"`, then `c_array_init` is `{'A', 'B', '3', endstr}` where `endstr` is `Integer_to_Unsigned_char(num(<<`**`package`** `Predefined` **`type`** `Character>>NUL))`.

The **unsigned char** array element `i` (<`n`) for the string is `Integer_to_Unsigned_char(num(zstring[i+1]))`, where `zstring` is the `Charstring` <character string>.

NOTE 1 – This definition allows a <c string literal> to be used with functions from the C library `string.h` (see clause 7.24 of [b-ISO/IEC 9899]), if these functions are defined in an additional package as external operations or procedures. The operator

```
<<package C_Predefined type Star_Unsigned_char>> Star_Unsigned_char_to_Charstring
```

allows a <c string literal> (or other `Unsigned_char` array values) to be converted to `Charstring` expressions.

<c s char sequence> ::=
<c s char>
| <c s char sequence> <c s char>

NOTE 2 – Clause 6.4.5 of [b-ISO/IEC 9899].

<c s char> ::=
<apostrophe> | <c other character> | <space> | <c escape sequence>

NOTE 3 – Clause 6.4.5 of [b-ISO/IEC 9899], modified to use SDL-2010 lexical items.

<c other character> ::=
<alphanumeric>
| <special> | <dollar sign> | <percent sign>
| <ampersand> | <question mark> | <commercial at>
| <circumflex accent> | <underline> | <grave accent>
| <vertical line> | <tilde>

<c escape sequence> ::=
<c simple escape sequence>
| <c octal escape sequence>
| <c hexadecimal escape sequence>

NOTE 4 – Clause 6.4.4.4 of [b-ISO/IEC 9899] excluding <c universal character name>.

<c simple escape sequence> ::=
<reverse solidus> <apostrophe>
| <reverse solidus> <quotation mark>
| <reverse solidus> <question mark>
| <reverse solidus> <reverse solidus>
| <reverse solidus> **a**
| <reverse solidus> **b**
| <reverse solidus> **f**
| <reverse solidus> **n**
| <reverse solidus> **r**
| <reverse solidus> **t**
| <reverse solidus> **v**

NOTE 5 – Clause 6.4.4.4 of [b-ISO/IEC 9899].

<c octal escape sequence> ::=
<reverse solidus> <c octal digit> <c octal digit> <c octal digit>

NOTE 6 – Clause 6.4.4.4 of [b-ISO/IEC 9899].

<c hexadecimal escape sequence> ::=
<reverse solidus> **x** <c hexadecimal digit> <c hexadecimal digit>

NOTE 7 – Clause 6.4.4.4 of [b-ISO/IEC 9899] but always with two hexadecimal digits.

The characters represented by the alternatives of <c escape sequence> are defined by [b-ISO/IEC 9899].

C.1.1.6 C identifiers, name resolution and visibility

Concrete grammar

<c identifier> ::=
<name>

NOTE – Clause 6.4.2.1 of [b-ISO/IEC 9899] is redefined as <name> because <c identifier> has the same syntax as <name>.

In a C diagram, a <c identifier> cannot have a <qualifier>. Identifiers in C language data definitions and expressions are in principle bound to names according to the SDL-2010 name binding rules in clause 6.6 of [ITU-T Z.101]. In a defining context for a *Name* in the abstract grammar, a <c identifier> represents the *Name*. In a non-defining context where a name is used, a <c identifier> represents an *Identifier* and the *Qualifier* is derived from the name binding. A <c identifier> that represents an *Identifier* may be marked as belonging to a subcategory: for example, <data type c identifier> requires the *Identifier* to represent a data type. When a <c identifier> represents an *Identifier*, if it cannot be uniquely bound to an *Identifier* it is ambiguous and therefore invalid.

C.1.1.7 Macros and pre-processing

It is assumed that pre-processing as defined by [b-ISO/IEC 9899] is applied before any C diagram is considered as part of an SDL-2010 model, and as a consequence that any C pre-processing tokens or alternative composite symbols (such as the alternative left curly bracket) are removed from the contents of a C diagram. It is assumed that SDL-2010 macro processing is applied to the model after any C pre-processing.

C.1.2 Data type definition

Concrete grammar

<data definition> ::=
<entity in data type>
| <interface definition>
| <c type definition>

A <c type definition> shall only be used in a C diagram.

<c type definition> ::=
typedef <c type specifier> <c declarator> <semicolon>
| <c struct or union specifier> <semicolon>
| <c enum specifier> <semicolon>

NOTE 1 – This is a valid subset of <c declaration> from clause 6.7 of [b-ISO/IEC 9899]. This subset includes only alternatives starting with the keywords **typedef**, **struct**, **union** and **enum**. The <c init declarator list> is simplified to a <c declarator> and is not allowed to be omitted.

<c declarator> ::=
[<c pointer>] <c direct declarator>

NOTE 2 – Clause 6.7.6 of [b-ISO/IEC 9899].

If the <c type qualifier list> of a <c pointer> of a <c declarator> contains a <c type qualifier> that is **const**, it is allowed to modify the contents of the target of the pointer, but the pointer itself should always point to the same target.

The <c identifier> of a <c declarator> is the <c identifier> of the <c direct declarator> of the <c declarator>, or if the <c direct declarator> of the <c declarator> contains a <c declarator> enclosed in <left parenthesis> <right parenthesis>, the <c identifier> of this enclosed <c declarator>.

<c direct declarator> ::=
<c identifier>
{ <left square bracket> <integer c constant expression> <right square bracket> }*
| <left parenthesis> <c declarator> <right parenthesis>

NOTE 3 – Clause 6.7.6 of [b-ISO/IEC 9899] simplified so that it either starts with a <c identifier> or is a <c declarator> in parentheses, with the assignment expression constrained to a <c constant expression> without type qualifiers or **static**.

<c pointer> ::=
<asterisk> [<c type qualifier list>]
| <asterisk> [<c type qualifier list>] <c pointer>

NOTE 4 – Clause 6.7.6 of [b-ISO/IEC 9899].

<c type qualifier list> ::=
<c type qualifier>
| <c type qualifier list> <c type qualifier>

NOTE 5 – Clause 6.7.6 of [b-ISO/IEC 9899].

A <c type definition> starting with **typedef** represents a *Syntype-definition*. The <c identifier> of the <c declarator> of the <c type definition> represents the *Syntype-name* of the *Syntype-definition*. The *Range-condition* of the *Syntype-definition* is the predefined Boolean value `true`. There is no *Default-initialization* for the *Syntype-definition*. The *Parent-sort-identifier* of the *Syntype-definition* of the <c type definition> is denoted `PS` below and is determined as follows.

a) If the <c declarator> has a <c pointer>, its type is a typed pointer and `PS` is the *Identifier* of this data type. If the <c pointer> does not include another <c pointer> and the <c type specifier> is the <c type keywords> item **void**, `PS` is the *Identifier* for `Star_void`. Otherwise, if it not `Star_void` and there is a (possibly anonymous) visible *Data-type-definition* for a subtype of `Star_type` where `Atype` of `Star_type` is bound to the type (`target`) of a reduced <c declarator> (that is, the <c declarator> with removal of the <asterisk> and any adjacent <c type qualifier list> from the <c pointer>), `PS` is the *Identifier* of this data type. Otherwise (if it is not `Star_void` and no such type is visible), the <c type definition> represents an anonymous *Data-type-definition* that is a *Value-data-type-definition* in addition to the *Syntype-definition* represented by the <c type definition>, and this anonymous type is the sort defined by the concrete SDL-2010 `value type PS inherits Star_type < target >;`

where

`target` is the target type;

`Star_type` is defined as a parameterized data type in `C_Predefined`.

If the reduced <c declarator> contains a <c pointer>, `target` is a typed pointer to a further typed pointer with a type for either a visible or an anonymous *Data-type-definition* determined in the same way described above. For example, for

```
typedef Existing_type ***Ppp_existing_type;
```

defines a syntype *Name* `Ppp_existing_type` and *Parent-sort-identifier* identifying the anonymous type defined by

```
value type PS inherits
   Star_type<value inherits
      Star_type<value inherits Star_type<Existing_type>>>;
```

and intermediate anonymous types defined by

`value inherits Star_type<value inherits Star_type<Existing_type>>` and `value inherits Star_type<Existing_type>` using inline sort definitions for the two outer `Star_type` parameters.

Otherwise, the reduced <c declarator> does not contain a <c pointer> and `target` is the type identified for the `PS` of the reduced <c declarator> (that is, the type of a <c declarator> with no <c pointer>) as determined in b), c), d) and e) below.

b) If there is no <c pointer> and the <c identifier> in the <c declarator> of the <c type definition> is followed by one or more index specifications (<left square bracket> [<integer c constant expression>] <right square bracket> list in <c direct declarator>), the type is a C array. The C array has a derived anonymous *Data-type-definition* that is a *Value-data-type-definition* in addition to the *Syntype-definition* represented by the <c type definition>. The `PS` is the *Identifier* of this anonymous value type defined by

```
value type PS inherits Cvector <itemsort, arraysize>;
```

where

`arraysize` is the value of the <Integer c constant expression> for this index specification,

`itemsort` is sort of the following index specification if there is one, or the sort identified for PS from the <c type specifier> after the **typedef** as in (c), (d) and (e) below, and

`Cvector` is defined as a parameterized data type in `C_Predefined`.

If there is a following index specification, for the `itemsort` there is a further derived anonymous *Data-type-definition* that is a *Value-data-type-definition* in addition to the *Syntype-definition* represented by the <c type definition>. This is formed in the same way as the *Data-type-definition* for a previous index specification, so that there are as many anonymous data types as index specifications. For example, for

```
typedef int x3d[3][5][7]
```

`PS` is defined by

```
value type PS inherits
   Cvector<value inherits Cvector<value inherits Cvector<int,7>,5>,3>
```

using inline sort definitions for the two outer `itemsort` parameters.

c) If there is no <c pointer> and no index specification after the <c identifier> in the <c declarator> of the <c type definition>, and the <c type specifier> is <c type keywords>, `PS` is the *Identifier* for the <c type keywords>. The identified type shall not be `Void`.

d) Otherwise, if there is no <c pointer> and no index specification after the <c identifier> in the <c declarator> of the <c type definition>, and the <c type specifier> is a <c identifier>, it shall identify a data type and `PS` is the *Identifier* of that data type.

e) Otherwise, if there is no <c pointer> and no index specification after the <c identifier> in the <c declarator> of the <c type definition>, and the <c type specifier> is <c struct or union specifier> or <c enum specifier>, `PS` is the (possibly anonymous) *Identifier* of the data type of the <c type specifier> that is the <c struct or union specifier> or <c enum specifier>. The <c type specifier> that is the <c struct or union specifier> or <c enum specifier> represents a *Data-type-definition* in addition to the *Syntype-definition* represented by the <c type definition>.

C.1.2.1 Data type specifier

Concrete grammar

```
<c type specifier> ::=
                    <c type keywords>
                |   <c struct or union specifier>
                |   <c enum specifier>
                |   <data type c identifier>
```

<c type keywords> ::=

void
| [**signed** | **unsigned**] **char**
| [**signed** | **unsigned**] **short** [**int**]
| [**signed** | **unsigned**] **int**
| [**signed** | **unsigned**] **long** [**long**] [**int**]
| **signed**
| **unsigned**
| **float**
| [**long**] **double**

NOTE – This is a valid subset of <c type specifier> from clause 6.7.2 of [b-ISO/IEC 9899], modified to syntactically restrict the allowed lists of keywords used for data types and also removing **_Bool**, **_Complex** and <c atomic type specifier>.

If the <c type specifier> is <c type keywords>, the keyword list represents the *Sort* for the identified data type of `<<package C_Predefined>>` as follows:

void	identifies `<<package C_Predefined>> Void`
char	identifies `<<package C_Predefined>> Signed_char`
signed char	identifies `<<package C_Predefined>> Unsigned_char`
unsigned char	identifies `<<package C_Predefined>> Unsigned_char`
short	identifies `<<package C_Predefined>> Signed_short`
short int	identifies `<<package C_Predefined>> Signed_short`
signed short	identifies `<<package C_Predefined>> Signed_short`
signed short int	identifies `<<package C_Predefined>> Signed_short`
unsigned short	identifies `<<package C_Predefined>> Unsigned_short`
unsigned short int	identifies `<<package C_Predefined>> Unsigned_short`
int	identifies `<<package C_Predefined>> Signed_int`
signed	identifies `<<package C_Predefined>> Signed_int`
signed int	identifies `<<package C_Predefined>> Signed_int`
unsigned	identifies `<<package C_Predefined>> Unsigned_int`
unsigned int	identifies `<<package C_Predefined>> Unsigned_int`
long	identifies `<<package C_Predefined>> Signed_long`
long int	identifies `<<package C_Predefined>> Signed_long`
signed long	identifies `<<package C_Predefined>> Signed_long`
signed long int	identifies `<<package C_Predefined>> Signed_long`
unsigned long	identifies `<<package C_Predefined>> Unsigned_long`
unsigned long int	identifies `<<package C_Predefined>> Unsigned_long`
long long	identifies `<<package C_Predefined>> Signed_long_long`
long long int	identifies `<<package C_Predefined>> Signed_long_long`
signed long long	identifies `<<package C_Predefined>> Signed_long_long`
signed long long int	identifies `<<package C_Predefined>> Signed_long_long`
unsigned long long	identifies `<<package C_Predefined>> Unsigned_long_long`
unsigned long long int	identifies `<<package C_Predefined>> Unsigned_long_long`
float	identifies `<<package C_Predefined>> Float`

double identifies <<**package** C_Predefined>> Double

long double identifies <<**package** C_Predefined>> Long_double

C.1.2.2 Struct or union specifier

Concrete grammar

<c struct or union specifier>::
<c struct or union> [<c identifier>]
<left curly bracket> <c struct declaration list> <right curly bracket>
| <c struct or union> <struct or union data type c identifier>

NOTE 1 – Clause 6.7.2.1 of [b-ISO/IEC 9899].

<c struct or union> ::=
struct
| **union**

NOTE 2 – Clause 6.7.2.1 of [b-ISO/IEC 9899].

A <c struct or union specifier> that has a <c struct declaration list> represents a *Data-type-definition* that is a *Value-data-type-definition*. The <c identifier> before the <left curly bracket> represents a *Name* that is the *Sort* of the *Value-data-type-definition*. If the <c identifier> is omitted, the *Value-data-type-definition* has an anonymous unique *Name*. The optional *Data-type-identifier* of the *Value-data-type-definition* is omitted. The *Literal-signature-**set***, *Procedure-definition-**set***, *Data-type-definition-**set*** and *Syntype-definition-**set*** of the *Value-data-type-definition* are empty. The *Static-operation-signature-**set*** is derived from <c struct declaration list>.

A <c struct or union specifier> that does not have a <c struct declaration list> represents an *Identifier*. If <c struct or union> is **struct**, the following <struct or union data type c identifier> shall represent an *Identifier* that identifies a *Value-data-type-definition* for a structure. If <c struct or union> is **union**, the following <struct or union data type c identifier> shall represent an *Identifier* that identifies a *Value-data-type-definition* for a union (a choice in SDL-2010 terminology).

<c struct declaration list> ::=
<c struct declaration>
| <c struct declaration list> <c struct declaration>

NOTE 3 – Clause 6.7.2.1 of [b-ISO/IEC 9899].

<c struct declaration> ::=
<c specifier qualifier list> [<c struct declarator list>] <semicolon>

NOTE 4 – Clause 6.7.2.1 of [b-ISO/IEC 9899] without the static-assert-declaration alternative.

<c specifier qualifier list> ::=
<c type specifier>

NOTE 5 – Clause 6.7.2.1 of [b-ISO/IEC 9899] without <c type qualifier> items.

<c struct declarator list> ::=
<c struct declarator>
| <c struct declarator list> <comma> <c struct declarator>

NOTE 6 – Clause 6.7.2.1 of [b-ISO/IEC 9899].

<c struct declarator> ::=
<c declarator>
| <c declarator> <colon> <c constant expression>

NOTE 7 – Clause 6.7.2.1 of [b-ISO/IEC 9899], modified so that the <c declarator> before a <colon> is not optional because SDL-2010 requires a field name. The <c constant expression> has no SDL-2010 meaning.

A <c struct declarator> is a field of a structure or union with a field name that is the <c identifier> of the <c declarator> of the <c struct declarator>. Each field name of a structure or union (<c identifier> of the <c declarator> of a <c struct declarator> of a <c struct declarator list> of a <c struct declaration> of a <c struct declaration list>) shall be different from every other field name

of the same structure or union (the <c struct declaration list>). A <c struct declarator> that has a <c declarator> that contains a <c pointer> is a pointer field. A <c struct declarator> is an array field if it has a <c declarator> that contains one or more index specifications (<left square bracket> <Integer c constant expression> <right square bracket> list in <c direct declarator>). A <c struct declarator> has a field sort that is the *Sort* determined in the same way as determining the *Parent-sort-identifier* of the *Syntype-definition* of a <c type definition> with a **typedef** (see clause C.1.2), except the <c type specifier> is the <c type specifier> of the <c specifier qualifier list> of the <c struct declaration> that encloses the <c struct declarator list> that encloses the <c struct declarator>. As a consequence, it is possible that a <c declarator> of a <c struct declarator> that contains a <c pointer> or index specification represents additional anonymous *Data-type-definition* items. The field sort shall not be `Void`.

The <c struct declaration list> for a structure `S` represents (in the *Static-operation-signature-**set*** of the *Data-type-definition* for `S`):

a) An *Operation-signature* for a generic operator named `Make` with an empty *Formal-argument* list and an *Operation-result* that is the *Sort-reference-identifier* of the `S` structure sort, and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

b) An *Operation-signature* for a generic operator named `Make` with a non-empty *Formal-argument* list where each item is the *Sort-reference-identifier* of the corresponding (in order) field name, and an *Operation-result* that is the *Sort-reference-identifier* of the `S` structure sort, each formal parameter of the procedure identified by the *Operation-signature* has its *Parameter-aggregation* that is **PART**, and a *Result-aggregation* that is **PART**.

c) For each field, if the field name is `fn` and the field sort is `fs`, an *Operation-signature* for the SDL-2010 <operation signature>

```
fnExtract ( S ) -> fs;
```

for a generic operator where

`fnExtract` is a *field-extract-name* formed from the concatenation of the field name and `"Extract"`,

`S` is an **in/out** parameter with a **PART** aggregation kind,

and the result has the same aggregation kind as the field `fn`.

NOTE 8 – A special syntax is provided as described in clause 12.2.3. To use `fnExtract` to extract the value of field `fn` from structure variable `vs` in the context of an expression the notation is:

```
vs.fn
```

d) For each field, if the field name is `fn` and the field sort is `fs`, an *Operation-signature* for the SDL-2010 <operation signature>

```
fnModify ( S, fs ) -> S;
```

for a generic operator where

`fnModify` is a *field-modify-name* formed from the concatenation of the field name and `"Modify"`,

`S` is an **in/out** parameter with a **PART** aggregation kind,

`fs` is an **in** parameter with the same aggregation kind as the field `fn`,

and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

NOTE 9 – A special syntax is provided as described in clause 12.3.3.1 modified to use `"="` for the assignment sign. To use `fnModify` to assign the value `fieldValue` (a value with the sort of field `fn`) to field `fn` of structure variable `vs`, the notation is:

```
vs.fn = fieldValue;
```

The <c struct declaration list> for a union sort `U` represents (in the *Operation-signature* set of the *Data-type-definition* for `U`):

a) An *Operation-signature* for a generic operator named `Make` with an empty *Formal-argument* list and an *Operation-result* that is the *Sort-reference-identifier* of the `U` union sort, and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

b) For a unique field sort with the sort `ufs` (if there are two or more fields with the same sort there is one unique field sort, `ufs`), an *Operation-signature* for a generic operator

```
Make ( ufs ) -> U;
```

for a generic field initialization operator for the leftmost (in the union definition) field with sort `ufs` where
`ufs` is an **in** parameter with the aggregation kind **PART**
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

c) For each field, if the field name is `fn` and the field sort is `fs`, an *Operation-signature* for the SDL-2010 operation signature

```
fn ( fs ) -> U;
```

for a generic field association operator where
`fn` is a *field-associate-name* which is the same as the field name,
`fs` is an **in** parameter with the same aggregation kind as the field `fn`,
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

d) For each field, if the field name is `fn` and the field sort is `fs`, an *Operation-signature* for the SDL-2010 <operation signature>

```
fnExtract ( U ) -> fs;
```

for a generic operator where
`fnExtract` is a *field-extract-name* formed from the concatenation of the field name and `"Extract"`,
`U` is an **in/out** parameter with a **PART** aggregation kind,
and the result has the same aggregation kind as the field `fn`.

NOTE 10 – A special syntax is provided as described in clause 12.2.3. To use `fnExtract` to extract the value of field `fn` from a choice variable `vu` in the context of an expression, the notation is:

```
vu.fn
```

e) For each field, if the field name is `fn` and the field sort is `fs`, an *Operation-signature* for the SDL-2010 <operation signature>

```
fnModify ( U, fs ) -> U;
```

for a generic operator where
`fnModify` is a *field-modify-name* formed from the concatenation of the field name and `"Modify"`,
`U` is an **in/out** parameter with a **PART** aggregation kind,
`fs` is an **in** parameter with a **PART** aggregation kind,
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

NOTE 11 – A special syntax is provided as described in clause 12.3.3.1 modified to use "`=`" for the assignment sign. To use `fnModify` to assign the value of `fieldValue` (a value with the sort of field `fn`) to field `fn` of a choice variable `vu`, the notation is:

```
vu.fn = fieldValue;
```

f) For each field, if the field name is `fn`, an *Operation-signature* for the SDL-2010 <operation signature>
`fnPresent ( U ) -> <<`**`package`** `Predefined>>Boolean;`
for a generic operator where
`fnPresent` is a *field-present-name* formed from the concatenation of the field name and "`Present`",
`U` is an **in/out** parameter with an empty <aggregation kind>,
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.
Whether `fnPresent` is visible in a C diagram is implementation dependent.

g) An *Operation-signature* for a generic operator named `PresentExtract` based on the SDL-2010 <operation signature>
`PresentExtract ( U )-> AnonPresent;`
where `AnonPresent` is defined as a literal constructor data type that uses the field names of the choice as literals as described below,
`U` is an **in/out** parameter with an empty <aggregation kind>,
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.
Whether `PresentExtract` is visible in a C diagram is implementation dependent.

The <c struct or union specifier> for a union type `U` also represents an additional (anonymous) *Data-type-definition*, that for the description above is called `AnonPresent`, in the context that the *Data-type-definition* for `U` occurs. This is defined with a *Literal-signature-**set*** where each field name of the union `U` represents a *Literal-signature*. The order of the literals is the same as the order in which the field names are specified left to right in the union `U`. The purpose of this data type is to allow the operation `PresentExtract` with a result that corresponds to the field name. The name of this data type being unknown prevents it being used for other purposes.

C.1.2.3 Enum specifier

Concrete grammar

<c enum specifier> ::=
enum [<c identifier>]
<left curly bracket> <c enumerator list> <right curly bracket>
| **enum** [<c identifier>]
<left curly bracket> <c enumerator list> <comma> <right curly bracket>
| **enum** <enum data type c identifier>

NOTE 1 – Clause 6.7.2.2 of [b-ISO/IEC 9899].

A <c enum specifier> that has a <left curly bracket> followed by a <c enumerator list> followed by a <comma> followed by a <right curly bracket> has the same meaning as a <c enum specifier> with <left curly bracket> followed by the same <c enumerator list> followed by a <right curly bracket>.

A <c enum specifier> that does not have a <c enumerator list> represents an *Identifier* that identifies a *Value-data-type-definition* for an enumerated type.

A <c enum specifier> that has a <c enumerator list> represents a *Data-type-definition* that is a *Value-data-type-definition* for an enumerated type. The <c identifier> before the <left curly bracket> represents a *Name* that is the *Sort* of the *Value-data-type-definition*. If the <c identifier> is omitted, the *Value-data-type-definition* has an anonymous unique *Name*. The optional *Data-type-identifier* of the *Value-data-type-definition* is omitted. The *Procedure-definition-**set***, *Data-type-definition-**set*** and *Syntype-definition-**set*** of the *Value-data-type-definition* are empty. The *Literal-signature-**set*** and *Static-operation-signature-**set*** are derived from <c enumerator list> as described below.

Each *Literal-signature* has a *Result* that is the *Sort* of the *Value-data-type-definition*.

<c enumerator list> ::=
<c enumerator>
| <c enumerator list> <comma> <c enumerator>

NOTE 2 – Clause 6.7.2.2 of [b-ISO/IEC 9899].

The number of items in the <c enumerator list> shall be less than or equal to `INT_MAX`.

<c enumerator> ::=
<c enumeration constant>
| <c enumeration constant> <equals sign> <c constant expression>

NOTE 3 – Clause 6.7.2.2 of [b-ISO/IEC 9899].

The <c constant expression> shall evaluate to a `Natural` simple expression less than or equal to `INT_MAX`.

<c enumeration constant> ::=
<c identifier>

NOTE 4 – Clause 6.4.4.3 of [b-ISO/IEC 9899].

Each <c enumerator> represents a *Literal-signature*.

Each *Literal-name* is unique within the defining scope unit in the abstract syntax even if the corresponding <c identifier> of the <c enumeration constant> of the <c enumerator> is not unique. The unique *Literal-name* is derived from:

a) the <c identifier> of the <c enumeration constant> of the <c enumerator>; plus

b) the *Result* of the *Literal-signature*.

The Natural value of the <c constant expression> after an <equals sign> represents the *Literal-natural* of the *Literal-signature.*

Each <c enumeration constant> not followed by an <equals sign> in a <c enumerator list> is given the lowest possible Natural value for the *Literal-natural* of the *Literal-signature* not occurring for any other <c enumeration constant> items of the same <c enumerator list>, considering the <c enumeration constant> items one by one from left to right.

C.1.2.4 Sizeof data types

As well as representing abstract grammar items described above, a <c type definition> also represents a synonym definition for the size of the data type as defined below. A synonym definition is a read-only *Variable-definition*. The *Variable-name* is the *Name* derived from prefixing the <c identifier> for the data type with "`sizeof_`" as further described below. The *Sort-reference-identifier* is the *Sort-reference-identifier* for the `Integer` data type. The *Constant-expression* is a *Literal* that is a *Literal-identifier* that is an *Identifier* for the `Integer` value for the size of the data type.

NOTE – If the data type has an anonymous name the synonym for the data type size is also anonymous.

A <c struct or union specifier> or <c enum specifier> that is not a <c type definition> also represents a synonym definition for the size of the data type as well as representing a data type definition. The synonym is named and defined in the same way as <c struct or union specifier> or <c enum specifier> that is a <c type definition>.

There is only a *Variable-definition* for the size of a data type, if the data type is defined in a C diagram and every element of the data type has a defined size. The size of the data types `Boolean`, `Character`, `Bit`, `Octet`, `NumericChar`, `PrintableChar`, `TeletexChar` and `IA5Char` of `<<`**`package`** `Predefined>>` have defined sizes and `<<`**`package`** `C_Predefined>>` defines the corresponding synonyms: `sizeof_Boolean`, `sizeof_Character`, `sizeof_Bit`, `sizeof_Octet`, `sizeof_NumericChar`, `sizeof_PrintableChar`, `sizeof_TeletexChar` and `sizeof_IA5Char`. Other data types of `<<`**`package`** `Predefined>>` such as `Integer` are unbounded, and although they

can be used in C diagrams the size of these data types and any data type having elements of these types is undefined and the corresponding synonym *Variable-definition* for the size does not exist.

If the <c type definition> is a <c type definition> for a C array, the size for the *Constant-expression* is the value of the <Integer c constant expression> for the number of array elements multiplied by the size of the item sort of the array.

If the <c type definition> defines the data type as a syntype for another type (a named data type identified by a <c identifier> or by <c type keywords>), the *Constant-expression* is the same as the *Constant-expression* for the size of the other type.

If the <c type definition> defines a syntype based on a <c type specifier> that is <c struct or union specifier> or <c enum specifier>, the *Constant-expression* is the size of the structure or union or enumerated type defined by the <c struct or union specifier> or <c enum specifier>.

If the <c type definition> is a <c struct or union specifier> for a structure type with a <c struct declaration list>, the *Constant-expression* is the size for the struct, which is implementation dependent but at least the sum of the sizes of all <c struct declarator> items in the <c struct declaration list>. The size of a <c struct declarator> in a structure that is not a C array is the size of the *Sort* of the field sort (that is, the *Sort* represented by <c type specifier> of the <c specifier qualifier list> of the <c struct declaration> that encloses the <c struct declarator list> that encloses the <c struct declarator>). The size of a <c struct declarator> in a structure that is a C array, is the size of a <c struct declarator> of the array item sort multiplied by the <Integer c constant expression> for the number of array elements.

If the <c type definition> is a <c struct or union specifier> for a union type with a <c struct declaration list>, the *Constant-expression* is the size for the union, which is the maximum of each of the sizes of all <c struct declarator> items in the <c struct declaration list>. The size of each <c struct declarator> in a union is the same as the size of the same <c struct declarator> in a structure.

If the <c type definition> is a <c enum specifier>, the *Constant-expression* is the same as the *Constant-expression* for the synonym `<<package C_Predefined>>sizeof_Signed_int`.

In `<<package C_Predefined>>` the following data types are defined with synonyms for their size:

```
Void                 has size <<package C_Predefined>> sizeof_Void
Signed_char          has size <<package C_Predefined>>
                     sizeof_Signed_char
Unsigned_char        has size <<package C_Predefined>>
                     sizeof_Unsigned_char
Signed_short         has size <<package C_Predefined>>
                     sizeof_Signed_short
Unsigned_short       has size <<package C_Predefined>>
                     sizeof_Unsigned_short
Signed_int           has size <<package C_Predefined>>
                     sizeof_Signed_int
Unsigned_int         has size <<package C_Predefined>>
                     sizeof_Unsigned_int
Signed_long          has size <<package C_Predefined>>
                     sizeof_Signed_long
Unsigned_long        has size <<package C_Predefined>>
                     sizeof_Unsigned_long
Signed_long_long     has size <<package C_Predefined>>
                     sizeof_Signed_long_long
Unsigned_long_long   has size <<package C_Predefined>>
                     sizeof_Unsigned_long_long
Float                has size <<package C_Predefined>> sizeof_Float
```

```
Double                  has size <<package C_Predefined>> sizeof_Double
Long_double             has size <<package C_Predefined>>
                        sizeof_Long_double
Star_Void               has size <<package C_Predefined>> sizeof_Star_Void
Star_Signed_char        has size <<package C_Predefined>>
                        sizeof_Star_Signed_char
Star_Unsigned_char      has size <<package C_Predefined>>
                        sizeof_Star_Unsigned_char
Star_Signed_short       has size <<package C_Predefined>>
                        sizeof_Star_Signed_short
Star_Unsigned_short     has size <<package C_Predefined>>
                        sizeof_Star_Unsigned_short
Star_Signed_int         has size <<package C_Predefined>>
                        sizeof_Star_Signed_int
Star_Unsigned_int       has size <<package C_Predefined>>
                        sizeof_Star_Unsigned_int
Star_Signed_long        has size <<package C_Predefined>>
                        sizeof_Star_Signed_long
Star_Unsigned_long      has size <<package C_Predefined>>
                        sizeof_Star_Unsigned_long
Star_Signed_long_long   has size <<package C_Predefined>>
                        sizeof_Star_Signed_long_long
Star_Unsigned_long_long has size <<package C_Predefined>>
                        sizeof_Star_Unsigned_long_long
Star_Float              has size <<package C_Predefined>>
                        sizeof_Star_Float
Star_Double             has size <<package C_Predefined>>
                        sizeof_Star_Double
Star_Long_double        has size <<package C_Predefined>>
                        sizeof_Star_Long_double
```

Other subtypes of the `Star_Type` have the same size as `Star_void`.

C.1.3 Use of C variable definitions

Abstract grammar

The *Sort-identifier* of a *Variable-definition* shall not represent the `Void` data type.

Concrete grammar

<variable definition> ::=
    dcl <variables of sort> {**,** <variables of sort> }* <end>
  | **dcl exported** <exported variables of sort> {**,** <exported variables of sort> }* <end>
  | <c declaration>

A <c declaration> shall only be used in a C diagram.

<c declaration> ::=
    <c declaration specifiers> <c init declarator list> <semicolon>

NOTE 1 – Clause 6.7 of [b-ISO/IEC 9899] <c declaration> omitting the static assert alternative and modified to exclude alternatives used for <c type definition> as an alternative of <data definition>. The declaration specifiers of <c declaration> are restricted to the alternatives defined by <c type specifier> items that start with a data type name or specific keywords. Compared with [b-ISO/IEC 9899] it is not allowed to omit the <c init declarator list>.

A <c declaration> represents a *Variable-definition-**set*** in the enclosing scope.

A <c declaration specifiers> identifies a *Sort-identifier* for a sort used in the *Variable-definition* for each <c init declarator> of the <c init declarator list>. This is either the sort of the variable or the target sort for a pointer or the item sort for a C array, depending on the <c declarator> for <c init declarator> as described below.

<c declaration specifiers> ::=
<c type specifier>
| <c storage class specifier> [<c declaration specifiers>]
| <c type qualifier> [<c declaration specifiers>]

NOTE 2 – Clause 6.7 of [b-ISO/IEC 9899] excluding function-specifier and alignment-specifier alternative. The syntax is rewritten so that there is one and only one <c type specifier> (that is, <c declaration specifiers> is not allowed after <c type specifier>).

If the <c type specifier> is <c type keywords> it identifies a *Sort-identifier* according to the list given for the syntax of <c type keywords> above.

If the <c type specifier> is a <data type c identifier> or <c struct or union specifier> with a <c identifier>, or <c enum specifier> with a <c identifier>, this <c identifier> represents the *Sort-identifier*.

If the <c type specifier> is a <c struct or union specifier> without a <c identifier>, or <c enum specifier> without a <c identifier>, it identifies the *Sort-identifier* for an anonymous unique identifier of the inline type definition given by the <c struct or union specifier> or <c enum specifier>.

If the <c type specifier> is a <c struct or union specifier> with a <c struct declaration list> or a <c enum specifier> with a <c enumerator list>, the <c type specifier> is an inline definition of a data type (see the grammar for <c type specifier>).

<c storage class specifier> ::=
extern
| **auto**
| **register**

NOTE 3 – Clause 6.7.1 of [b-ISO/IEC 9899] excluding **typedef** (used in <c type definition>), **static** and **_Thread_local**. The variables specified as **static** are not considered, as they are difficult to map in SDL-2010 and can be replaced with system or block variables. The keywords **auto** and **register** have no meaning in SDL-2010 and are ignored.

<c type qualifier> ::=
const
| **restrict**
| **volatile**

NOTE 4 – Clause 6.7.3 of [b-ISO/IEC 9899] excluding **_Atomic**. The keywords **volatile** and **restrict** have no meaning in SDL-2010 and are ignored.

The keyword **const** means the item should be initialized and should not be subsequently changed, but has no mapping to the abstract grammar and is therefore treated as annotation with respect to the SDL-2010 semantics. It is permitted for a tool to treat the keyword **const** as defined in clause 6.7.3 of [b-ISO/IEC 9899].

<c init declarator list> ::=
<c init declarator>
| <c init declarator list> <comma> <c init declarator>

NOTE 5 – Clause 6.7 of [b-ISO/IEC 9899].

<c init declarator> ::=
<c declarator>
| <c declarator> <equals sign> <c initializer>

NOTE 6 – Clause 6.7 of [b-ISO/IEC 9899].

Each <c init declarator> represents a *Variable-definition*. The <c identifier> (of the <c declarator> of the <c init declarator>) represents the *Variable-name* of the *Variable-definition*. The *Aggregation-kind* of the *Variable-definition* is **PART**.

If the <c declarator> of a <c init declarator> contains no <c pointer> items and no index specifications (<left square bracket> <Integer c constant expression> <right square bracket> list), the *Sort-reference-identifier* of the *Variable-definition* is the *Sort-identifier* from the <c declaration specifiers> as described above.

If the <c declarator> of a <c init declarator> contains a <c pointer> or one or more index specifications, its type is a typed pointer, or the type of a C Array and *Sort-reference-identifier* of the *Variable-definition* is the *Identifier* of this data type. The data type is derived in the same way as the *Parent-sort-identifier* of the *Syntype-definition* of a <c type definition> with a **typedef** (see clause C.1.2), except the <c type specifier> is the <c type specifier> of the <c declaration specifiers> of the <c declaration> that encloses the <c init declarator list> that encloses the <c init declarator>. As a consequence, it is possible that a <c declarator> of a <c init declarator> that contains a <c pointer> or index specification represents additional anonymous *Data-type-definition* items.

If the <c init declarator> contains a <c initializer>, this represents the optional *Constant-expression* of the *Variable-definition*.

If a <c type qualifier> is **const** for the <c declaration specifiers> of a <c declaration>, each *Variable-definition* represented by the <c declaration> is a read-only variable and shall not be used for the target of an assignment or otherwise modified. In this case, each <c init declarator> shall contain a <c initializer> that initializes the variable.

<c initializer> ::=
<constant c conditional expression>
| <left curly bracket> <c initializer list> <right curly bracket>
| <left curly bracket> <c initializer list> <comma> <right curly bracket>

NOTE 7 – Clause 6.7.9 of [b-ISO/IEC 9899].

The trailing <comma> after a <c initializer list> is ignored so this alternative has the same meaning as the alternative without a <comma>.

<c initializer list> ::=
<c initializer>
<c initializer list> <comma> <c initializer>

NOTE 8 – Clause 6.7.9 of [b-ISO/IEC 9899] excluding designation items.

A <c initializer> shall only contain a <c initializer list> within curly brackets if the variable is a structure or a union or an array. In the case of a structure, there shall be a <c initializer> in the <c initializer list> for every field of the structure and each item shall be compatible with the sort of the corresponding field. The <c initializer list> represents a call of `Make` for the structure with constants represented by the <c initializer> items as actual parameters. In the case of a union, the <c initializer> represents a call of `Make` for the union with the constant represented by the <c initializer> items as the actual parameter. In the case of an array there shall be as many items in the <c initializer list> as there are elements in the array and the sort of each element shall be the same as the item sort of the array. The <c initializer list> represents a static evaluation of `Modify` for the array with the integer value 0 as the index and the constant represented by the first <c initializer> as the other 2 parameters. If this is last element of the array, the first parameter of `Modify` is a static evaluation of the `Make` operator for the array with one parameter as a default value for the element type; otherwise it is a static evaluation of `Modify` with the index value of the next array element and the constant represented by the next <c initializer>. Nested evaluations of `Modify` are repeated until `Modify` is evaluated for the last element. For example, in

```
enum ABC {A, B, C} a3[3] = {A, B, C};
```

the <c initializer list> `{A, B, C}` is evaluated

```
Modify(Modify(Modify(Make,C,2),B,1),A,0)
```

as the *Constant-expression* for the *Variable-definition* of `a3`.

If a field sort or array element sort is itself a structure or array, the <c initializer> for the field or array element that is a nested <c initializer list> within curly brackets initializes the nested structure fields or nested array elements.

C.1.4 Use of C expressions

Concrete grammar

<expression> ::=
<expression0>
| <range check expression>
| <c expression>

The alternatives <expression0> and <range check expression> are not valid in diagrams using C syntax. The alternative <c expression> is only valid in diagrams using C syntax.

<c expression> ::=
<c assignment expression>

NOTE – Clause 6.5.17 of [b-ISO/IEC 9899] without multiple assignments separated by commas.

C.1.4.1 Assignment expression

Concrete grammar

<c assignment expression> ::=
<c conditional expression>
| <c unary expression> <c assignment operator> <c assignment expression>

NOTE 1 – Clause 6.5.16 of [b-ISO/IEC 9899].

The <c unary expression> of a <c assignment expression> is valid if it is

a) a <c postfix expression> that
   i) is a <c primary expression> that is <c identifier> for a variable, and has the type of this variable; or
   ii) is a <c expression> in parentheses that is a valid <c unary expression> for a <c assignment expression>, and has the type of this <c unary expression>; or
   iii) is a <c postfix expression> that identifies a C array item (variable, array element or field) followed by an index specification (<left square bracket> <c expression> <right square bracket>), and has the element type of the array; or
   iv) is a <c postfix expression> that identifies a structure or union item (variable, array element or field) followed by field selection (<full stop> <c identifier>), and has the type of the field selected by <c identifier>;

   or

b) an <asterisk> (dereference) <c unary operator> followed by a <c cast expression> that is a <c postfix expression> that is a valid <c unary expression> for a <c assignment expression>, and <asterisk> (dereference) is defined for the type of the <c unary expression>; or

c) an <asterisk> (dereference) <c unary operator> followed by a <c cast expression> that is a cast (<left parenthesis> <c type name> <right parenthesis> <c cast expression>), and <asterisk> (dereference) is defined for the type of the <c type name >.

A <c postfix expression> identifies C array if it:

a) is a <c primary expression> that is <c identifier> for a variable that is a C array, or

b) starts with an inner <c postfix expression> that identifies C array.

```
<c assignment operator> ::=
                <equals sign>
        |       <c multiplication assignment sign>
        |       <c division assignment sign>
        |       <c remainder assignment sign>
        |       <c addition assignment sign>
        |       <c subtraction assignment sign>
        |       <c shift left assignment sign>
        |       <c shift right assignment sign>
        |       <c bitwise and assignment sign>
        |       <c bitwise excl or assignment sign>
        |       <c bitwise incl or assignment sign>
```

NOTE 2 – Clause 6.5.17 of [b-ISO/IEC 9899].

A <c assignment expression> that is not a <c conditional expression> represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for a method of **package** `C_Predefined` or an implicit type for a pointer (a subtype of `Star_type` of **package** `C_Predefined`). The <c assignment operator> determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "=" if the <c assignment operator> is <equals sign>; "*=" for the <c multiplication assignment sign>; and so on). The type of the <c unary expression> determines the type that defines the method. For example, if this is `Signed_int` the method is defined by `Signed_int` and `Signed_int` corresponds to the *Sort-reference-identifier* of the first *Formal-argument* of the *Operation-signature*. The type of the inner <c assignment expression> determines the *Sort-reference-identifier* of the second and last *Formal-argument* of the *Operation-signature*. These items identify at most one *Operation-signature*, a method of **package** `C_Predefined` or implicit types for pointers. If no *Operation-signature* is identified the <c assignment expression> is not valid. The <c assignment expression> has a type identified by the *Sort-reference-identifier* of the *Operation-result* of the *Operation-signature*.

The <c unary expression> represents the *Expression* for the first item of the *Actual-parameters* of the *Operation-application*.

The inner <c assignment expression> represents the *Expression* for the second item of the *Actual-parameters* of the *Operation-application.*

C.1.4.2 Conditional expression

Concrete grammar

```
<c conditional expression> ::=
                <c logical or expression>
        |       <c logical or expression> <question mark>
                        <c expression> <colon> <c conditional expression>
```

NOTE – Clause 6.5.15 of [b-ISO/IEC 9899].

```
<c constant expression> ::=
                <c conditional expression>
```

A <c conditional expression> containing a <question mark> represents a *Conditional-expression* (see clause 12.2.5 of [ITU-T Z.101]). In this case, the sort of the <c logical or expression> shall be scalar sort, which is defined as one of the following:

a) an integer sort (that is, `Char`, `Signed_char`, `Unsigned_char`, `Signed_short`, `Unsigned_short`, `Signed_int`, `Unsigned_int`, `Signed_long`, `Unsigned_long_long`, `Integer` or the sort of a data type that inherits from one of these); or

b) a floating sort (`Float`, `Double`, `Long_double`, `Real` or the sort of a data type that inherits from one of these); or

c) a pointer sort (`Star_void`, a subtype of `Star_void` or a subtype of `Star_type`); or

d) a boolean sort (the `Boolean` sort or the sort of a data type that inherits from `Boolean`); or

e) a pid sort.

An integer, floating, boolean sort (other than `Boolean`) or pid sort <c logical or expression> before the <question mark> represents the *First-operand* of a negative ("/=") *Equality-expression* that is the *Boolean-expression* of the *Conditional-expression*, and the *Second-operand* of this *Equality-expression* is:

a) the value of the sort equivalent to "0" if the sort of the <c logical or expression> is an integer sort; or

b) the value of the sort equivalent to "0.0" if the sort of the <c logical or expression> is a floating sort; or

c) the value of the sort equivalent to "`false`" if the sort of the <c logical or expression> is a boolean sort; or

d) the *Null-literal-signature* if the sort of the <c logical or expression> is a pointer sort; or

e) the *Null-literal-signature* if the sort of the <c logical or expression> is a pid.

If the sort of the <c logical or expression> before the <question mark> is `Boolean`, the <c logical or expression> directly represents the *Boolean-expression* of the *Conditional-expression*.

The *Consequence-expression* in the *Conditional-expression* is represented by the <c expression> in the <c conditional expression>. The *Alternative-expression* is represented by the inner <c conditional expression> after the <colon>.

C.1.4.3 Logical and bitwise operation expressions

For most operators the operator is treated in a similar way to operators in the native SDL-2010 syntax: that is, the operator is treated as if it were written as a prefix operator or as a quoted operator name such as "+" applied to the operands.

In the case of the logical operators (<c logical or sign> `"||"`, <c logical and sign> `"&&"`), in [b-ISO/IEC 9899] the left-hand operand is evaluated first and the right-hand operand is only evaluated if necessary. So these both have to be handled as a *Conditional-expression*.

Concrete grammar

<c logical or expression> ::=
<c logical and expression>
| <c logical or expression> <c logical or sign> <c logical and expression>

NOTE 1 – Clause 6.5.14 of [b-ISO/IEC 9899].

A <c logical or expression> before a <c logical or sign> of a <c logical or expression> shall not represent a pid sort.

A <c logical or expression> containing a <c logical or sign> represents a *Conditional-expression*. In this case, the <c logical or expression> represents the *Boolean-expression* of the *Conditional-expression* as described in clause C.1.4.2 for the <c logical or expression> of a <c conditional expression> containing a <question mark>. The sort and value of the *Consequence-expression* is:

a) the value of the sort equivalent to "1" if the sort of the <c logical or expression> is an integer sort; or

b) the value of the sort equivalent to "1.0" if the sort of the <c logical or expression> is a floating sort; or

c) the value of the sort equivalent to "`true`" if the sort of the <c logical or expression> is a boolean sort or `Boolean`.

The *Alternative-expression* of the *Conditional-expression* is represented by the <c logical and expression> after the <c logical or sign>.

<c logical and expression> ::=
<c inclusive or expression>
| <c logical and expression> <c logical and sign> <c inclusive or expression>

NOTE 2 – Clause 6.5.13 of [b-ISO/IEC 9899].

A <c logical and expression> containing a <c logical and sign> represents a *Conditional-expression*. In this case, the <c logical and expression> represents the *Boolean-expression* of the *Conditional-expression* as described in clause C.1.4.2 for the <c logical or expression> of a <c conditional expression> containing a <question mark>. The *Consequence-expression* of the *Conditional-expression* is represented by the <c inclusive or expression> after the <c logical and sign>. The sort and value of the *Alternative-expression* is:

a) the value of the sort equivalent to "0" if the sort of the <c logical and expression> is an integer sort; or

b) the value of the sort equivalent to "0.0" if the sort of the <c logical and expression> is a floating sort; or

c) the value of the sort equivalent to "`false`" if the sort of the <c logical and expression> is a boolean sort; or

d) the *Null-literal-signature*, if the sort of the <c logical and expression> is a pid.

<c inclusive or expression> ::=
<c exclusive or expression>
| <c inclusive or expression> <vertical line> <c exclusive or expression>

NOTE 3 – Clause 6.5.12 of [b-ISO/IEC 9899].

<c exclusive or expression> ::=
<c and expression>
| <c exclusive or expression> <circumflex accent> <c and expression>

NOTE 4 – Clause 6.5.11 of [b-ISO/IEC 9899].

<c and expression> ::=
<c equality expression>
| <c and expression> <ampersand> <c equality expression>

NOTE 5 – Clause 6.5.10 of [b-ISO/IEC 9899].

A <c inclusive or expression> containing a <vertical line>, <c exclusive or expression> containing a <circumflex accent> or <c and expression> containing an <ampersand> represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **`package`** `C_Predefined`. The infix operator (<vertical line>, <circumflex accent> or <ampersand>) determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "`|`" for <vertical line>; "`^`" for <circumflex accent>; "`&`" for <ampersand>). The sort of the context of the <c inclusive or expression>, <c exclusive or expression> or <c and expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the infix operator, the expression is not valid. The expression before the infix operator (<c inclusive or expression> for <vertical line>; <c exclusive or expression> for <circumflex accent>; <c and expression> for <ampersand>) represents the first *Expression* of the *Actual-parameters* list of the *Operation-application*. The expression after the infix operator (<c exclusive or expression> for <vertical line>; <c and expression> for <circumflex accent>;

<c equality expression> for <ampersand>) represents the second (and last) *Expression* of the *Actual-parameters* list of the *Operation-application*.

C.1.4.4 Equality and relational expressions

Concrete grammar

```
<c equality expression> ::=
                        <c relational expression>
                |       <c equality expression> <c equality sign>  <c relational expression>
                |       <c equality expression> <c inequality sign> <c relational expression>
```

NOTE 1 – Clause 6.5.9 of [b-ISO/IEC 9899].

A <c equality expression> containing an equality infix operator (<c equality sign> or <c inequality sign>) represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **`package`** `C_Predefined`. The equality infix operator determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "==" for <c equality sign>; "!=" for <c inequality sign>). The sort of the context of the <c equality expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the equality infix operator, the expression is not valid. The <c equality expression> before the equality infix operator represents the first *Expression* of the *Actual-parameters* list of the *Operation-application*. The <c relational expression> after the equality infix operator represents the second (and last) *Expression* of the *Actual-parameters* list of the *Operation-application*.

```
<c relational expression> ::=
                        <c shift expression>
                |       <c relational expression> <less than sign> <c shift expression>
                |       <c relational expression> <greater than sign> <c shift expression>
                |       <c relational expression> <less than or equals sign> <c shift expression>
                |       <c relational expression> <greater than or equals sign> <c shift expression>
```

NOTE 2 – Clause 6.5.8 of [b-ISO/IEC 9899].

A <c relational expression> containing a relational infix operator (<less than sign> or <greater than sign> or <less than or equals sign> or <greater than or equals sign>) represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **`package`** `C_Predefined`. The relational infix operator determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "<" for <less than sign>; ">" for <greater than sign>; "<=" for <less than or equals sign>; ">=" for <greater than or equals sign>). The sort of the context of the <c relational expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the relational infix operator, the expression is not valid. The <c relational expression> before the relational infix operator represents the first *Expression* of the *Actual-parameters* list of the *Operation-application*. The <c shift expression> after the relational infix operator represents the second (and last) *Expression* of the *Actual-parameters* list of the *Operation-application*.

C.1.4.5 Bit shift expressions

Concrete grammar

```
<c shift expression> ::=
                        <c additive expression>
                |       <c shift expression> <c shift left sign> <c additive expression>
                |       <c shift expression> <c shift right sign> <c additive expression>
```

NOTE – Clause 6.5.7 of [b-ISO/IEC 9899].

A <c shift expression> containing a shift infix operator (<c shift left sign> or <c shift right sign>) represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **package** `C_Predefined`. The shift infix operator determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "<<" for <c shift left sign>; ">>" for <c shift right sign>). The sort of the context of the <c shift expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the shift infix operator, the expression is not valid. The <c shift expression> before the shift infix operator represents the first *Expression* of the *Actual-parameters* list of the *Operation-application*. The <c additive expression> after the shift infix operator represents the second (and last) *Expression* of the *Actual-parameters* list of the *Operation-application*.

C.1.4.6 Binary operation expressions

Concrete grammar

<c additive expression> ::=
<c multiplicative expression>
| <c additive expression> <plus sign> <c multiplicative expression>
| <c additive expression> <hyphen> <c multiplicative expression>

NOTE 1 – Clause 6.5.6 of [b-ISO/IEC 9899].

A <c additive expression> containing an additive infix operator (<plus sign> or <hyphen>) represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **package** `C_Predefined`. The additive infix operator determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "+" for <plus sign>; "-" for <hyphen>). The sort of the context of the <c additive expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the additive infix operator, the expression is not valid. The <c additive expression> before the additive infix operator represents the first *Expression* of the *Actual-parameters* list of the *Operation-application*. The <c multiplicative expression> after the additive infix operator represents the second (and last) *Expression* of the *Actual-parameters* list of the *Operation-application*.

<c multiplicative expression> ::=
<c cast expression>
| <c multiplicative expression> <asterisk> <c cast expression>
| <c multiplicative expression> <solidus> <c cast expression>
| <c multiplicative expression> <percent sign> <c cast expression>

NOTE 2 – Clause 6.5.5 of [b-ISO/IEC 9899].

A <c multiplicative expression> containing a multiplicative infix operator (<asterisk> or <solidus> or <percent sign>) represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **package** `C_Predefined`. The multiplicative infix operator determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: `"*"` for <asterisk>; "/" for <solidus>; "%" for <percent sign>). The sort of the context of the <c multiplicative expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the multiplicative infix operator, the expression is not valid. The <c multiplicative expression> before the multiplicative infix operator represents the first *Expression* of the *Actual-parameters* list of the *Operation-application*. The <c cast expression> after the multiplicative infix operator represents the second (and last) *Expression* of the *Actual-parameters* list of the *Operation-application*.

C.1.4.7 Cast expressions

Concrete grammar

<c cast expression> ::=
<c unary expression>
| <left parenthesis> <c type name> <right parenthesis> <c cast expression>

NOTE 1 – Clause 6.5.4 of [b-ISO/IEC 9899].

<c type name> ::=
<c specifier qualifier list>

NOTE 2 – Clause 6.7.7 of [b-ISO/IEC 9899] without the abstract declarator suffix.

The representation of a <c cast expression> without a parenthesized <c type name> is given by the representation of the <c unary expression>. In the following text a <c cast expression> that contains a parenthesized <c type name> is casting expression, and <c cast expression> after the parenthesized <c type name> in a <c cast expression> is a casted expression.

The parenthesized <c type name> in a casting expression shall identify a C integer sort or a C floating sort or a C boolean sort or `Star_void` or a subtype of `Star_type`. This sort is the casting sort.

The sort of the casted expression (the casted sort) shall be the sort of `<<`**`package`** `Predefined>> Integer` or a C integer sort or a C boolean sort or `Star_void` or a subtype of `Star_type`.

The casting expression represents the *Operator-application* with *Operator-identifier* of the operator of the data type for the casting sort to convert a `<<`**`package`** `Predefined>> Integer` value to a value of the casting sort. For example, if <c type name> is **`short int`** the operator is `to_Signed_short` of the data type `Signed_short`; if <c type name> is `Star_Unsigned_long` the operator is `Integer_to_Star_Unsigned_short` of the data type `Star_Unsigned_long`; and if the <c type name> is **`double`** the operator is `Integer_to_Float` of the data type `Float`. The *Actual-parameters* list of the *Operator-application* has one *Expression*.

If the casted sort is `<<`**`package`** `Predefined>> Integer` sort, the *Expression* of the *Actual-parameters* list of the *Operator-application* is represented by the casted expression. Otherwise, *Expression* of the *Actual-parameters* list of the *Operator-application* is represented by another *Operator-application* to convert the value of the casted expression to the `<<`**`package`** `Predefined>> Integer` sort. The *Actual-parameters* list of this *Operator-application* has one *Expression* of the casted sort. For a casted sort that is a C integer sort or C boolean sort, the operator is the `num` operator of the type of the casted sort. For a casted sort that is `Star_void`, the operator is the `Star_void_to_Integer` operator of the type `Star_void`. For a casted sort that is a subtype of `Star_type`, the operator is renamed `Star_type_to_Integer` operator of the subtype of `Star_type` (for example, `Star_Unsigned_int_to_Integer` for the type `Star_Unsigned_int`). The *Expression* of the *Actual-parameters* list of the *Operator-application* (to convert to the `<<`**`package`** `Predefined>> Integer` sort) is represented by the casted expression.

C.1.4.8 Unary operation expression

Concrete grammar

<c unary expression> ::=
<c postfix expression>
| <c increment operator> <c unary expression>
| <c decrement operator> <c unary expression>
| <c unary operator> <c cast expression>
| **sizeof** <c unary expression>
| **sizeof** <left parenthesis> <c type name> <right parenthesis>

NOTE 1 – Clause 6.5.3 of [b-ISO/IEC 9899] excluding the **_Alignof** alternative.

<c unary operator> ::=
    <ampersand>
  | <asterisk>
  | <plus sign>
  | <hyphen>
  | <tilde>
  | <exclamation mark>

NOTE 2 – Clause 6.5.3 of [b-ISO/IEC 9899].

A <c unary expression> containing a <c increment operator> or <c decrement operator> represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **package** C_Predefined. The <c increment operator> or <c decrement operator> determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "++" for <c increment operator>; "--" for <c decrement operator>). The sort of the context of the <c unary expression> and sort of the parameter (the *Expression* item of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the <c increment operator> or <c decrement operator>, the expression is not valid. The <c unary expression> after the <c increment operator> or <c decrement operator> represents the *Expression* of the *Actual-parameters* list of the *Operation-application*.

A <c unary expression> containing a <c unary operator> represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **package** C_Predefined. The <c unary operator> determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "&" for <ampersand>; "*" for <asterisk>; "+" for <plus sign>; "-" for <hyphen>; "~" for <tilde>; "!" for <exclamation mark>). The sort of the context of the <c unary expression> and sort of the parameter (the *Expression* item of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the <c unary operator>, the expression is not valid. The <c cast expression> after the <c unary operator> represents the *Expression* of the *Actual-parameters* list of the *Operation-application*.

A <c unary expression> containing **sizeof** followed by a <c unary expression> represents an *Expression* that is a *Variable_access* for the sizeof_ synonym for the data type of the sort of the <c unary expression>. For example, if the sort is Signed_long, the <c unary expression> represents a *Variable_access* for the sizeof_Signed_long.

A <c unary expression> containing **sizeof** followed by a parenthesized <c type name> represents an *Expression* that is a *Variable_access* for the sizeof_ synonym for the data type of the <c type name>. For example, if the <c type name> is **signed long**, the <c unary expression> represents a *Variable_access* for the sizeof_Signed_long.

C.1.4.9 Postfix expression

Concrete grammar

<c postfix expression> ::=
    <c primary expression>
  | <c call expression>
  | <c postfix expression> <left square bracket> <c expression> <right square bracket>
  | <c postfix expression> <full stop> <c identifier>
  | <c postfix expression> <c pointer operator> <c identifier>
  | <c postfix expression> <c increment operator>
  | <c postfix expression> <c decrement operator>
  | <left parenthesis> <c type name> <right parenthesis>
        <left curly bracket> <c initializer list> <right curly bracket>
  | <left parenthesis> <c type name> <right parenthesis>
        <left curly bracket> <c initializer list> <comma> <right curly bracket>

NOTE 1 – Clause 6.5.2 of [b-ISO/IEC 9899] excluding the alternative with a *postfix-expression* followed by a parenthesized *argument-expression-list* restricted to <c call expression>.

A <c postfix expression> before a square bracketed <c expression> or a <full stop> or a <c increment operator> or a <c decrement operator> shall not be a <c primary expression> that is a <c constant> or a <c string literal>.

A <c postfix expression> containing an additive postfix operator (<c increment operator> or <c decrement operator>) represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **`package`** `C_Predefined`. The additive postfix operator determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: `postfix_inc` for <c increment operator>; `postfix_dec` for <c decrement operator>). The sort of the context of the <c multiplicative expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the multiplicative infix operator, the expression is not valid. The <c postfix expression> before the additive postfix operator represents the *Expression* of the *Actual-parameters* list of the *Operation-application*.

A <c postfix expression> that starts with a parenthesized <c type name> (without or with a <comma> after the <c initializer list>) is a compound literal and shall not be followed by a <c increment operator> or <c decrement operator>. The compound literal represents the static evaluation of the nested calls of the operators as described in clause C.1.3 and the <c postfix expression> represents an *Operator-application* for calling outermost `Modify` operator.

<c call expression> ::=
<c identifier> <left parenthesis> [<c argument expression list>] <right parenthesis>

The <c identifier> in a <c call expression> shall uniquely identify either a *Procedure-identifier* or an *Operation-identifier*. In the context of a <c postfix expression> procedures that do not have a *Result* are excluded. The *Procedure-identifier* or *Operation-identifier* is determined using the name of the <c identifier>, the sorts of the argument list and the sort of the context of the <c call expression> as described in clause 12.2.6 of [ITU-T Z.101]. If more than one procedure or operation is identified, the <c call expression> is invalid. The <c call expression> represents the *Call-node* if it is a <c call expression>; otherwise the <c call expression> represents *Value-returning-call-node* or *Operation-application* of the identified procedure or operation.

<c argument expression list> ::=
<c assignment expression>
| <c argument expression list> <comma> <c assignment expression>

NOTE 2 – Clause 6.5.2 of [b-ISO/IEC 9899].

Each <c assignment expression> in the <c argument expression list> represents an *Expression* in the *Actual-parameters* of the *Call-node* or *Value-returning-call-node* or *Operation-application*.

Model

A <c postfix expression> followed by a <c expression> between square brackets is derived concrete syntax for:

`Extract` (<c postfix expression> , <c assignment expression>)

treated as a <c call expression> where the <c argument expression list> is the <c postfix expression> followed by the <c expression>. If this <c postfix expression> is also a <c postfix expression> followed by a <c expression> between square brackets, this is similarly derived concrete syntax. For example, `a[i][j]` becomes `Extract(Extract(a,i),j)`. The abstract syntax is determined from the derived concrete expression.

A <field primary> is derived concrete syntax for:

field-extract-name (<primary>)

where the *field-extract-name* is formed from the concatenation of the field name and "`Extract`" in that order. The abstract syntax is determined from this concrete expression according to clause 12.2.1.

A <c postfix expression> followed by a <full stop> and a <c identifier> is derived concrete syntax for:

field-extract-name (<c postfix expression>)

where the *field-extract-name* is formed from the concatenation of the field name identified by the <c identifier> and "`Extract`" in that order. The abstract syntax is determined from the derived concrete expression.

A <c postfix expression> followed by <c pointer operator> {`->`) and a <c identifier> is transformed to the dereference operator (`*`) applied to the <c postfix expression> followed by a field extraction, so that it is derived concrete syntax for:

field-extract-name ("`*`" (<c postfix expression>))

where the *field-extract-name* is formed from the concatenation of the field name and "`Extract`" in that order. The field name given by <c identifier> shall be a field for the structure or union pointed at by the <c postfix expression>. The abstract syntax is determined from this concrete expression.

NOTE 3 – Expression `struct_or_union_pointer->field` is the same as `(*struct_or_union_pointer).field`.

C.1.4.10 Primary expression

Concrete grammar

<c primary expression> ::=
<c identifier>
| <c constant>
| <c string literal>
| <left parenthesis> <c expression> <right parenthesis>

NOTE 1 – Clause 6.5.1 of [b-ISO/IEC 9899] excluding generic selection.

NOTE 2 – The lexical rules in clause C.1.1 describe binding to abstract grammar for <c identifier> (see clause C.1.1.6), <c constant> (see clause C.1.1.4) and <c string literal> (see clause C.1.1.5).

A <c identifier> of a <c primary expression> is a variable access or a call of an operation (that has no parameters) or a call of a value-returning procedure (that has no parameters) or the literal (enumeration constant) of an enumerated type. If it is not possible to bind the <c identifier> to exactly one of these items, the <c primary expression> is invalid.

A c primary expression> represents an *Expression* that is either a *Constant-expression* or *Active-expression*.

If the <c identifier> is bound to the *Variable-name* of a *Variable-definition*, the <c identifier> represents an *Active-expression* that is a *Variable-access* of the identified variable.

If the <c identifier> is bound to the *Operation-name* of an *Operation-signature*, the <c identifier> represents an *Active-expression* that is an *Operation-application* of the identified operation.

If the <c identifier> is bound to the *Procedure-name* of a *Procedure-definition*, the <c identifier> represents an *Active-expression* that is a *Value-returning-call-node* of the identified procedure.

If the <c identifier> is bound to the *Identifier* for the *Literal-signature* for an enumeration, the <c identifier> represents a *Constant-expression* for the identified *Literal*.

A <c string literal> of a <c primary expression> represents an *Active-expression* that is a *Variable-access* for the `<<package Predefined>> Star_Unsigned_char` synonym of the <c string literal> as defined in clause C.1.1.5.

A parenthesized <c expression> of a <c primary expression> represents the *Expression* represented by the <c expression>.

<c constant> ::=
<c integer constant>
| <real name>
| <c character constant>

NOTE 3 – Clause 6.4.4 of [b-ISO/IEC 9899] but using <real name> for a floating constant and excluding enumeration constant, because <c identifier> covers this.

A <c integer constant> of a <c constant> represents a *Constant-expression* that is a `<<package Predefined>> Integer` *Literal* for the `Integer` value as in clause C.1.1.2.

A <real name> of a <c constant> represents a *Constant-expression* that is a `<<package Predefined>> Real` *Literal* for the `Real` value of the integer as determined by clause 14.7.1.

A <c character constant> represents a `<<package Predefined type Unsigned_char>>` *Constant-expression* that is the *Operation-application* defined in clause C.1.1.4.

C.1.5 Use of C statements

Statements are allowed in task bodies and in compound statements. A compound statement is itself a statement and the native SDL-2010 compound statement is extended to include C type definitions so that it matches the compound statement of clause 6.8 of [b-ISO/IEC 9899].

A statement is either a terminating statement that transfers the thread of control, or otherwise a non-terminating statement. In a C diagram a terminating statement or non-terminating statement is always a C statement (which includes the extended compound statement).

C.1.5.1 Compound statement

Abstract grammar

Compound-node :: *Connector-name*
*Data-type-definition-**set***
*Variable-definition-**set***
*Init-graph-node**
While-graph-node
Transition
*Step-graph-node**

Compound-node is extended compared with clause 11.14.1 of [ITU-T Z.102] to include a *Data-type-definition-**set***.

Concrete grammar

<compound statement> ::=
[<connector name> :] [<comment body>]
<left curly bracket>
{ <c type definition> <end> }*
[<variable definitions> <end>]
[<statements>] <end>*
<right curly bracket>

The syntax of <compound statement> is extended compared with clause 11.14.1 of [ITU-T Z.102] to allow <c type definition> items each of which represents an element of the *Data-type-definition-**set***.

A <comment body> of a <compound statement> is not allowed in C diagrams. A <c type definition> of a <compound statement> is only allowed in C diagrams.

Semantics

A *Compound-node* is a scope unit for each *Data-type-definition* of the *Compound-node*.

C.1.5.2 Non-terminating and terminating statement

<non terminating statement> ::=
<statement>
| <compound statement>
| <loop statement>
| <decision statement>
| <c labeled statement>
| <c expression statement>
| <c selection statement>
| <c iteration statement>

In <non terminating statement> the alternatives <statement>, <loop statement> and <decision statement> are not allowed in C diagrams. In <non terminating statement> the alternatives <c labeled statement>, <c expression statement>, <c selection statement> and <c iteration statement> are only allowed in C diagrams.

<terminating statement> ::=
<return statement>
| <stop statement>
| <break statement>
| <c jump statement>

In <terminating statement> the alternatives <return statement>, <stop statement> and <break statement> are not allowed in C diagrams. The alternative <c jump statement> of <terminating statement> is only allowed in C diagrams.

C.1.5.3 Labeled statement

<c labeled statement> ::=
<connector name> <colon> <c statement>
| **case** <c constant expression> <colon> <c statement>
| **default** <colon> <c statement>

NOTE – Clause 6.8.1 of [b-ISO/IEC 9899] with <connector name> instead of identifier before the colon.

A <c statement> with a <connector name> represents the *Compound-statement* where the <connector name> represents the *Connector-name*, the <c statement> represents the *Transition* and the *Variable-definition-**set***, *Init-graph-node* list, *While-graph-node Expression* list and *Step-graph-node* list are all empty.

A **case** <c constant expression> or **default** label shall appear only as part of the <c statement> of a **switch** <c selection> statement, and the binding to the abstract grammar for these is described in clause C.1.5.5. The <c constant expression> of each **case** label shall have an integer sort. No two of the **case** <c constant expression> items in the same **switch** <c selection> shall have the same value except any **case** <c constant expression> item for a **switch** <c selection> within the **switch** <c selection>. There shall be at most one **default** label for a **switch** <c selection> except a **default** label for a **switch** <c selection> within the **switch** <c selection>.

C.1.5.4 C statement

<c statement> ::=
<c labeled statement>
| <compound statement>
| <c expression statement>
| <c selection statement>
| <c iteration statement>
| <c jump statement>

NOTE 1 – Clause 6.8 of [b-ISO/IEC 9899] with <compound statement> for the C compound statement. <c jump statement> is a <terminating statement>; the other statements are non-terminating.

<c expression statement> ::=
[<c assignment statement> | <c call expression>] <semicolon>

NOTE 2 – Clause 6.8.4 of [b-ISO/IEC 9899] but with expression changed to [<c assignment statement> | <c call expression>] so that <c expression> is restricted to a <c assignment expression> with a <c assignment operator> or a <c postfix expression> that is <c identifier> <left parenthesis> [<c argument list>] <right parenthesis> (a <c call expression>).

<c assignment statement> ::=
<c postfix expression> <c assignment operator> <c assignment expression>

The <c postfix expression> of a <c assignment statement> shall start with a <c primary> that is a <c identifier> for a variable and shall not contain a <c increment operator> or <c decrement operator>.

A <c assignment statement> represents an *Assignment* or a *Call-node*, as further described below.

A <c assignment statement> where the <c assignment operator> is <equals> represents an *Assignment* (of a *Task-node* of a *Graph-node* of a *Transition*) as further described below. Otherwise the <c assignment statement> represents a *Call-node*. In this case the <c assignment operator>, the sort of the <c postfix expression> and sort of the <c assignment expression> identify an *Operation-signature* in the way described in clause C.1.4.1 (where the <c unary expression> in clause C.1.4.1 is the <c postfix expression> of the <c assignment statement>). If no such method exists, the <c assignment statement> is invalid. The *Call-node* invokes *Procedure-definition* identified by the *Procedure-identifier* of the *Operation-signature*. The <c postfix expression> and <c assignment expression> represent the first and second *Expression* of the *Actual-parameters* list of the *Call-node*. If the <c postfix expression> contains <left square bracket> <c expression> <left square bracket> (an indexed variable) or <full stop> <c identifier> (a field variable), this is an extended variable used as the first parameter of the procedure.

The <c identifier> that starts the <c postfix expression> of a <c assignment statement> for an *Assignment* represents the *Variable-identifier* of the *Assignment*. If the <c postfix expression> contains <left square bracket> <c expression> <left square bracket> (an indexed variable) or <full stop> <c identifier> (a field variable), this is an extended variable and the *Expression* that the <c assignment expression> represents is derived in the same way as for an SDL-2010 extended variable as described in clause 12.3.3.1 of [ITU-T Z.101]. Otherwise the <c assignment expression> represents the *Expression* of the *Assignment*. However, if the sort of the *Expression* is not the same as the sort of the *Variable-identifier* in the following cases this *Expression* is the *Expression* of an *Actual-parameters* list of an *Operation-application* for an operator used to convert the right-hand side to the type of the left-hand side as described below (or for a `Boolean` <c assignment expression> a *Boolean-expression*).

The *Expression* of the *Assignment* is:

a) If the type of the *Variable-identifier* is `<<package Predefined>> Integer`, and the type of the right-hand side *Expression* is:

   i) a C integer type or a C boolean type, an *Operation-application* for the `num` operator for the C type with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the `num` operator;

   ii) a C floating type, an *Operation-application* for the `fix` operator for `<<package Predefined>> Real` with *Operation-application* of the `to_Real` operator of the C floating type as the *Expression* of the *Actual-parameters* list of the `fix` operator. The *Expression* from the <c assignment expression> is the *Actual-parameters* list of the `to_Real` operator;

   iii) `<<package C_Predefined>> Star_void`, an *Operation-application* for the `Star_void_to_Integer` operator of `Star_void` with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the `Star_void_to_Integer` operator;

   iv) a subtype of `<<package C_Predefined>> Star_type`, an *Operation-application* for the renamed `Star_type_to_Integer` operator of the subtype of `Star_type` with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the renamed `Star_type_to_Integer` operator;

   v) `<<package Predefined>> Real`, an *Operation-application* for the `fix` operator for `<<package Predefined>> Real` with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the `fix` operator;

   vi) `<<package Predefined>> Boolean`, a *Conditional-expression* with a *Boolean-expression* that is the *Expression* from the <c assignment expression>, a *Consequence-expression* that is the `<<package Predefined>> Integer` value 1, and a *Consequence-expression* that is the `<<package Predefined>> Integer` value 0.

b) If the type of the *Variable-identifier* is a C integer type, and the type of the right-hand side *Expression* is:

   i) `<<package Predefined>> Integer`, an *Operation-application* for the `integer` operator of the type `Integern` renamed for the C integer type with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the operator;

   ii) a C integer type or a C boolean type or `<<package C_Predefined>> Star_void` or a subtype of `<<package C_Predefined>> Star_type` or `<<package Predefined>> Real`, an *Operation-application* for the `integer` operator of the type `Integern` renamed for the left-hand side C integer type with a `<<package Predefined>> Integer` *Expression* as in a) above as the *Actual-parameters* list of the operator.

c) If the type of the *Variable-identifier* is a C integer type, and the type of the right-hand side *Expression* is:

   i) `<<package Predefined>> Integer`, an *Operation-application* for the `integer` operator of the type `Integern` renamed for the C integer type with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the operator;

   ii) a C integer type or a C boolean type or `<<package C_Predefined>> Star_void` or a subtype of `<<package C_Predefined>> Star_type` or `<<package Predefined>> Real`, an *Operation-application* for the `integer` operator of the type `Integern` renamed for the left-hand side C integer type with a `<<package Predefined>> Integer` *Expression* as in a) above as the *Actual-parameters* list of the operator.

d) If the type of the *Variable-identifier* is a C floating type, and the type of the right-hand side *Expression* is:

   i) `<<package Predefined>> Integer`, an *Operation-application* for the `Integer_to_Float` operator for `<<package C_Predefined>> Float` with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the operator;

   ii) a C integer type or a C boolean type or `<<package C_Predefined>> Star_void` or a subtype of `<<package C_Predefined>> Star_type` or `<<package Predefined>> Real`, an *Operation-application* for the `Integer_to_Float` operator for `<<package C_Predefined>> Float` with a `<<package Predefined>> Integer` *Expression* as in a) above as the *Actual-parameters* list of the operator.

e) If the type of the *Variable-identifier* is `<<package C_Predefined>> Star_void`, and the type of the right-hand side *Expression* is:

   i) `<<package Predefined>> Integer`, an *Operation-application* for the `Integer_to_Star_void` operator for `<<package C_Predefined>> Star_void` with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the operator;

   ii) a C integer type or a C boolean type or a subtype of `<<package C_Predefined>> Star_type` or `<<package Predefined>> Real`, an *Operation-application* for the `Integer_to_Star_void` operator for `<<package C_Predefined>> Star_void` with a `<<package Predefined>> Integer` *Expression* as in a) above as the *Actual-parameters* list of the operator.

f) If the type of the *Variable-identifier* is a subtype of `<<package C_Predefined>> Star_type`, and the type of the right-hand side *Expression* is:

   i) `<<package Predefined>> Integer`, an *Operation-application* for the renamed `Integer_to_Star_type` operator for the subtype of `<<package C_Predefined>> Star_type` with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the operator;

   ii) a C integer type or a C boolean type or `<<package C_Predefined>> Star_void` or a subtype of `<<package C_Predefined>> Star_type` or `<<package Predefined>> Real`, an *Operation-application* for the renamed `Integer_to_Star_type` operator for the subtype of `<<package C_Predefined>> Star_type` with a `<<package Predefined>> Integer` *Expression* as in a) above as the *Actual-parameters* list of the operator.

g) If the type of the *Variable-identifier* is the type of `<<package Predefined>> Charstring`, and the type of the right-hand side *Expression* is `<<package C_Predefined>> Star_Unsigned_char`, an *Operation-application* of the `<<package C_Predefined type Star_Unsigned_char>> Star_Unsigned_char_to_Charstring` operator with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the operator.

C.1.5.5 C selection statement

Concrete grammar

<c selection statement> ::=
  if <left parenthesis> <c expression> <right parenthesis> <c statement>
| **if** <left parenthesis> <c expression> <right parenthesis> <c statement> **else** <c statement>
| **switch** <left parenthesis> <c expression> <right parenthesis> <c statement>

NOTE – Clause 6.8.4 of [b-ISO/IEC 9899].

The <c expression> shall be:

a) an integer sort (that is, `Char`, `Signed_char`, `Unsigned_char`, `Signed_short`, `Unsigned_short`, `Signed_int`, `Unsigned_int`, `Signed_long`, `Unsigned_long_long`, `Signed_long`, `Unsigned_long_long`, `Integer` or the sort of a data type that inherits from one of these); or

b) a floating sort (`Float`, `Double`, `Long_double`, `Real` or the sort of a data type that inherits from one of these); or

c) provided the <c expression> is not in a **switch** <c selection statement>, a pointer sort (`Star_void`, a subtype of `Star_void` or a subtype of `Star_type`); or

d) a boolean sort (the `Boolean` sort or the sort of a data type that inherits from `Boolean`); or

e) a pid sort.

A <c selection statement> represents a *Compound-node*. A newly created anonymous name represents the *Connector-name*. The *Variable-definition-**set***, *Init-graph-node* list, *Expression* list of the *While-graph-node*, and *Step-graph-node* list of the *Compound-node* are empty. There is no *Finalization-node* in the *While-graph-node*. The *Transition* is an empty *Graph-node* list followed by a *Decision-node* and in the case of a **switch** <c selection statement> a set of case *Free-action* items that follow the *Decision-node*. The <c expression> of the <c selection statement> represents the *Decision-question* of the *Decision-body* of the *Decision-node*. The *Decision-answer-**set*** of the *Decision-body* of the *Decision-node* is represented by the <c statement> after the <right parenthesis> in the <c selection statement> as described below.

For an **if** <c selection statement>, the *Decision-answer-**set*** has only one element and the <c statement> after the <right parenthesis> represents the *Transition* of the *Decision-answer* of the *Decision-answer-**set***. The *Range-condition* for this *Decision-answer* is represented by an *Open-range* where the *Operator-identifier* identifies the operator of a negative ("/=") *Equality-expression* of the sort of <c expression>, the <c expression> represents the *First-operand* of the *Equality-expression* and the *Second-operand* of the *Equality-expression* is:

a) the value of the sort equivalent to "0" if the sort of the <c logical or expression> is an integer sort; or

b) the value of the sort equivalent to "0.0" if the sort of the <c logical or expression> is a floating sort; or

c) the value of the sort equivalent to "`false`" if the sort of the <c logical or expression> is a boolean sort; or

d) the *Null-literal-signature* if the sort of the <c logical or expression> is a pointer sort; or

e) the *Null-literal-signature* if the sort of the <c logical or expression> is a pid.

If there is an **else** in an **if** <c selection statement>, the <c statement> after the **else** represents the *Transition* of the optional *Else-answer* of the *Decision-body* of the *Decision-node*.

For a **switch** <c selection statement>, the <c statement> after the <right parenthesis> represents the *Decision-answer-**set*** and also represents the set of *Free-action* items following the *Decision-node*.

Each **case** <c labeled statement> within this <c statement> (and not within an inner **switch** <c selection statement>) represents a *Decision-answer*. The *Range-condition* for this *Decision-answer* is represented by the <c constant expression> of the **case** <c labeled statement>. The *Transition* for this *Decision-answer* is a *Join-node* to an implicit anonymous *Connector-name* for the *Free-action* represented by the <c statement> of the **case** <c labeled statement>. The *Free-action* consists of this *Connector-name* followed by the *Transition* represented by the <c statement> of the **case** <c labeled statement>. If this <c statement> represents a *Graph-node* list without a *Terminator* or *Decision-node*, a *Join-node* to the *Free-action* for the following **case** <c labeled statement> (or **default** <c labeled statement>) is inserted. If there is no such following

<c labeled statement>, a *Break-node* for the *Compound-statement* is inserted. A **default** <c labeled statement> (within the <c statement> after the <right parenthesis> of a **switch** <c selection statement> and not within an inner **switch** <c selection statement>) represents the optional *Else-answer* of the *Decision-body* of the *Decision-node*. The *Transition* for this *Else-answer* is a *Join-node* to an implicit anonymous *Connector-name* for the *Free-action* represented by the <c statement> of the **default** <c labeled statement>. The *Free-action* consists of this *Connector-name* followed by the *Transition* represented by the <c statement> of the **default** <c labeled statement>. The *Transition* is completed if necessary by a *Join-node* or *Break-node* in the same way as for a **case** <c labeled statement>.

C.1.5.6 C iteration and jump statements

Concrete grammar

<c iteration statement> ::=
while <left parenthesis> <c expression> <right parenthesis> <c statement>
| **do** <c statement> **while**
<left parenthesis> <c expression> <right parenthesis> <semicolon>
| **for** <left parenthesis>
[<c expression> { <comma> <c expression> }*] <semicolon>
[<c expression>] <semicolon>
[<c expression> { <comma> <c expression> }*]
<right parenthesis> <c statement>

NOTE 1 – Clause 6.8.5 of [b-ISO/IEC 9899] with **for** syntax modified to allow multiple expressions separated commas because multiple expressions separated by commas are excluded from <c expression>. The **for** alternative with <c declaration> is excluded.

A <c iteration statement> represents a *Compound-node*. The *Connector-name* of the *Compound-node* is a newly created anonymous name. The *Variable-definition-**set*** of the *Compound-node* is empty. The *Transition* of the *Compound-node* is the *Graph-node* represented by the <c statement> of the <c iteration statement> followed by a *Continue-node* with the *Connector-name* of the *Compound-node*. The *Finalization-node* of the *While-graph-node* is absent.

In a **while** <c iteration statement> the *Init-graph-node* list and *Step-graph-node list* are both empty. The parenthesized <c expression> after **while** shall be an expression of a scalar sort (see clause C.1.4.2). This scalar sort expression represents the Boolean *Expression* that forms the *Expression* list of the *While-graph-node* in the same way as the *Boolean-expression* of the *Conditional-expression* is represented in clause C.1.4.2. If the parenthesized <c expression> after **while** is a boolean expression it represents the Boolean *Expression* that forms the *Expression* list of the *While-graph-node*.

In a **do** <c iteration statement> the *Init-graph-node* list is empty and the *While-graph-node* has an empty *Expression* list. The parenthesized <c expression> after **while** shall be an expression of a scalar sort (see clause C.1.4.2). This scalar expression represents Boolean *Expression* in the same way as the *Boolean-expression* of the *Conditional-expression* is represented in clause C.1.4.2. The *Step-graph-node* list is a single *Decision-node* with the Boolean *Expression* as the *Decision-question* of the *Decision-body*, and the *Decision-answer-**set*** is a single *Decision-answer* with the *Range-condition* represented by the Boolean value `false`. The *Transition* of this *Decision-answer* is a *Break-node* with the *Connector-name* being the anonymous name of the *Compound-node* for the <c iteration statement>.

The list of <c expression> items before the <semicolon> of a **for** <c iteration statement> form an *Expression* list for the *Init-graph-node* list in the order of the <c expression> items left to right. Each *Expression* in the list is the *Expression* of an *Assignment-node* that is a *Task-node* in the *Init-graph-node* list of the *Compound-node*.

The <c expression> between the first and second <semicolon> of a **for** <c iteration statement> shall be an expression of a scalar sort and represents the *While-graph-node* of the *Compound-node* in the same way as described above for <c expression> of a **while** <c iteration statement>.

The list of <c expression> items after the second <semicolon> of a **for** <c iteration statement> form an *Expression* list for the *Step-graph-node* item in the order of the <c expression> items left to right. Each *Expression* in the list is the *Expression* of an *Assignment-node* that is a *Task-node* in the *Step-graph-node* list of the *Compound-node*.

Each *Assignment-node* that is a *Task-node* in an *Init-graph-node* or a *Step-graph-node* list has an anonymous implicit variable identified by its *Variable-identifier* of the same sort as the *Expression* of the *Assignment-node*.

<c jump statement> ::=
    goto <c identifier> <semicolon>
  | **break** <semicolon>
  | **continue** <semicolon>
  | **return** [<c expression>] <semicolon>

NOTE 2 – Clause 6.8.6 of [b-ISO/IEC 9899].

A **goto** <c jump statement> represents a *Join-node* with the *Connector-name* represented by the <c identifier> of the <c jump statement>.

A **break** <c jump statement> is only valid within the <c statement> in a **switch**, **while**, **do** or **for** <c iteration statement>. It is mapped to a **join** on the anonymous connector *anon-break* for the enclosing **switch**, **while**, **do** / **while** or **for** statement.

A **break** <c jump statement> represents a *Break-node* and the *Connector-name* is the name of the immediately enclosing *Compound-node*.

A **continue** <c jump statement> is only valid within the <c statement> in a **while**, **do** or **for** <c iteration statement>. It is mapped to a **join** on the anonymous connector *anon-continue* for the enclosing **while**, **do** / **while** or **for** statement.

A **continue** <c jump statement> represents a *Continue-node* and the *Connector-name* is the name of the immediately enclosing *Compound-node*.

If a **return** <c jump statement> with an <c expression> represents a *Value-return-node* and the <c expression> represents the *Expression* of the *Value-return-node*.

A **return** <c jump statement> without an <expression> represents an *Action-return-node*.

C.1.6 Package C_Predefined

In the following definitions, all references to names defined in the **`package`** `Predefined` are treated as prefixed by the qualification `<<`**`package`** `Predefined>>`. Similarly, all references to names defined in the **`package`** `C_Predefined` are treated as prefixed by the qualification `<<`**`package`** `C_Predefined>>`. To increase readability, these qualifications are omitted.

The extensions defined in Annex A are used in this package.

There are no literals for the C Integer types. When a <c integer constant> is used, it represents an `Integer` literal defined in the **`package`** `Predefined`, so no ambiguity is introduced between various integer literals.

Every C Integer type defined below includes a cast operator that converts an `Integer` value to the C Integer type. The name of this operator is `to_` concatenated with the name of the type.

This package assumes that C Integer types values do not have padding bits, have a sign bit for signed integers, use two's complement arithmetic and are wrapped when they overflow. For example, for `Unsigned_char` as defined below with 8 bits `to_Unsigned_char(255) + to_Unsigned_char(1) = to_unsigned_char (0)` and for `Signed_char` as defined below with 8 bits `to_Signed_char(127) + to_Signed_char(1) = to_signed_char(-128)`.

Library functions of clause 7 of the C standard clause 5.2.4.2 of [b-ISO/IEC 9899] are not included in **`package`** `C_Predefined`. The mechanism for including such C library items is implementation defined, but one possibility is to use the standard header names (such as `assert`) as the names of packages. The C library item `stdint` defines some general cases of integers where the number of bits for the C Integer type is *`N`* with names `uint`*`N`*`_t` for the unsigned case and `int`*`N`*`_t` for the signed case, the value range for `uint`*`N`*`_t` is `to_Uint`*`N`*`_t(0)` to `to_Uint`*`N`*`_t(power(2,N)-1)` and `to_Uint`*`N`*`_t(power(2,`*`N`*`)-1) + to_Uint`*`N`*`_t(1) = to_Uint`*`N`*`_t(0)`, and the value range for `int`*`N`*`_t` is `to_int`*`N`*`_t(-power(2,`*`N`*`-1))` to `to_intN_t(power(2,`*`N`*`-1)-1)` and `to_int`*`N`*`_t(power(2,`*`N`*`-1)-1) + to_int`*`N`*`_t(1) = to_intN_t(-power(2,`*`N`*`-1))`.

```
/* */
package C_Predefined
/*
```

The following <package public> defines the synonyms and types that are visible wherever `C_Predefined` is visible.

It is not necessary to explicitly list operations as visible, because literals and operations defined by a type are visible where the type is visible except if the operation has **`private`** visibility (in which case it is only visible within the data type where it is defined) or **`protected`** visibility (in which case the operator is visible only within the data type where it is defined and within any specialization of this data type).

```
*/
public
/* synonyms */
synonym CHAR_BIT,
synonym CHAR_MAX,
synonym CHAR_MIN,
synonym DBL_DECIMAL_DIG,
synonym DBL_DIG,
synonym DBL_EPSILON,
synonym DBL_MANT_DIG,
synonym DBL_MAX_10_EXP,
synonym DBL_MAX_EXP,
synonym DBL_MAX,
synonym DBL_MIN_10_EXP,
synonym DBL_MIN_EXP,
synonym DBL_MIN,
synonym DBL_TRUE_MIN,
synonym DECIMAL_DIG,
synonym FLT_DECIMAL_DIG,
synonym FLT_DIG,
synonym FLT_EPSILON,
synonym FLT_MANT_DIG,
synonym FLT_MAX_10_EXP,
synonym FLT_MAX_EXP,
synonym FLT_MAX,
synonym FLT_MIN_10_EXP,
synonym FLT_MIN_EXP,
synonym FLT_MIN,
synonym FLT_RADIX,
synonym FLT_TRUE_MIN,
synonym INT_BIT,
synonym INT_MAX,
synonym INT_MIN,
synonym LDBL_DECIMAL_DIG,
synonym LDBL_DIG,
synonym LDBL_EPSILON,
synonym LDBL_MANT_DIG,
```

```
synonym LDBL_MAX_10_EXP,
synonym LDBL_MAX_EXP,
synonym LDBL_MAX,
synonym LDBL_MIN_10_EXP,
synonym LDBL_MIN_EXP,
synonym LDBL_MIN,
synonym LDBL_TRUE_MIN,
synonym LLONG_BIT,
synonym LLONG_MAX,
synonym LLONG_MIN,
synonym LONG_BIT,
synonym LONG_MAX,
synonym LONG_MIN,
synonym MB_LEN_MAX,
synonym SCHAR_MAX,
synonym SCHAR_MIN,
synonym SHRT_BIT,
synonym SHRT_MAX,
synonym SHRT_MIN,
synonym sizeof_Bit ,
synonym sizeof_Boolean ,
synonym sizeof_Character ,
synonym sizeof_Float,
synonym sizeof_IA5Char ,
synonym sizeof_NumericChar ,
synonym sizeof_Octet ,
synonym sizeof_PrintableChar ,
synonym sizeof_Signed_char,
synonym sizeof_Signed_int,
synonym sizeof_Signed_long_long,
synonym sizeof_Signed_long,
synonym sizeof_Signed_short,
synonym sizeof_Star_Float,
synonym sizeof_Star_Signed_char,
synonym sizeof_Star_Signed_int,
synonym sizeof_Star_Signed_long_long,
synonym sizeof_Star_Signed_long,
synonym sizeof_Star_Signed_short,
synonym sizeof_Star_Unsigned_char,
synonym sizeof_Star_Unsigned_int,
synonym sizeof_Star_Unsigned_long_long,
synonym sizeof_Star_Unsigned_long,
synonym sizeof_Star_Unsigned_short,
synonym sizeof_Star_void,
synonym sizeof_TeletexChar ,
synonym sizeof_Unsigned_char,
synonym sizeof_Unsigned_int,
synonym sizeof_Unsigned_long_long,
synonym sizeof_Unsigned_long,
synonym sizeof_Unsigned_short,
synonym sizeof_void,
synonym UCHAR_MAX,
synonym UINT_MAX,
synonym ULLONG_MAX,
synonym ULONG_MAX,
synonym USHRT_MAX,
/*types*/
type Cvector,
type Double,
type Float,
type Long_Double
type Signed_char,
type Signed_int,
type Signed_long_long,
type Signed_long,
type Signed_short,
type Star_Float
type Star_Signed_char,
type Star_Signed_int,
type Star_Signed_long_long,
type Star_Signed_long,
```

```
type Star_Signed_short,
type Star_type,
type Star_Unsigned_char,
type Star_Unsigned_int,
type Star_Unsigned_long_long,
type Star_Unsigned_long,
type Star_Unsigned_short,
type Star_void,
type Unsigned_char,
type Unsigned_char,
type Unsigned_int,
type Unsigned_long_long,
type Unsigned_long,
type Void;
/*
```

C.1.6.1 Numerical limits

A C implementation is required to document all the limits specified as synonyms in this clause (see clause 5.2.4.2 of [b-ISO/IEC 9899]).

C.1.6.1.1 Private limits

The following external synonyms are additional to limits specified in clause 5.2.4.2 of [b-ISO/IEC 9899] for use within **package** C_Predefined and are considered private to this package. Each of these synonyms can therefore be consistently renamed in this package to avoid name clashes with a C implementation library or items in the system being defined.

```
*/
/* number of bits for the type Short_int*/
synonym SHRT_BIT Integer = external;/* 16 */;
/* */
/* number of bits for the type int*/
synonym INT_BIT Integer = external;/* 16 */;
/* */
/* number of bits for the type Long_int*/
synonym LONG_BIT Integer = external;/* 32 */;
/* */
/* number of bits for the type Long_long_int*/
synonym LLONG_BIT Integer = external;/* 64 */;
/*
```

C.1.6.1.2 Sizes of integer types

The values given here vary according to the implementation, and therefore are given as external synonyms. The values in the comments are derived from clause 5.2.4.2.1 of [b-ISO/IEC 9899].

```
*/
/* number of bits for smallest item that is not a bit-field (byte)*/
synonym CHAR_BIT Integer = external; /* 8 */
/* */
/* minimum value for type Signed_char */
synonym SCHAR_MIN Integer = external; /* -128 that is –power(2, CHAR_BIT-1) */
/* */
/* maximum value for type Signed_char */
synonym SCHAR_MAX Integer = external; /* +127 that is power(2, CHAR_BIT-1) – 1 */
/* */
/* maximum value for type Unsigned_char */
synonym UCHAR_MAX Integer = external; /* 255 that is power(2, CHAR_BIT) – 1 */
/* */
/* minimum value for type denoted by char */
synonym CHAR_MIN Integer = external; /* 0 - char treated as a signed integer */
/* */
/* maximum value for type denoted by char */
synonym CHAR_MAX Integer = external; /* SCHAR_MAX - char treated as a signed integer */
/* */
/* maximum number of bytes in a multibyte character, for any supported locale */
synonym MB_LEN_MAX Integer = external; /* 1 that is multibyte characters are not
allowed*/
/* */
/* minimum value for type Signed_short */
```

```
synonym SHRT_MIN Integer = external; /* -32768 that is –power(2,SHRT_BIT-1) */
/* */
/* maximum value for type Signed_short */
synonym SHRT_MAX Integer = external; /* +32767 that is power(2,SHRT_BIT-1) – 1 */
/* */
/* maximum value for type Unsigned_short */
synonym USHRT_MAX Integer = external; /* 65535 that is power(2,SHRT_BIT) – 1 */
/* */
/* minimum value for type Signed_int */
synonym INT_MIN Integer = external; /* -32768 that is –power(2,INT_BIT-1) */
/* */
/* maximum value for type Signed_int */
synonym INT_MAX Integer = external; /* +32767 that is power(2,INT_BIT-1) – 1 */
/* */
/* maximum value for type Unsigned_int */
synonym UINT_MAX Integer = external; /* 65535 that is power(2,INT_BIT) – 1 */
/* */
/* minimum value for type Signed_long */
synonym LONG_MIN Integer = external; /* -2147483648 that is –power(2,LONG_BIT-1) */
/* */
/* maximum value for type Signed_long */
synonym LONG_MAX Integer = external; /* +2147483647 that is power(2,LONG_BIT-1) – 1 */
/* */
/* maximum value for type Unsigned_long */
synonym ULONG_MAX Integer = external; /* 4294967295 that is power(2,LONG_BIT) – 1 */
/* */
/* minimum value for type Signed_long_long */
synonym LLONG_MIN Integer = external; /* -9223372036854775808
                                         that is –power(2,LLONG_BIT-1) */
/* */
/* maximum value for type Signed_long_long */
synonym LLONG_MAX Integer = external; /* +9223372036854775807
                                         that is power(2,LLONG_BIT-1) – 1 */
/* */
/* maximum value for type Unsigned_long_long */
synonym ULLONG_MAX Integer = external; /* 18446744073709551615
                                          that is power(2,LLONG_BIT) – 1 */
/*
```

C.1.6.1.3 Characteristics of floating types

The values given here vary according to the implementation, and therefore are given as external synonyms. The values in the comments are derived from clause 5.2.4.2.2 of [b-ISO/IEC 9899].

```
*/
/*  number of decimal digits, n, such that any Double floating-point number with p radix
b digits can be rounded to a floating-point number with n decimal digits and back again
without change to the value */
synonym DBL_DECIMAL_DIG Integer = external; /* 10 */
/* */
/* number of decimal digits, q, such that any Double floating-point number with q decimal
digits can be rounded into a floating-point number with p radix b digits and back again
without change to the q decimal digit */
synonym DBL_DIG Integer  = external; /* 10 */
/* */
/* the difference between 1 and the least value greater than 1 that is representable in
the Double floating point type*/
synonym DBL_EPSILON Real = external; /*10.0E-5*/
/* */
/* number of base-FLT_RADIX digits in the Double floating-point significand, p */
synonym DBL_MANT_DIG Integer = external;
/* */
/* maximum integer such that 10 raised to that power is in the range of representable
finite Double floating-point numbers */
synonym DBL_MAX_10_EXP Integer = external; /* +37 */
/* */
/*maximum representable finite Double floating-point number*/
synonym DBL_MAX Real = external; /* 10.0E37 */
/* */
/* maximum integer such that FLT_RADIX raised to one less than that power is a
representable finite Double floating-point number */
synonym DBL_MAX_EXP Integer = external;
```

```
/* */
/* minimum normalized positive double floating-point number*/
synonym DBL_MIN Real = external; /* 10.0E-37 */
/* */
/* minimum negative integer such that 10 raised to that power is in the range of
normalized Double floating-point numbers */
synonym DBL_MIN_10_EXP Integer = external; /* -37 */
/* */
/* minimum negative integer such that FLT_RADIX raised to one less than that power is a
normalized Double floating-point number, emin */
synonym DBL_MIN_EXP Integer = external;
/* */
/* minimum positive Double floating-point number */
synonym DBL_TRUE_MIN Real = external; /* 10.0E-37 */
/* */
/* number of decimal digits, n, such that any floating-point number in the widest
supported floating type with pmax radix b digits can be rounded to a floating-point
number with n decimal digits and back again without change to the value, */
synonym DECIMAL_DIG Integer = external; /* 10 */
/* */
/* number of decimal digits, n, such that any Float floating-point number with p radix b
digits can be rounded to a floating-point number with n decimal digits and back again
without change to the value */
synonym FLT_DECIMAL_DIG Integer = external; /* 6 */
/* */
/* number of decimal digits, q, such that any Float floating-point number with q decimal
digits can be rounded into a floating-point number with p radix b digits and back again
without change to the q decimal digits */
synonym FLT_DIG Integer = external; /* 6 */
/* */
/* the difference between 1 and the least value greater than 1 that is representable in
the Float floating point type*/
synonym FLT_EPSILON Real = external; /*10.0E-5*/
/* */
/* number of base-FLT_RADIX digits in the Float floating-point significand, p*/
synonym FLT_MANT_DIG Integer = external;
/* */
/*maximum representable finite Float floating-point number*/
synonym FLT_MAX Real = external; /* 10.0E37 */
/* */
/* maximum integer such that 10 raised to that power is in the range of representable
finite Float floating-point numbers */
synonym FLT_MAX_10_EXP Integer = external; /* +37 */
/* */
/* maximum integer such that FLT_RADIX raised to one less than that power is a
representable Float finite floating-point number */
synonym FLT_MAX_EXP Integer = external;
/* */
/* minimum normalized positive Float floating-point number*/
synonym FLT_MIN Real = external; /* 10.0E-37 */
/* */
/* minimum negative integer such that 10 raised to that power is in the range of
normalized Float floating-point numbers */
synonym FLT_MIN_10_EXP Integer = external; /* -37 */
/* */
/* minimum negative integer such that FLT_RADIX raised to one less than that power is a
normalized Float floating-point number, emin */
synonym FLT_MIN_EXP Integer = external; /*
/* */
/* the radix (or base) for floating point numbers */
synonym FLT_RADIX Integer = external; /*2
/* */
/* minimum positive Float floating-point number */
synonym FLT_TRUE_MIN Real = external; /* 10.0E-37 */
/* */
/* number of decimal digits, n, such that any long double  floating-point number with p
radix b digits can be rounded to a floating-point number with n decimal digits and back
again without change to the value */
synonym LDBL_DECIMAL_DIG Integer = external; /* 10 */
/* */
/* number of decimal digits, q, such that any Long_double floating-point number with q
```

```
decimal digits can be rounded into a floating-point number with p radix b digits and back
again without change to the q decimal digits*/
synonym LDBL_DIG Integer = external; /*10*/
/* */
/* the difference between 1 and the least value greater than 1 that is representable in
the Long_double floating point type*/
synonym LDBL_EPSILON Real = external; /*10.0E-5 */
/* */
/* number of base-FLT_RADIX digits in the Long_double floating-point significand, */
synonym LDBL_MANT_DIG Integer = external;
/* */
/*maximum representable finite Long_double floating-point number*/
synonym LDBL_MAX Real = external; /* 10.0E37 */
/* */
/* maximum integer such that 10 raised to that power is in the range of representable
finite Long_double floating-point numbers */
synonym LDBL_MAX_10_EXP Integer = external; /* +37 */
/* */
/* maximum integer such that FLT_RADIX raised to one less than that power is a
representable finite Long_double floating-point number */
synonym LDBL_MAX_EXP Integer = external;
/* */
/* minimum normalized positive Long_double floating-point number*/
synonym LDBL_MIN Real = external; /* 10.0E-37 */
/* */
/* minimum negative integer such that 10 raised to that power is in the range of
normalized Long_double floating-point numbers */
synonym LDBL_MIN_10_EXP Integer = external; /* -37 */
/* */
/* minimum negative integer such that FLT_RADIX raised to one less than that power is a
normalized Long_double floating-point number, emin */
synonym LDBL_MIN_EXP Integer = external;
/* */
/* minimum positive Long_double floating-point number */
synonym LDBL_TRUE_MIN Real = external; /* 10.0E-37 */
/*
```

C.1.6.1.4 Sizes of package Predefined types

The following synonyms are defined for the sizes of some types defined in **package** `Predefined`. Other types defined in **package** `Predefined` are unbounded, therefore no synonym is defined for the size.

```
*/
synonym sizeof_Bit Integer = 1;
synonym sizeof_Boolean Integer = 1;
synonym sizeof_Character Integer = 1;
synonym sizeof_IA5Char Integer = 1;
synonym sizeof_NumericChar Integer = 1;
synonym sizeof_Octet Integer = 1;
synonym sizeof_PrintableChar Integer = 1;
synonym sizeof_TeletexChar Integer = 1;
/*
```

C.1.6.2 Parameterized types for integers

A C integer type is one of 5 signed types or 5 unsigned types: `Signed_char`, `Signed_short`, `Signed_int`, `Signed_long` or `Signed_long_long`, `Unsigned_char`, `Unsigned_short`, `Unsigned_int`, `Unsigned_long` or `Unsigned_long_long`.

The types for C integers inherit the parameterized types below with the parameters bound to the size in bits of the integer type and the other signed or unsigned integer types. The operation identifiers are overloaded. For a binary operator such as the "+" operator the first and second parameter is any of the 10 signed and unsigned C integer types or the `Integer` type. The types for C integers therefore define 100 binary `"+"` operators with different signatures. Because the result is always the `Integer` type, and the integer literals always represent values of the `Integer` type, it is always possible (providing no further overloading of `"+"` is introduced by the user) to determine the correct `"+"` operator from the sorts of the parameters.

Except for comparison operators in a context where a Boolean is required and the "&" operator to obtain a pointer to an object, each operation for a C integer types has an Integer result. As a consequence, it is not necessary to provide operators to convert between C integer types for expressions. When there is an explicit or implied casting (for example, when an Unsigned_char variable c is assigned to a Signed_short variable s), the appropriate num operator is used to convert to Integer and the resulting value converted to the target sort: in the example given to_Signed_short(<<type Unsigned_char>>num(s)).

C.1.6.2.1 Integern for a generic integer type with n bits

This type is defined as a generic type for both signed and unsigned integers. The parameters Int_char, Int_short, Int_int, Int_long and Int_long_long are bound to each of the corresponding unsigned integer types to produce a parameterized type Integers for signed Integer types. For unsigned integer types, the parameters Int_char, Int_short, Int_int, Int_long and Int_long_long are bound to each of the corresponding signed integer types to produce a parameterized type Integeru.

Each binary operator is defined for Integern with Integer, Integer with Integern, Integern with Integern, and Integern with each of the other 9 integer types, and that are given below in Integern as Integer1, Integer2, Integer3, Integer4, Int_char, Int_short, int_int, Int_long and int_long_long. Each signed integer type is defined using Integern with n bound to the number of bits in Integern, and the other context parameters bound to the other signed integer types.

There are no literals for Integern or types based on Integern. Instead the values of Integern are represented by <<type Integern>>integer (i) where i is an Integer value between -power(2,n-1) and power(2,n-1)-1 for signed integer types and zero to power(2,n)-1 for unsigned integer types. The implicit ordering of values in clause 12.1.6.1 of [ITU-T Z.101] does not apply, therefore unordered is given.

To shorten the description of Integern below, for each operation only the <operation definition> is given from which the <operation signature> is constructed (see the *Model* part of clause 12.1.7).

```
abstract value type Integern <
     value type Int_char; value type Int_short; value type Int_int;
     value type Int_long; value type Int_long_long;
     /* For signed integers these are given actual parameters for unsigned integers:
        Int_char=Unsigned_char, Int_sort=Unsigned_short, Int_int = Unsigned_int,
        Int_long=Unsigned_long, Int_sort=Unsigned_long_long.
        For unsigned integers these are given actual parameters for signed integers:
        Int_char=Signed_char, Int_sort=Signed_short, Int_int = Signed_int,
        Int_long=Signed_long, Int_sort=Signed_long_long */
     synonym n Natural; /*number of bits in Integern*/
     value type Integer1; value type Integer2; value type Integer3; value type Integer4
     /* These are given actual parameters for the other integer types.
        For exmaple when defininining Signed_long_long the values are:
        Integer1 = Signed_char, Integer2 = Signed_short,
        Integer3 = Signed_int,  Integer4 = Signed_long */ >
{
     literals unordered; /* "<", ">","<=",">=", first, last, pred, succ,
        and num are not implicitly defined.
     operators and methods signature lists are constructed from the definitions
        of the operations below except for operations used here but defined in subtypes.
*/
     operators /* Signatures for operations used here but only defined in subtypes. */
        integer ( Integer )-> this Integern; /* convert to a C integer */
        num (Integern )-> Integer; /* convert to a SDL-2010 Integer */
     /* shift left operators */
        "<<" (Integern,Integer ) -> Integer;
        "<<" (Integer,Integern ) -> Integer;
     /* shift right operators */
```

```
      ">>" (Integern, i Integer ) -> Integer;
      ">>" (Integer, nv Integern ) -> Integer;
      /* */
      "&" (Integern, Integer ) -> Integer; /* bitwise 'and' */
      "^" (Integern, Integer ) -> Integer; /* bitwise 'xor' */
      "|" (Integern, Integer ) -> Integer /*bitwise 'or' */
/*
   operator integer ( i Integer )-> this Integern
      is defined for signed/unsigned integers with different body definitions,
      and with different names.
   operator num ( nv this Integern )-> Integer – for signed integers{}
   operator num ( nv this Integern )-> Natural– for unsigned integers{}
      are defined for signed/unsigned integers with different body definitions.
*/
   operator "&" ( in/out nv this Integern )-> Star_void external;
/*  The untyped 'address of' operator.
      The typed 'address of' operator
      "&"( Integern )-> the typed star type for the integer type, and
      the dereference operator
      "*" (the typed star type for the integer type) -> Integern are
         defined using Star_type with the integer type as the actual parameter.
*/
/*  Negation and Comparisons with Boolean results */
   operator "!"  ( nv1 Integern) -> Boolean {return num(nv1)= 0 }
   operator "<"  ( nv1 Integern, i Integer) -> Boolean {return num(nv1)< i }
   operator "<"  ( nv1 Integern, nv2 Integern) -> Boolean {return num(nv1)< num(nv2)}
   operator "<"  ( i Integer, nv1 Integern) -> Boolean {return i < num(nv1)}
   operator "<"  ( nv1 Integern, nv2 Integer1) -> Boolean {return num(nv1)< num(nv2)}
   operator "<"  ( nv1 Integern, nv2 Integer2) -> Boolean {return num(nv1)< num(nv2)}
   operator "<"  ( nv1 Integern, nv2 Integer3) -> Boolean {return num(nv1)< num(nv2)}
   operator "<"  ( nv1 Integern, nv2 Integer4) -> Boolean {return num(nv1)< num(nv2)}
   operator "<"  (nv1 Integern,nv2 Int_char)->Boolean{return num(nv1)< num(nv2)}
   operator "<"  (nv1 Integern,nv2 Int_short)->Boolean{return num(nv1)< num(nv2)}
   operator "<"  (nv1 Integern,nv2 Int_int)->Boolean{return num(nv1)< num(nv2)}
   operator "<"  (nv1 Integern,nv2 Int_long)->Boolean{return num(nv1)< num(nv2)}
   operator "<"  (nv1 Integern,nv2 Int_long_long)->Boolean{return num(nv1)< num(nv2)}
   operator "<=" ( nv1 Integern, nv2 Integern) -> Boolean {return num(nv1)<=num(nv2)}
   operator "<=" ( nv1 Integern, i Integer) -> Boolean {return num(nv1)<=i }
   operator "<=" ( i Integer, nv1 Integern) -> Boolean {return i<=num(nv1) }
   operator "<=" ( nv1 Integern, nv2 Integer1) -> Boolean {return num(nv1)<=num(nv2)}
   operator "<=" ( nv1 Integern, nv2 Integer2) -> Boolean {return num(nv1)<=num(nv2)}
   operator "<=" ( nv1 Integern, nv2 Integer3) -> Boolean {return num(nv1)<=num(nv2)}
   operator "<=" ( nv1 Integern, nv2 Integer4) -> Boolean {return num(nv1)<=num(nv2)}
   operator "<=" (nv1 Integern,nv2 Int_char)->Boolean{return num(nv1)<=num(nv2)}
   operator "<=" (nv1 Integern,nv2 Int_short)->Boolean{return num(nv1)<=num(nv2)}
   operator "<=" (nv1 Integern,nv2 Int_int)->Boolean{return num(nv1)<=num(nv2)}
   operator "<=" (nv1 Integern,nv2 Int_long)->Boolean{return num(nv1)<=num(nv2)}
   operator "<=" (nv1 Integern,nv2 Int_long_long)->Boolean{return num(nv1)<=num(nv2)}
   operator "==" ( nv1 Integern, i Integer) -> Boolean {return num(nv1)= i }
   operator "==" ( nv1 Integern, nv2 Integern) -> Boolean {return num(nv1)= num(nv2)}
   operator "==" ( i Integer, nv1 Integern) -> Boolean {return i= num(nv1) }
   operator "==" ( nv1 Integern, nv2 Integer1) -> Boolean {return num(nv1)= num(nv2)}
   operator "==" ( nv1 Integern, nv2 Integer2) -> Boolean {return num(nv1)= num(nv2)}
   operator "==" ( nv1 Integern, nv2 Integer3) -> Boolean {return num(nv1)= num(nv2)}
   operator "==" ( nv1 Integern, nv2 Integer4) -> Boolean {return num(nv1)= num(nv2)}
   operator "==" (nv1 Integern,nv2 Int_char)->Boolean{return num(nv1)= num(nv2)}
   operator "==" (nv1 Integern,nv2 Int_short)->Boolean{return num(nv1)= num(nv2)}
   operator "==" (nv1 Integern,nv2 Int_int)->Boolean{return num(nv1)= num(nv2)}
   operator "==" (nv1 Integern,nv2 Int_long)->Boolean{return num(nv1)= num(nv2)}
   operator "==" (nv1 Integern,nv2 Int_long_long)->Boolean{return num(nv1)= num(nv2)}
   operator "!=" ( nv1 Integern, nv2 Integern) -> Boolean {return num(nv1)/=num(nv2)}
   operator "!=" ( nv1 Integern, i Integer) -> Boolean {return num(nv1)/=i}
   operator "!=" ( i Integer, nv1 Integern) -> Boolean {return i/=num(nv1)}
   operator "!=" ( nv1 Integern, nv2 Integer1) -> Boolean {return num(nv1)/=num(nv2)}
   operator "!=" ( nv1 Integern, nv2 Integer2) -> Boolean {return num(nv1)/=num(nv2)}
   operator "!=" ( nv1 Integern, nv2 Integer3) -> Boolean {return num(nv1)/=num(nv2)}
   operator "!=" ( nv1 Integern, nv2 Integer4) -> Boolean {return num(nv1)/=num(nv2)}
   operator "!=" (nv1 Integern,nv2 Int_char)->Boolean{return num(nv1)/=num(nv2)}
   operator "!=" (nv1 Integern,nv2 Int_short)->Boolean{return num(nv1)/=num(nv2)}
   operator "!=" (nv1 Integern,nv2 Int_int)->Boolean{return num(nv1)/=num(nv2)}
   operator "!=" (nv1 Integern,nv2 Int_long)->Boolean{return num(nv1)/=num(nv2)}
```

```
     operator "!=" (nv1 Integern,nv2 Int_long_long)->Boolean{return num(nv1)/=num(nv2)}
     operator ">"  ( nv1 Integern, nv2 Integern) -> Boolean {return num(nv1)> num(nv2)}
     operator ">"  ( nv1 Integern, i Integer) -> Boolean {return num(nv1)> i }
     operator ">"  ( i Integer, nv1 Integern) -> Boolean {return i> num(nv1) }
     operator ">"  ( nv1 Integern, nv2 Integer1) -> Boolean {return num(nv1)> num(nv2)}
     operator ">"  ( nv1 Integern, nv2 Integer2) -> Boolean {return num(nv1)> num(nv2)}
     operator ">"  ( nv1 Integern, nv2 Integer3) -> Boolean {return num(nv1)> num(nv2)}
     operator ">"  ( nv1 Integern, nv2 Integer4) -> Boolean {return num(nv1)> num(nv2)}
     operator ">"  (nv1 Integern,nv2 Int_char)->Boolean{return num(nv1)> num(nv2)}
     operator ">"  (nv1 Integern,nv2 Int_short)->Boolean{return num(nv1)> num(nv2)}
     operator ">"  (nv1 Integern,nv2 Int_int)->Boolean{return num(nv1)> num(nv2)}
     operator ">"  (nv1 Integern,nv2 Int_long)->Boolean{return num(nv1)> num(nv2)}
     operator ">"  (nv1 Integern,nv2 Int_long_long)->Boolean{return num(nv1)> num(nv2)}
     operator ">=" ( nv1 Integern, i Integer) -> Boolean {return num(nv1)>=i }
     operator ">=" ( i Integer, nv1 Integern) -> Boolean {return i>=num(nv1) }
     operator ">=" ( nv1 Integern, nv2 Integern) -> Boolean {return num(nv1)>=num(nv2)}
     operator ">=" ( nv1 Integern, nv2 Integer1) -> Boolean {return num(nv1)>=num(nv2)}
     operator ">=" ( nv1 Integern, nv2 Integer2) -> Boolean {return num(nv1)>=num(nv2)}
     operator ">=" ( nv1 Integern, nv2 Integer3) -> Boolean {return num(nv1)>=num(nv2)}
     operator ">=" ( nv1 Integern, nv2 Integer4) -> Boolean {return num(nv1)>=num(nv2)}
     operator ">=" (nv1 Integern,nv2 Int_char)->Boolean{return num(nv1)>=num(nv2)}
     operator ">=" (nv1 Integern,nv2 Int_short)->Boolean{return num(nv1)>=num(nv2)}
     operator ">=" (nv1 Integern,nv2 Int_int)->Boolean{return num(nv1)>=num(nv2)}
     operator ">=" (nv1 Integern,nv2 Int_long)->Boolean{return num(nv1)>=num(nv2)}
     operator ">=" (nv1 Integern,nv2 Int_long_long)->Boolean{return num(nv1)>=num(nv2)}
/* unary operators "+", "-" and "~" */
     operator "+" ( nv Integern )-> Integer { return num(nv);}
     operator "-" ( nv Integern )-> Integer { return 0-num(nv)}
     operator "~" ( nv Integern )-> Integer { return -(num(nv)+1)}
/* binary arithmetic operators */
     operator "*" ( nv Integern, i Integer ) -> Integer {return num(nv) * i}
     operator "/" ( nv Integern, i Integer ) -> Integer {return num(nv) / i}
     operator "%" ( nv Integern, i Integer ) -> Integer {return num(nv) rem i}
     operator "+" ( nv Integern, i Integer ) -> Integer {return num(nv) + i}
     operator "-" ( nv Integern, i Integer ) -> Integer {return num(nv) - i}
     operator "*" ( i Integer, nv Integern ) -> Integer {return i * num(nv)}
     operator "/" ( i Integer, nv Integern ) -> Integer {return i / num(nv)}
     operator "%" ( i Integer, nv Integern ) -> Integer {return i rem num(nv)}
     operator "+" ( i Integer, nv Integern ) -> Integer {return i + num(nv)}
     operator "-" ( i Integer, nv Integern ) -> Integer {return i - num(nv)}
     operator "*" ( nv1 Integern, nv2 Integern ) -> Integer {return num(nv1) * num(nv2)}
     operator "/" ( nv1 Integern, nv2 Integern ) -> Integer {return num(nv1) / num(nv2)}
     operator "%" ( nv1 Integern, nv2 Integern ) -> Integer {return num(nv1)rem num(nv2)}
     operator "+" ( nv1 Integern, nv2 Integern ) -> Integer {return num(nv1) + num(nv2)}
     operator "-" ( nv1 Integern, nv2 Integern ) -> Integer {return num(nv1) - num(nv2)}
     operator "*" ( nv1 Integern, nv2 Integer1 ) -> Integer {return num(nv1) * num(nv2)}
     operator "/" ( nv1 Integern, nv2 Integer1 ) -> Integer {return num(nv1) / num(nv2)}
     operator "%" ( nv1 Integern, nv2 Integer1 ) -> Integer {return num(nv1)rem num(nv2)}
     operator "+" ( nv1 Integern, nv2 Integer1 ) -> Integer {return num(nv1) + num(nv2)}
     operator "-" ( nv1 Integern, nv2 Integer1 ) -> Integer {return num(nv1) - num(nv2)}
     operator "*" ( nv1 Integern, nv2 Integer2 ) -> Integer {return num(nv1) * num(nv2)}
     operator "/" ( nv1 Integern, nv2 Integer2 ) -> Integer {return num(nv1) / num(nv2)}
     operator "%" ( nv1 Integern, nv2 Integer2 ) -> Integer {return num(nv1)rem num(nv2)}
     operator "+" ( nv1 Integern, nv2 Integer2 ) -> Integer {return num(nv1) + num(nv2)}
     operator "-" ( nv1 Integern, nv2 Integer2 ) -> Integer {return num(nv1) - num(nv2)}
     operator "*" ( nv1 Integern, nv2 Integer3 ) -> Integer {return num(nv1) * num(nv2)}
     operator "/" ( nv1 Integern, nv2 Integer3 ) -> Integer {return num(nv1) / num(nv2)}
     operator "%" ( nv1 Integern, nv2 Integer3 ) -> Integer {return num(nv1)rem num(nv2)}
     operator "+" ( nv1 Integern, nv2 Integer3 ) -> Integer {return num(nv1) + num(nv2)}
     operator "-" ( nv1 Integern, nv2 Integer3 ) -> Integer {return num(nv1) - num(nv2)}
     operator "*" ( nv1 Integern, nv2 Integer4 ) -> Integer {return num(nv1) * num(nv2)}
     operator "/" ( nv1 Integern, nv2 Integer4 ) -> Integer {return num(nv1) / num(nv2)}
     operator "%" ( nv1 Integern, nv2 Integer4 ) -> Integer {return num(nv1)rem num(nv2)}
     operator "+" ( nv1 Integern, nv2 Integer4 ) -> Integer {return num(nv1) + num(nv2)}
     operator "-" ( nv1 Integern, nv2 Integer4 ) -> Integer {return num(nv1) - num(nv2)}
     operator "*" (nv1 Integern,nv2 Int_char)->Integer{return num(nv1) * num(nv2)}
     operator "/" (nv1 Integern,nv2 Int_char)->Integer{return num(nv1) / num(nv2)}
     operator "%" (nv1 Integern,nv2 Int_char)->Integer{return num(nv1)rem num(nv2)}
     operator "+" (nv1 Integern,nv2 Int_char)->Integer{return num(nv1) + num(nv2)}
     operator "-" (nv1 Integern,nv2 Int_char)->Integer{return num(nv1) - num(nv2)}
     operator "*" (nv1 Integern,nv2 Int_short)->Integer{return num(nv1) * num(nv2)}
```

```
     operator "/" (nv1 Integern,nv2 Int_short)->Integer{return num(nv1) / num(nv2)}
     operator "%" (nv1 Integern,nv2 Int_short)->Integer{return num(nv1)rem num(nv2)}
     operator "+" (nv1 Integern,nv2 Int_short)->Integer{return num(nv1) + num(nv2)}
     operator "-" (nv1 Integern,nv2 Int_short)->Integer{return num(nv1) - num(nv2)}
     operator "*" (nv1 Integern,nv2 Int_int)->Integer{return num(nv1) * num(nv2)}
     operator "/" (nv1 Integern,nv2 Int_int)->Integer{return num(nv1) / num(nv2)}
     operator "%" (nv1 Integern,nv2 Int_int)->Integer{return num(nv1)rem num(nv2)}
     operator "+" (nv1 Integern,nv2 Int_int)->Integer{return num(nv1) + num(nv2)}
     operator "-" (nv1 Integern,nv2 Int_int)->Integer{return num(nv1) - num(nv2)}
     operator "*" (nv1 Integern,nv2 Int_long)->Integer{return num(nv1) * num(nv2)}
     operator "/" (nv1 Integern,nv2 Int_long)->Integer{return num(nv1) / num(nv2)}
     operator "%" (nv1 Integern,nv2 Int_long)->Integer{return num(nv1)rem num(nv2)}
     operator "+" (nv1 Integern,nv2 Int_long)->Integer{return num(nv1) + num(nv2)}
     operator "-" (nv1 Integern,nv2 Int_long)->Integer{return num(nv1) - num(nv2)}
     operator "*" (nv1 Integern,nv2 Int_long_long)->Integer{return num(nv1) * num(nv2)}
     operator "/" (nv1 Integern,nv2 Int_long_long)->Integer{return num(nv1) / num(nv2)}
     operator "%" (nv1 Integern,nv2 Int_long_long)->Integer{return num(nv1)rem num(nv2)}
     operator "+" (nv1 Integern,nv2 Int_long_long)->Integer{return num(nv1) + num(nv2)}
     operator "-" (nv1 Integern,nv2 Int_long_long)->Integer{return num(nv1) - num(nv2)}
/* shift left  other than "<<"(Integern,Integer) and "<<"(Integer,Integern)*/
     operator "<<" (nv1 Integern,nv2 Integern     )->Integer {return nv1 << num(nv2)}
     operator "<<" (nv1 Integern,nv2 Integer1     )->Integer {return nv1 << num(nv2)}
     operator "<<" (nv1 Integern,nv2 Integer2     )->Integer {return nv1 << num(nv2)}
     operator "<<" (nv1 Integern,nv2 Integer3     )->Integer {return nv1 << num(nv2)}
     operator "<<" (nv1 Integern,nv2 Integer4     )->Integer {return nv1 << num(nv2)}
     operator "<<" (nv1 Integern,nv2 Int_char     )->Integer {return nv1 << num(nv2)}
     operator "<<" (nv1 Integern,nv2 Int_short    )->Integer {return nv1 << num(nv2)}
     operator "<<" (nv1 Integern,nv2 Int_int      )->Integer {return nv1 << num(nv2)}
     operator "<<" (nv1 Integern,nv2 Int_long     )->Integer {return nv1 << num(nv2)}
     operator "<<" (nv1 Integern,nv2 Int_long_long)->Integer {return nv1 << num(nv2)}
/* shift right  other than ">>"(Integern,Integer) and ">>"(Integer,Integern)*/
     operator ">>" (nv1 Integern,nv2 Integern     )->Integer {return nv1 >> num(nv2)}
     operator ">>" (nv1 Integern,nv2 Integer1     )->Integer {return nv1 >> num(nv2)}
     operator ">>" (nv1 Integern,nv2 Integer2     )->Integer {return nv1 >> num(nv2)}
     operator ">>" (nv1 Integern,nv2 Integer3     )->Integer {return nv1 >> num(nv2)}
     operator ">>" (nv1 Integern,nv2 Integer4     )->Integer {return nv1 >> num(nv2)}
     operator ">>" (nv1 Integern,nv2 Int_char     )->Integer {return nv1 >> num(nv2)}
     operator ">>" (nv1 Integern,nv2 Int_short    )->Integer {return nv1 >> num(nv2)}
     operator ">>" (nv1 Integern,nv2 Int_int      )->Integer {return nv1 >> num(nv2)}
     operator ">>" (nv1 Integern,nv2 Int_long     )->Integer {return nv1 >> num(nv2)}
     operator ">>" (nv1 Integern,nv2 Int_long_long)->Integer {return nv1 >> num(nv2)}
/*   Negation and Comparisons with Boolean results */
     operator "!"  ( nv1 Integern) -> Integer
        return if num(nv1)= 0 then 1 else 0 fi}
     operator "<"  ( nv1 Integern, i Integer) -> Integer
        return if num(nv1)< i then 1 else 0 fi}
     operator "<"  ( nv1 Integern, nv2 Integern) -> Integer
        return if num(nv1)< num(nv2)then 1 else 0 fi}
     operator "<"  ( i Integer, nv1 Integern) -> Integer
        return if i < num(nv1)then 1 else 0 fi}
     operator "<"  ( nv1 Integern, nv2 Integer1) -> Integer
        return if num(nv1)< num(nv2)then 1 else 0 fi}
     operator "<"  ( nv1 Integern, nv2 Integer2) -> Integer
        return if num(nv1)< num(nv2)then 1 else 0 fi}
     operator "<"  ( nv1 Integern, nv2 Integer3) -> Integer
        return if num(nv1)< num(nv2)then 1 else 0 fi}
     operator "<"  ( nv1 Integern, nv2 Integer4) -> Integer
        return if num(nv1)< num(nv2)then 1 else 0 fi}
     operator "<"  (nv1 Integern,nv2 Int_char)->Integer
        return if num(nv1)< num(nv2)then 1 else 0 fi}
     operator "<"  (nv1 Integern,nv2 Int_short)->Integer
        return if num(nv1)< num(nv2)then 1 else 0 fi}
     operator "<"  (nv1 Integern,nv2 Int_int)->Integer
        return if num(nv1)< num(nv2)then 1 else 0 fi}
     operator "<"  (nv1 Integern,nv2 Int_long)->Integer
        return if num(nv1)< num(nv2)then 1 else 0 fi}
     operator "<"  (nv1 Integern,nv2 Int_long_long)->Integer
        return if num(nv1)< num(nv2)then 1 else 0 fi}
     operator "<=" ( nv1 Integern, nv2 Integern) -> Integer
        return if num(nv1)<=num(nv2)then 1 else 0 fi}
     operator "<=" ( nv1 Integern, i Integer) -> Integer
```

```
   return if num(nv1)<=i then 1 else 0 fi}
operator "<=" ( i Integer, nv1 Integern) -> Integer
   return if i<=num(nv1) then 1 else 0 fi}
operator "<=" ( nv1 Integern, nv2 Integer1) -> Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" ( nv1 Integern, nv2 Integer2) -> Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" ( nv1 Integern, nv2 Integer3) -> Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" ( nv1 Integern, nv2 Integer4) -> Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" (nv1 Integern,nv2 Int_char)->Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" (nv1 Integern,nv2 Int_short)->Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" (nv1 Integern,nv2 Int_int)->Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" (nv1 Integern,nv2 Int_long)->Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" (nv1 Integern,nv2 Int_long_long)->Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "==" ( nv1 Integern, i Integer) -> Integer
   return if num(nv1)= i then 1 else 0 fi}
operator "==" ( nv1 Integern, nv2 Integern) -> Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" ( i Integer, nv1 Integern) -> Integer
   return if i= num(nv1) then 1 else 0 fi}
operator "==" ( nv1 Integern, nv2 Integer1) -> Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" ( nv1 Integern, nv2 Integer2) -> Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" ( nv1 Integern, nv2 Integer3) -> Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" ( nv1 Integern, nv2 Integer4) -> Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" (nv1 Integern,nv2 Int_char)->Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" (nv1 Integern,nv2 Int_short)->Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" (nv1 Integern,nv2 Int_int)->Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" (nv1 Integern,nv2 Int_long)->Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" (nv1 Integern,nv2 Int_long_long)->Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "!=" ( nv1 Integern, nv2 Integern) -> Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" ( nv1 Integern, i Integer) -> Integer
   return if num(nv1)/=ithen 1 else 0 fi}
operator "!=" ( i Integer, nv1 Integern) -> Integer
   return if i/=num(nv1)then 1 else 0 fi}
operator "!=" ( nv1 Integern, nv2 Integer1) -> Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" ( nv1 Integern, nv2 Integer2) -> Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" ( nv1 Integern, nv2 Integer3) -> Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" ( nv1 Integern, nv2 Integer4) -> Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" (nv1 Integern,nv2 Int_char)->Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" (nv1 Integern,nv2 Int_short)->Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" (nv1 Integern,nv2 Int_int)->Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" (nv1 Integern,nv2 Int_long)->Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" (nv1 Integern,nv2 Int_long_long)->Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator ">"  ( nv1 Integern, nv2 Integern) -> Integer
   return if num(nv1)> num(nv2)then 1 else 0 fi}
```

```
     operator ">"  ( nv1 Integern, i Integer) -> Integer
        return if num(nv1)> i then 1 else 0 fi}
     operator ">"  ( i Integer, nv1 Integern) -> Integer
        return if i> num(nv1) then 1 else 0 fi}
     operator ">"  ( nv1 Integern, nv2 Integer1) -> Integer
        return if num(nv1)> num(nv2)then 1 else 0 fi}
     operator ">"  ( nv1 Integern, nv2 Integer2) -> Integer
        return if num(nv1)> num(nv2)then 1 else 0 fi}
     operator ">"  ( nv1 Integern, nv2 Integer3) -> Integer
        return if num(nv1)> num(nv2)then 1 else 0 fi}
     operator ">"  ( nv1 Integern, nv2 Integer4) -> Integer
        return if num(nv1)> num(nv2)then 1 else 0 fi}
     operator ">"  (nv1 Integern,nv2 Int_char)->Integer
        return if num(nv1)> num(nv2)then 1 else 0 fi}
     operator ">"  (nv1 Integern,nv2 Int_short)->Integer
        return if num(nv1)> num(nv2)then 1 else 0 fi}
     operator ">"  (nv1 Integern,nv2 Int_int)->Integer
        return if num(nv1)> num(nv2)then 1 else 0 fi}
     operator ">"  (nv1 Integern,nv2 Int_long)->Integer
        return if num(nv1)> num(nv2)then 1 else 0 fi}
     operator ">"  (nv1 Integern,nv2 Int_long_long)->Integer
        return if num(nv1)> num(nv2)then 1 else 0 fi}
     operator ">=" ( nv1 Integern, i Integer) -> Integer
        return if num(nv1)>=i then 1 else 0 fi}
     operator ">=" ( i Integer, nv1 Integern) -> Integer
        return if i>=num(nv1) then 1 else 0 fi}
     operator ">=" ( nv1 Integern, nv2 Integern) -> Integer
        return if num(nv1)>=num(nv2)then 1 else 0 fi}
     operator ">=" ( nv1 Integern, nv2 Integer1) -> Integer
        return if num(nv1)>=num(nv2)then 1 else 0 fi}
     operator ">=" ( nv1 Integern, nv2 Integer2) -> Integer
        return if num(nv1)>=num(nv2)then 1 else 0 fi}
     operator ">=" ( nv1 Integern, nv2 Integer3) -> Integer
        return if num(nv1)>=num(nv2)then 1 else 0 fi}
     operator ">=" ( nv1 Integern, nv2 Integer4) -> Integer
        return if num(nv1)>=num(nv2)then 1 else 0 fi}
     operator ">=" (nv1 Integern,nv2 Int_char)->Integer
        return if num(nv1)>=num(nv2)then 1 else 0 fi}
     operator ">=" (nv1 Integern,nv2 Int_short)->Integer
        return if num(nv1)>=num(nv2)then 1 else 0 fi}
     operator ">=" (nv1 Integern,nv2 Int_int)->Integer
        return if num(nv1)>=num(nv2)then 1 else 0 fi}
     operator ">=" (nv1 Integern,nv2 Int_long)->Integer
        return if num(nv1)>=num(nv2)then 1 else 0 fi}
     operator ">=" (nv1 Integern,nv2 Int_long_long)->Integer
        return if num(nv1)>=num(nv2)then 1 else 0 fi}
/* bitwise logical operators */
/*   operator "&" ( nv Integern, i Integer ) -> Integer; bitwise 'and'
     operator "^" ( nv Integern, i Integer ) -> Integer; bitwise 'xor'
     operator "|" ( nv Integern, i Integer ) -> Integer; bitwise 'or'
        For bitwise 'and','xor' and 'or' are defined for signed/unsigned integers.
        However, the other bitwise 'and'/'xor',/'or' operators are defined using the
        Here using the signatures above. Bitwise 'xor' is defined here.
*/
/*   Bitwise 'and' with signatures other than "&" (Integern, Integer ) */
     operator "&" ( i Integer, nv Integern ) -> Integer {return nv & i}
     operator "&" ( nv1 Integern, nv2 Integern ) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Integer1 ) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Integer2 ) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Integer3 ) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Integer4 ) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Int_char )->Integer{return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Int_short) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Int_Int  ) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Int_long ) -> Integer {return nv1 & num(nv2)}
     operator "&" (nv1 Integern,nv2 Int_long_long )->Integer{return nv1 & num(nv2)}
/*   Bitwise 'xor' with signatures other than "^" (Integern, Integer ) */
     operator "^" ( i Integer, nv Integern )-> Integer {return nv ^ i}
     operator "^" ( nv1 Integern, nv2 Integern ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Integer1 ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Integer2 ) -> Integer {return nv ^ num(nv2)}
```

```
     operator "^" ( nv1 Integern, nv2 Integer3 ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Integer4 ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Int_char ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Int_short) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Int_int  ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Int_long ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Int_long_long ) -> Integer {return nv ^ num(nv2)}
/*   Bitwise 'or' with signatures other than "|" (Integern, Integer ) */
     operator "|" ( i Integer, nv Integern ) -> Integer {return nv | i}
     operator "|" ( nv Integern, i Integer ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Integern ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Integer1 ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Integer2 ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Integer3 ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Integer4 ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Integern ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Int_char ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Int_short) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Int_int  ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Int_long ) -> Integer {return nv | num(nv2)}
     operator "|" (nv1 Integern, nv1 Int_long_long) -> Integer{return nv | num(nv2)}
/* Real/Float arithemetic */
     operator "+"(i this Integern, r  Real)->Real{return r+float(num(i))}
     operator "+"(r  Real, i this Integern)->Real{return r+float(num(i))}
     operator "+"(i this Integern, f Float)->Real{return to_Real(f)+float(num(i))}
     operator "+"(f Float, i this Integern)->Real{return to_Real(f)+float(num(i))}
     operator "-"(i this Integern, r  Real)->Real{return float(num(i))-r}
     operator "-"(r  Real, i this Integern)->Real{return r-float(num(i))}
     operator "-"(i this Integern, f Float)->Real{return float(num(i))-to_Real(f)}
     operator "-"(f Float, i this Integern)->Real{return to_Real(f)-float(num(i))}
     operator "*"(i this Integern, r  Real)->Real{return r*float(num(i))}
     operator "*"(r  Real, i this Integern)->Real{return r*float(num(i))}
     operator "*"(i this Integern, f Float)->Real{return to_Real(f)*float(num(i))}
     operator "*"(f Float, i this Integern)->Real{return to_Real(f)*float(num(i))}
     operator "/"(i this Integern, r  Real)->Real{return float(num(i))/r}
     operator "/"(r  Real, i this Integern)->Real{return r/float(num(i))}
     operator "/"(i this Integern, f Float)->Real{return float(num(i))/to_Real(f)}
     operator "/"(f Float, i this Integern)->Real{return to_Real(f)/float(num(i))}
     /* equal to Real/Float - Boolean and Integer result*/
     operator "=="(i this Integern, r  Real)->Boolean{return float(num(i))=r}
     operator "=="(r  Real, i this Integern)->Boolean{return r=float(num(i))}
     operator "=="(i this Integern, f Float)->Boolean{return float(num(i))=to_Real(f)}
     operator "=="(f Float, i this Integern)->Boolean{return to_Real(f)=float(num(i))}
     operator "=="(i this Integern, r  Real)->Integer{return if i==r then 1 else 0 fi}
     operator "=="(r  Real, i this Integern)->Integer{return if r==i then 1 else 0 fi}
     operator "=="(i this Integern, f Float)->Integer{return if i==f then 1 else 0 fi}
     operator "=="(f Float, i this Integern)->Integer{return if r==f then 1 else 0 fi}
/* not equal to Real/Float - Boolean and Integer result*/
     operator "!="(i this Integern, r  Real)->Boolean{return float(num(i))/=r}
     operator "!="(r  Real, i this Integern)->Boolean{return r/=float(num(i))}
     operator "!="(i this Integern, f Float)->Boolean{return float(num(i))/=to_Real(f)}
     operator "!="(f Float, i this Integern)->Boolean{return to_Real(f)/=float(num(i))}
     operator "!="(i this Integern, r  Real)->Integer{return if i!=r then 1 else 0 fi}
     operator "!="(r  Real, i this Integern)->Integer{return if r!=i then 1 else 0 fi}
     operator "!="(i this Integern, f Float)->Integer{return if i!=f then 1 else 0 fi}
     operator "!="(f Float, i this Integern)->Integer{return if r!=f then 1 else 0 fi}
/* less than Real/Float - Boolean and Integer result*/*/
     operator "<"(i this Integern, r  Real)->Boolean{return float(num(i))<r}
     operator "<"(r  Real, i this Integern)->Boolean{return r<float(num(i))}
     operator "<"(i this Integern, f Float)->Boolean{return float(num(i))<to_Real(f)}
     operator "<"(f Float, i this Integern)->Boolean{return to_Real(f)<float(num(i))}
     operator "<"(i this Integern, r  Real)->Integer{return if i<r then 1 else 0 fi}
     operator "<"(r  Real, i this Integern)->Integer{return if r<i then 1 else 0 fi}
     operator "<"(i this Integern, f Float)->Integer{return if i<f then 1 else 0 fi}
     operator "<"(f Float, i this Integern)->Integer{return if r<f then 1 else 0 fi}
/* less than or equal to Real/Float - Boolean and Integer result*/
     operator "<="(i this Integern, r  Real)->Boolean{return float(num(i))<=r}
     operator "<="(r  Real, i this Integern)->Boolean{return r<=float(num(i))}
```

```
     operator "<="(i this Integern, f Float)->Boolean{return float(num(i))<=to_Real(f)}
     operator "<="(f Float, i this Integern)->Boolean{return to_Real(f)<=float(num(i))}
     operator "<="(i this Integern, r  Real)->Integer{return if i<=r then 1 else 0 fi}
     operator "<="(r  Real, i this Integern)->Integer{return if r<=i then 1 else 0 fi}
     operator "<="(i this Integern, f Float)->Integer{return if i<=f then 1 else 0 fi}
     operator "<="(f Float, i this Integern)->Integer{return if r<=f then 1 else 0 fi}
/* greater than Real/Float - Boolean and Integer result*/*/
     operator ">"(i this Integern, r  Real)->Boolean{return float(num(i))>r}
     operator ">"(r  Real, i this Integern)->Boolean{return r>float(num(i))}
     operator ">"(i this Integern, f Float)->Boolean{return float(num(i))>to_Real(f)}
     operator ">"(f Float, i this Integern)->Boolean{return to_Real(f)>float(num(i))}
     operator ">"(i this Integern, r  Real)->Integer{return if i>r then 1 else 0 fi}
     operator ">"(r  Real, i this Integern)->Integer{return if r>i then 1 else 0 fi}
     operator ">"(i this Integern, f Float)->Integer{return if i>f then 1 else 0 fi}
     operator ">"(f Float, i this Integern)->Integer{return if r>f then 1 else 0 fi}
/* greater than or equal to Real/Float - Boolean and Integer result*/*/
     operator ">="(i this Integern, r  Real)->Boolean{return float(num(i))>=r}
     operator ">="(r  Real, i this Integern)->Boolean{return r>=float(num(i))}
     operator ">="(i this Integern, f Float)->Boolean{return float(num(i))>=to_Real(f)}
     operator ">="(f Float, i this Integern)->Boolean{return to_Real(f)>=float(num(i))}
     operator ">="(i this Integern, r  Real)->Integer{return if i>=r then 1 else 0 fi}
     operator ">="(r  Real, i this Integern)->Integer{return if r>=i then 1 else 0 fi}
     operator ">="(i this Integern, f Float)->Integer{return if i>=f then 1 else 0 fi}
     operator ">="(f Float, i this Integern)->Integer{return if r>=f then 1 else 0 fi}
/* Prefix increment/decrement */
     method "++"   -> Integer {this := <<type Integern>>integer(this+1); return this }
     method "--"   -> Integer {this := <<type Integern>>integer(this-1); return this }
/* Postfix increment/decrement */
     method postfix_inc-> r Integer
     {  r:=num(this);
        this:=<<type Integern>>integer(r+1);
        return r
     }
     method postfix_dec-> r Integer
     {  r:=num(this);
        this:=<<type Integern>>integer(r-1);
        return r
     }
/* simple assignment */
     method "="   (i Integern     ) -> Integer {this:=i;              return num(this)}
     method "="   (i Integer      ) -> Integer {this:=integer(i);     return num(this)}
     method "="   (i Integer1     ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Integer2     ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Integer3     ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Integer4     ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Int_char     ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Int_short    ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Int_int      ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Int_long     ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Int_long_long) -> Integer {this:=integer(num(i));return num(this)}
/* multiplication assignment */
     method "*="  (i Integern     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Integer      ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Integer1     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Integer2     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Integer3     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Integer4     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Int_char     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Int_short    ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Int_int      ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Int_long     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Int_long_long) -> Integer {this:=this  * i; return num(this)}
/* division assignment */
     method "/="  (i Integern     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Integer      ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Integer1     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Integer2     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Integer3     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Integer4     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Int_char     ) -> Integer {this:=this  / i; return num(this)}
```

```
     method "/="  (i Int_short    ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Int_int      ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Int_long     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Int_long_long) -> Integer {this:=this  / i; return num(this)}
/* remainder assignment */
     method "%="  (i Integern     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Integer      ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Integer1     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Integer2     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Integer3     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Integer4     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Int_char     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Int_short    ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Int_int      ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Int_long     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Int_long_long) -> Integer {this:=this  % i; return num(this)}
/* addition assignment */
     method "+="  (i Integern     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Integer      ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Integer1     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Integer2     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Integer3     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Integer4     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Int_char     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Int_short    ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Int_int      ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Int_long     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Int_long_long) -> Integer {this:=this  + i; return num(this)}
/- subtraction assignment -/
     method "-="  (i Integern     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Integer      ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Integer1     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Integer2     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Integer3     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Integer4     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Int_char     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Int_short    ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Int_int      ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Int_long     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Int_long_long) -> Integer {this:=this  - i; return num(this)}
/* shift left assignment */
     method "<<=" (i Integern     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Integer      ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Integer1     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Integer2     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Integer3     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Integer4     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Int_char     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Int_short    ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Int_int      ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Int_long     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Int_long_long) -> Integer {this:=this  << i; return num(this)}
/* shift right assignment */
     method ">>=" (i Integern     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Integer      ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Integer1     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Integer2     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Integer3     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Integer4     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Int_char     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Int_short    ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Int_int      ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Int_long     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Int_long_long) -> Integer {this:=this  >> i; return num(this)}
/* bitwise and assignment */
     method "&="  (i Integern     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Integer      ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Integer1     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Integer2     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Integer3     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Integer4     ) -> Integer {this:=this  & i; return num(this)}
```

```
     method "&="  (i Int_char     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Int_short    ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Int_int      ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Int_long     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Int_long_long) -> Integer {this:=this  & i; return num(this)}
/* bitwise xor assignment */
     method "^="  (i Integern     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Integer      ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Integer1     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Integer2     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Integer3     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Integer4     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Int_char     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Int_short    ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Int_int      ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Int_long     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Int_long_long) -> Integer {this:=this  ^ i; return num(this)}
/* bitwise or assignment */
     method "|="  (i Integern     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Integer      ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Integer1     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Integer2     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Integer3     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Integer4     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Int_char     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Int_short    ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Int_int      ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Int_long     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Int_long_long) -> Integer {this:=this  | i; return num(this)}
     default integer(0);
} /* end parameterized value type Integern;
```

C.1.6.2.2 Integers for a signed integer type with n bits

The parameters `n`, `Integer1`, `Integer2`, `Integer3` and `Integer4` from `Integern` are left unbound. Each signed integer type is defined using `Integern` with `n` bound to the number of bits, and the other context parameters bound to the other C signed integer types.

```
*/
abstract value type Integers inherits Integern
     <  /* Int_char      = */ Unsigned_char,
        /* Int_short     = */ Unsigned_short,
        /* Int_int       = */ Unsigned_int
        /* Int_long      = */ Unsigned_long,
        /* Int_long_long = */ Unsigned_long_long >
     /* this leaves the following parameters unbound:
        synonym n Natural;
        value type Integer1; value type Integer2;
        value type Integer3; value type Integer4 */
{
     operator integer ( i Integer )-> this Integers
     synonym p Integer = power(2,n-1);
     { /* integer(i) of Integers is the unique constructor for Integers values. */
        return if i > p-1 then integer((i - (i/p)*p)
               /* reduce to integer in range of Integers */
        else
           if i < -p then integer(i - ((i+1)/p)*p)
                 /* increase to integer in range of Integers */
           else integer(i) /* which represents the Integers value */
           fi
        fi}
     operator num ( nv this Integers )-> Integer {
     /* return the integer value for which integer(num(nv)) = nv */
     /* Although this is expressed here as an iterative algorithm,
        it is expected that the implementation does the conversion
        as an atomic action,
        or null action if the effect is to just to do a type conversion. */
        return if integer(0) = nv then 0
```

```
      else
         if (nv > integer (0)) then 1 + num(nv-<<type Integers>>integer(1));
         else /*(nv < integer (0))*/ -1 + num(nv+<<type Integers>>integer(1))
         fi
      fi}
/* shift left */
   operator "<<" ( nv Integers, i Integer ) -> Integer
   /* sign bit propagated left for << */
      {return num(nv) * power(2, if i > 0 then i else 0 fi)}
   operator "<<" ( i Integer, nv Integers ) -> Integer
      {return i * power(2, if num(nv) > 0 then num(nv) else 0 fi)}
/* shift right */
   operator ">>"( nv Integers, i Integer ) -> Integer {
   /* In clause 6.5.7 of [b-ISO/IEC 9899] for a negative number
      the resulting value is implementation-defined.
      A negative number is -power(2,n) for the sign bit
      plus power(2,m) for each of other 1 bit
      where the bits are numbered m to 0 (most to least significant). */
      return if i <= 0 then num(nv)
      else
         if num(nv) > 0 then num(nv)/power(2,i)
         else /* num(nv) < 0 */ (num(nv)+power(2,n))/ power(2,i)
         fi
      fi}
   operator ">>"( i Integer, nv Integers ) -> Integer {
      return if num(nv) <= 0 then i
      else
         if i >= 0 then i/power(2,num(nv))
         else /* i < 0 */
            (num(<<type Integers>>integer(i))+power(2,n))/ power(2,num(nv));
         fi
      fi}
/* bit wise and */
   operator private bitwise_and ( i1, i2, m Integer ) -> Integer
   dcl p, r1, r2 Integer; /* least significant bit is bit 1 */
   {  if (m=1) return (i1 > 0) and (i2 > 0) then 1 else 0;
      p := power(2,m-1);/*value of bit m */
      r1 = i1 rem p;
      r2 = i2 rem p;
      /* bit m is 1 in i1 if i1-r1 > 0 */
      /* bit m is 1 in i2 if i2-r2 > 0 */
      return if ((i1-r1) > 0) and ((i2-r2) > 0) then p else 0 fi +
         bitwise_and (r1, r2, m-1)
   }
   operator "&" ( nv Integers, i Integer ) -> Integer
   synonym p Integer = power(2,n);
   dcl nvi Integer;
   {  nvi := num(nv);
      i := num(<<type Integers>>integer(i));
      return if (nvi < 0) and (i <0) then p else 0 /* the sign bit */ fi
         /* add the other bits, in each case removing the sign */
         + bitwise_and( nvi + if nvi < 0 then p else 0 fi,
               i + if i < 0 then p else 0 fi, n-2)
   }
/* bitwise xor*/
   operator private bitwise_xor ( i1, i2, m Integer ) -> Integer
   dcl p, r1, r2 Integer; /* least significant bit is bit 1 */
   {  if (m=1) return if ((i1 > 0) xor (i2 > 0)) then 1 else 0 fi
      p := power(2,m-1); /*value of bit m */
      r1 = i1 rem p;
      r2 = i2 rem p;
      /* bit m is 1 in i1 if i1-r1 > 0 */
      /* bit m is 1 in i2 if i2-r2 > 0 */
      return if ((i1-r1) > 0) xor ((i2-r2) > 0)  then p else 0 fi +
         bitwise_xor (r1, r2, m-1)
   }
   operator "^" ( nv Integers, i Integer ) -> Integer
   synonym p Integer = power(2,n-1);
   dcl nvi, i_only_bits Integer;
   {  nvi := num(nv);
      i_only_bits := i
```

```
         /* subtract bits in range to neutralise these*/
          - num(<<type Integers>>integer(i));
      i := num(<<type Integers>>integer(i));
      return  i_only_bits +
              if (nvi < 0) xor (i <0 )
              then p else 0 /* the sign bit */
              fi + /* add the other bits, in each case removing the sign */
              bitwise_xor( nvi + if nvi < 0 then p else 0 fi,
               i+ if i < 0 then p else 0 fi, n-1);
   }
/* bitwise or*/
   operator private bitwise_or ( i1, i2, m Integer ) -> Integer
   dcl p, r1, r2 Integer; /* least significant bit is bit 1 */
   {  if (m=1) return if ((i1 > 0) or (i2 > 0)) then 1 else 0 fi;
      p := power(2,m-1); /*value of bit m*/
      r1 = i1 rem p;
      r2 = i2 rem p;
      /* bit m is 1 in i1 if i1-r1 > 0 */
      /* bit m is 1 in i2 if i2-r2 > 0 */
      return if ((i1-r1) > 0) or ((i2-r2) > 0)  then p else 0 fi
        + bitwise_or (r1, r2, m-1)
   }
   operator "|" ( nv Integers, i Integer ) -> Integer
   synonym p Integer = power(2,n);
   dcl nvi, i_only_bits Integer;
   {  nvi := num(nv);
      i_only_bits := i
         /* subtract bits in range to neutralise these*/
          - num(<<type Integers>>integer(i));
      i := num(<<type Integers>>integer(i));
      return  i_only_bits +
              if (nvi < 0) or (i <0 )
              then p else 0 /* the sign bit */
              fi + /* add the other bits, in each case removing the sign */
              bitwise_or( nvi + if nvi < 0 then p else 0 fi,
               i+ if i < 0 then p else 0 fi, n-1);
   }
} /* end parameterized value type Integers;
```

C.1.6.2.3 Integeru for an unsigned integer type with n bits

The parameters `n`, `Integer1`, `Integer2`, `Integer3` and `Integer4` from `Integern` are left unbound. Each signed integer type is defined using `Integeru` with `n` bound to the number of bits, and the other context parameters bound to the other C unsigned integer types.

```
*/
abstract value type Integeru inherits Integern
   <  /* Int_char      = */ Signed_char,
      /* Int_short     = */ Signed_short,
      /* Int_int       = */ Signed_int
      /* Int_long      = */ Signed_long,
      /* Int_long_long = */ Signed_long_long >
   /* this leaves the following parameters unbound:
      synonym n Natural;
      value type Integer1; value type Integer2;
      value type Integer3; value type Integer4 */
{
   operator integer ( i Integer )-> this Integeru
   synonym p Integer = power(2,n-1);
      /* integer(i) of Integeru is the unique constructor for Integeru values. */
   {  return  if i > p-1 then integer((i - (i/p)*p)
              /* reduce to integer in range of Integeru */
      else
         if i < -p then integer(i - ((i+1)/p)*p)
           /* increase to integer in range of Integeru */
         else integer(i) /* which represents the Integeru value */
         fi
      fi}
```

```
     operator num ( nv this Integeru )-> Natural {
     /* return the positive integer value for which integer(num(nv)) = nv */
     /* Although this is expressed here as an iteration algorithm,
        it is expected that the implementation does the conversion
        as an atomic action,
        or null action if the effect is to just to do a type conversion. */
        if (integer(0) = nv) then return 0;
        return 1 + num(nv-<<type Integeru>>integer(1))
     }
/* shift left */
     operator "<<" ( nv Integeru, i Integer ) -> Integer
        {return num(nv) * power(2, if i > 0 then i else 0 fi)}
     operator "<<" ( i Integer, nv Integeru ) -> Integer {return i * power(2 num(nv))}
/* shift right */
     operator ">>"( nv Integeru, i Integer ) -> Integer
        {return if i <= 0 then num(nv) else num(nv)/power(2,i) fi}
     operator ">>"( i Integer, nv Integeru ) -> Integer {
        return if i >= 0
        then i/power(2,num(nv))
        else (num(<<type Integeru>>integer(i))+power(2,n))/ power(2,num(nv))fi
     }
/*bitwise and*/
     operator private bitwise_and ( i1, i2, m Integer ) -> Integer
     dcl p, r1, r2 Integer; /* least significant bit is bit 1 */
     {  if (m=1) return if (i1 > 0) and (i2 > 0) then 1 else 0 fi;
        p := power(2,m-1); /*value of bit m */
        r1 = i1 rem p;
        r2 = i2 rem p;
        /* bit m-1 is 1 in i1 if i1-r1 > 0 */
        /* bit m-1 is 1 in i2 if i2-r2 > 0 */
        return if ((i1-r1) > 0) and ((i2-r2) > 0) then p else 0 fi
           + bitwise_and (r1, r2, m-1)
     }
     operator "&" ( nv Integeru, i Integer ) -> Integer;
        {return bitwise_and(num(nv), num(<<type Integeru>>integer(i)), n)}
/* bitwise xor*/
     operator private bitwise_xor ( i1, i2, m Integer ) -> Integer
     dcl p, r1, r2 Integer; /* least significant bit is bit 1 */
     {  if (m=1) return if ((i1 > 0) xor (i2 > 0)) then 1 else 0 fi;
        p := power(2,m-1); /*value of bit m */
        r1 = i1 rem p;
        r2 = i2 rem p;
        /* bit m is 1 in i1 if i1-r1 > 0 */
        /* bit m is 1 in i2 if i2-r2 > 0 */
        return if ((i1-r1) > 0) xor ((i2-r2) > 0)  then p else 0 fi
           + bitwise_xor (r1, r2, m-1)
     }
     operator "^" ( nv Integeru, i Integer ) -> Integer;
     dcl i_only_bits Integer;
     {  i_only_bits := i
           /* subtract bits in range to neutralise these*/
            - num(<<type Integeru>>integer(i));
        i := num(<<type Integeru>>integer(i));
        return  i_only_bits +  bitwise_xor(num(nv), i, n);
     }
/* bitwise or*/
     operator private bitwise_or ( i1, i2, m Integer ) -> Integer
     dcl p, r1, r2 Integer;
     {  if (m=1) return if ((i1 > 0) or (i2 > 0)) then 1 else 0 fi;
        p := power(2,m-1); /*value of bit m, least significant bit is bit 1*/
        r1 = i1 rem p;
        r2 = i2 rem p;
        /* bit m is 1 in i1 if i1-r1 > 0 */
        /* bit m is 1 in i2 if i2-r2 > 0 */
        return if ((i1-r1) > 0) or ((i2-r2) > 0)  then p else 0 fi
           + bitwise_or (r1, r2, m-1)
     }
     operator "|" ( nv Integeru, i Integer ) -> Integer;
     dcl i_only_bits Integer;
     {  i_only_bits := i
           /* subtract bits in range to neutralise these*/
```

```
            - num(<<type Integeru>>integer(i));
        i := num(<<type Integeru>>integer(i));
        return  i_only_bits + bitwise_or(num(nv), i, n);
    }
} /* end parameterized value type Integeru;
```

C.1.6.3 Types for void and pointers

The `Void` type comprises an empty set of values; it is an incomplete type that cannot be completed. The `Void` type is denoted by a <c type keywords> item **void**. It is not allowed to define a variable or parameter with the type `Void`.

```
*/
abstract value type Void
     /* Has an implicit literal signature denoted by null – see [ITU-T Z.107 12.1] */
endvalue type Void;
synonym sizeof_Void Integer = 0;
/*
```

The `Star_void` type represents a 'pointer to `Void`' (an untyped pointer) such as that introduced by a <c declaration> of the form:

```
void *id;
```

The operators `Star_void_to_Integer` and `Integer_to_Star_void` of `Star_void` convert to and from `Integer`. It is required that:

```
Star_void_to_Integer (Integer_to_Star_void (i)) == i
```
and
```
Integer_to_Star_void (Star_void_to_Integer (ptr) == ptr
```

Arithmetic operators are not defined for `Star_void` because the size of the pointer is not known.

The dereference operator is not defined for `Star_void` because the actual type of the pointer is not known and therefore the type of the result is not known.

```
*/
value type Star_void {
     /* Has an implicit literal signature denoted by null – see [ITU-T Z.107 12.1] */
     operator Star_void_to_Integer (ptr Star_void) -> Integer external;
     operator Integer_to_Star_void (i Integer) -> Star_void external;
     default null;
} /* end value type Star_void; */
synonym sizeof_Star_void Integer = external /*Depends on the architecture.*/
/*
```

The parameterized type `Star_type` with a type as an actual parameter defines the typed pointer type for the type.

It is possible to convert a `Star_void` (a pointer to **`void`**) to or from a pointer of any type. If a pointer to a type is converted to a `Star_void` (a pointer to **`void`**) and back again, the result compares equal to the original pointer. The operators to convert to each type from `Star_void` and from each type to `Star_void` are defined in the typed pointer type for the type.

The operators `Star_type_to_Integer` and `Integer_to_Star_type` of `Star_type` convert to and from `Integer`. It is required that:

```
Star_type_to_Integer(Integer_to_Star_type(i))==i
```
and
```
Integer_to_Star_type(Star_type_to_Integer(p)==p
```
and
```
Star_void_to_Integer(Star_type_to_Star_void(p))==Star_type_to_Integer(p)
```
and
```
Star_void_to_Star_type(Integer_to_Star_void(i))== Integer_to_Star_type(i)
*/
abstract value type Star_type <type Atype, synonym sizeof_type Integer>{
     operator "&" (in/out x Atype) -> Star_type /* address of */
        {return Star_void_to_Star_type(<<type Atype>>"&"(x)}
     operator "*" (p Star_type) -> Atype external; /*dereference the Star_type */
     operator Star_type_to_Integer (p Star_type) -> Integer external;
     operator Integer_to_Star_type (i Integer) -> Star_type external;
```

```
     operator Star_type_to_Star_void (p Star_type) -> Star_void
        { return Integer_to_Star_void(Star_type_to_Integer(p)) }
     operator Star_void_to_Star_type (p Star_void) -> Star_type
        { return Integer_to_Star_type(Star_void_to_Integer(p))}
     operator "+" (p Star_type, i Integer) -> Star_type /* pointer arithmetic - add */
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Signed_char   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Signed_short  ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Signed_int    ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Signed_long   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Signed_long_long) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Unsigned_char   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Unsigned_short  ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Unsigned_int    ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Unsigned_long   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Unsigned_long_long) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "-" (p Star_type, i Integer) -> Star_type /* pointer arithmetic - sub */
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Signed_char   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Signed_short  ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Signed_int    ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Signed_long   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Signed_long_long) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Unsigned_char   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Unsigned_short  ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Unsigned_int    ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Unsigned_long   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Unsigned_long_long) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     method "++"  -> Star_type {this := this+1; return this }
     method "--"  -> Star_type {this := this-1; return this }
     method postfix_inc-> r Star_type {r:= this; this:= r+1; return r }
     method postfix_dec-> r Star_type {r:= this; this:= r-1; return r }
     /* simple assignment */
     method "=" (p Star_type)->Star_type {this:=p;return this}
     method "=" (i Integer)->Star_type{this:=Integer_to_Star_type(i);return this}
     method "=" (i Signed_char)->Star_type{this:=Integer_to_Star_type(num(i));return
this}
     method "=" (i Signed_short)->Star_type
        {this:=Integer_to_Star_type(num(i));return this}
     method "=" (i Signed_int)->Star_type{this:=Integer_to_Star_type(num(i));return this}
     method "=" (i Signed_long)->Star_type{this:=Integer_to_Star_type(num(i));return
this}
     method "=" (i Signed_long_long)->Star_type
        {this:=Integer_to_Star_type(num(i));return this}
     method "=" (i Unsigned_char)->Star_type
        {this:=Integer_to_Star_type(num(i));return this}
     method "=" (i Unsigned_short)->Star_type
        {this:=Integer_to_Star_type(num(i));return this}
     method "=" (i Unsigned_int)->Star_type
        {this:=Integer_to_Star_type(num(i));return this}
     method "=" (i Unsigned_long)->Star_type
```

```
      {this:=Integer_to_Star_type(num(i));return this}
   method "=" (i Unsigned_long_long)->Star_type
      {this:=Integer_to_Star_type(num(i));return this}
   /* addition assignment */
   method "+=" (i Integer)            -> Star_type {this:=this + i; return this}
   method "+=" (i Signed_char)        -> Star_type {this:=this + i; return this}
   method "+=" (i Signed_short)       -> Star_type {this:=this + i; return this}
   method "+=" (i Signed_int)         -> Star_type {this:=this + i; return this}
   method "+=" (i Signed_long)        -> Star_type {this:=this + i; return this}
   method "+=" (i Signed_long_long)   -> Star_type {this:=this + i; return this}
   method "+=" (i Unsigned_char)      -> Star_type {this:=this + i; return this}
   method "+=" (i Unsigned_short)     -> Star_type {this:=this + i; return this}
   method "+=" (i Unsigned_int)       -> Star_type {this:=this + i; return this}
   method "+=" (i Unsigned_long)      -> Star_type {this:=this + i; return this}
   method "+=" (i Unsigned_long_long) -> Star_type {this:=this + i; return this}
   /* subtraction assignment */
   method "-=" (i Integer)            -> Star_type {this:=this - i; return this}
   method "-=" (i Signed_char)        -> Star_type {this:=this - i; return this}
   method "-=" (i Signed_short)       -> Star_type {this:=this - i; return this}
   method "-=" (i Signed_int)         -> Star_type {this:=this - i; return this}
   method "-=" (i Signed_long)        -> Star_type {this:=this - i; return this}
   method "-=" (i Signed_long_long)   -> Star_type {this:=this - i; return this}
   method "-=" (i Unsigned_char)      -> Star_type {this:=this - i; return this}
   method "-=" (i Unsigned_short)     -> Star_type {this:=this - i; return this}
   method "-=" (i Unsigned_int)       -> Star_type {this:=this - i; return this}
   method "-=" (i Unsigned_long)      -> Star_type {this:=this - i; return this}
   method "-=" (i Unsigned_long_long) -> Star_type {this:=this - i; return this}
   /* Other Assignment methods not defined for pointers. */
   default null;
}/* end parameterized value type Star_type
*/
```

C.1.6.4 C signed and unsigned char types

A C char is an 8-bit integer, also used to represent character.

```
*/
value type Signed_char
   inherits Integers <
      /*n*/ CHAR_BIT /*external, usually 8 */,
      /*Integer1*/ Signed_short,
      /*Integer2*/ Signed_int,
      /*Integer3*/ Signed_long,
      /*Integer4*/ Signed_long_long >
      (to_Signed_char = integer );
synonym sizeof_Signed_char Integer = (CHAR_BIT+7)/8 /*usually 1*/;
value type Star_Signed_char
   inherits Star_type < Signed_char, sizeof_Signed_char>
   {  Star_Signed_char_to_Star_void = Star_type_to_Star_void,
      Star_void_to_Star_Signed_char = Star_void_to_Star_type,
      Star_Signed_char_to_Integer = Star_type_to_Integer,
      Integer_to_Star_Signed_char = Integer_to_Star_type);
synonym sizeof_Star_Signed_char Integer = sizeof_Star_void;
value type Unsigned_char
   inherits Integeru <
      /*n*/ CHAR_BIT /*external, usually 8 */,
      /*Integer1*/ Unsigned_short,
      /*Integer2*/ Unsigned_int,
      /*Integer3*/ Unsigned_long,
      /*Integer4*/ Unsigned_long_long >
      (to_Unsigned_char = integer );
synonym sizeof_Unsigned_char Integer = sizeof_Signed_char;
value type Star_Unsigned_char
   inherits Star_type < Unsigned_char, sizeof_Unsigned_char>
   {  Star_Unsigned_char_to_Star_void = Star_type_to_Star_void,
      Star_void_to_Star_Unsigned_char = Star_void_to_Star_type,
      Star_Unsigned_char_to_Integer = Star_type_to_Integer,
      Integer_to_Star_Unsigned_char = Integer_to_Star_type)
```

```
{
     operator Star_Unsigned_char_to_Charstring (sc Star_Unsigned_char)-> cs
Charstring
     {  cs := ""; /*empty string*/
        loop ("*"(sc)<>0) /* while character is not NUL */
        {  cs := cs//mkstring(chr(num("*"(sc))); /* convert and add to result */
           sc := (Star_Unsigned_char_Integer(sc) + sizeof_Signed_char); /*next char
*/
        } /* end of loop body*/
        return cs
     } /*end of Star_Unsigned_char_to_Charstring*/
     operator length(sc Star_Unsigned_char)-> Integer
     { return length(Star_Unsigned_char_to_Charstring(sc))}
} /*end of Star_Unsigned_char
synonym sizeof_Star_Unsigned_char Integer = sizeof_Star_void;
/*
```

C.1.6.5 C signed and unsigned short types

A C short is a 16-bit integer.

```
*/
value type Signed_short
     inherits Integers <
        /*n*/ SHRT_BIT /*external, usually 16 */,
        /*Integer1*/ Signed_char,
        /*Integer2*/ Signed_int,
        /*Integer3*/ Signed_long,
        /*Integer4*/ Signed_long_long >
        (to_Signed_short = integer );
synonym sizeof_Signed_short Integer = (SHRT_BIT+7)/8 /*usually 2*/;
value type Star_Signed_short
     inherits Star_type < Signed_short, sizeof_Signed_short>
     {  Star_Signed_short_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Signed_short = Star_void_to_Star_type,
        Star_Signed_short_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Signed_short = Integer_to_Star_type);
synonym sizeof_Star_Signed_short Integer = sizeof_Star_void;
value type Unsigned_short
     inherits Integeru <
        /*n*/ SHRT_BIT /*external, usually 16 */,
        /*Integer1*/ Unsigned_char,
        /*Integer2*/ Unsigned_int,
        /*Integer3*/ Unsigned_long,
        /*Integer4*/ Unsigned_long_long >
        (to_Unsigned_short = integer );
synonym sizeof_Unsigned_short Integer = sizeof_Signed_short;
value type Star_Unsigned_short
     inherits Star_type < Unsigned_short, sizeof_Unsigned_short>
     {  Star_Unsigned_short_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Unsigned_short = Star_void_to_Star_type,
        Star_Unsigned_short_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Unsigned_short = Integer_to_Star_type);
synonym sizeof_Star_Unsigned_short Integer = sizeof_Star_void;
/*
```

C.1.6.6 C signed and unsigned int types

A C int is a minimum of 16 bits.

```
*/
value type Signed_int
     inherits Integers <
        /*n*/ INT_BIT /*external, minimum 16 - same as short */,
        /*Integer1*/ Signed_char,
        /*Integer2*/ Signed_short,
        /*Integer3*/ Signed_long,
        /*Integer4*/ Signed_long_long >
        (to_Signed_int = integer );
synonym sizeof_Signed_int Integer = (INT_BIT+7)/8 /*minimum 2*/;
```

```
value type Star_Signed_int
      inherits Star_type < Signed_int, sizeof_Signed_int>
      {  Star_Signed_int_to_Star_void = Star_type_to_Star_void,
         Star_void_to_Star_Signed_int = Star_void_to_Star_type,
         Star_Signed_int_to_Integer = Star_type_to_Integer,
         Integer_to_Star_Signed_int = Integer_to_Star_type);
synonym sizeof_Star_Signed_int Integer = sizeof_Star_void;
value type Unsigned_int
      inherits Integeru <
         /*n*/ INT_BIT /*external, minimum 16 */,
         /*Integer1*/ Unsigned_char,
         /*Integer2*/ Unsigned_short,
         /*Integer3*/ Unsigned_long,
         /*Integer4*/ Unsigned_long_long >
         (to_Unsigned_int = integer );
synonym sizeof_Unsigned_int Integer = sizeof_Signed_int;
value type Star_Unsigned_int
      inherits Star_type < Unsigned_int, sizeof_Unsigned_int>
      {  Star_Unsigned_int_to_Star_void = Star_type_to_Star_void,
         Star_void_to_Star_Unsigned_int = Star_void_to_Star_type,
         Star_Unsigned_int_to_Integer = Star_type_to_Integer,
         Integer_to_Star_Unsigned_int = Integer_to_Star_type);
synonym sizeof_Star_Unsigned_int Integer = sizeof_Star_void;
/*
```

C.1.6.7 C signed and unsigned long types

A C long is a minimum of 32 bits.

```
*/
value type Signed_long
      inherits Integers <
         /*n*/ LONG_BIT /*external, minimum 32 – twice int */,
         /*Integer1*/ Signed_char,
         /*Integer2*/ Signed_short,
         /*Integer3*/ Signed_int,
         /*Integer4*/ Signed_long_long >
         (to_Signed_long = integer );
synonym sizeof_Signed_long Integer = (LONG_BIT+7)/8 /*minimum 4*/;
value type Star_Signed_long
      inherits Star_type < Signed_long, sizeof_Signed_long>
      {  Star_Signed_long_to_Star_void = Star_type_to_Star_void,
         Star_void_to_Star_Signed_long = Star_void_to_Star_type,
         Star_Signed_long_to_Integer = Star_type_to_Integer,
         Integer_to_Star_Signed_long = Integer_to_Star_type);
synonym sizeof_Star_Signed_long Integer = sizeof_Star_void;
value type Unsigned_long
      inherits Integeru <
         /*n*/ LONG_BIT /*external, minimum 32 */,
         /*Integer1*/ Unsigned_char,
         /*Integer2*/ Unsigned_short,
         /*Integer3*/ Unsigned_int,
         /*Integer4*/ Unsigned_long_long >
         (to_Unsigned_long = integer );
synonym sizeof_Unsigned_long Integer = sizeof_Signed_ilong;
value type Star_Unsigned_long
      inherits Star_type < Unsigned_long, sizeof_Unsigned_long>
      {  Star_Unsigned_long_to_Star_void = Star_type_to_Star_void,
         Star_void_to_Star_Unsigned_long = Star_void_to_Star_type,
         Star_Unsigned_long_to_Integer = Star_type_to_Integer,
         Integer_to_Star_Unsigned_long = Integer_to_Star_type);
synonym sizeof_Star_Unsigned_long Integer = sizeof_Star_void;
/*
```

C.1.6.8 C signed and unsigned long long types

A C long long is a minimum of 64 bits.

```
*/
value type Signed_long_long
     inherits Integers <
        /*n*/ LLONG_BIT /*external, minimum 64 – twice long */,
        /*Integer1*/ Signed_char,
        /*Integer2*/ Signed_short,
        /*Integer3*/ Signed_int,
        /*Integer4*/ Signed_long >
        (to_Signed_long_long = integer );
synonym sizeof_Signed_long_long Integer = (LLONG_BIT+7)/8 /*minimum 8*/;
value type Star_Signed_long_long
     inherits Star_type < Signed_long_long, sizeof_Signed_long_long>
     {  Star_Signed_long_long_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Signed_long_long = Star_void_to_Star_type,
        Star_Signed_long_long_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Signed_long_long = Integer_to_Star_type);
synonym sizeof_Star_Signed_long_long Integer = sizeof_Star_void;
value type Unsigned_long_long
     inherits Integeru <
        /*n*/ LLONG_BIT /*external, minimum 64 */,
        /*Integer1*/ Unsigned_char,
        /*Integer2*/ Unsigned_short,
        /*Integer3*/ Unsigned_int,
        /*Integer4*/ Unsigned_long >
        (to_Unsigned_long = integer );
synonym sizeof_Unsigned_long_long Integer = sizeof_Signed_long_long;
value type Star_Unsigned_long_long
     inherits Star_type < Unsigned_long_long, sizeof_Unsigned_long_long>
     {  Star_Unsigned_long_long_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Unsigned_long_long = Star_void_to_Star_type,
        Star_Unsigned_long_long_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Unsigned_long_long = Integer_to_Star_type);
synonym sizeof_Star_Unsigned_long_long Integer = sizeof_Star_void;
/*
```

C.1.6.9 C floating numbers types

C has three different floating types: `Float`, `Double` and `Long_double` with potentially different levels of precision and range of the values. By contrast, `<<`**`package`** `Predefined>>Real` in theory can represent values to any level of precision and an unbounded range. In practice the implementation of floating numbers is (usually) dependent on hardware support and therefore is implementation dependent. The 3 types (`Float`, `Double` and `Long_double`) are therefore considered formally equivalent in this Recommendation except the size of each as defined by synonyms below, so that `Double` and `Long_double` are each defined as a syntype of `Float`.

```
*/
syntype Double = Float {} /* alternative name for Float */
syntype Long_double = Float {} /* alternative name for Float */
synonym sizeof_Float Integer = external /* at least 4 */;
synonym sizeof_Double Integer = external /* at least 8 */;
synonym sizeof_Long_double Integer = external /* at least 16 */;
/*
```

A new value type is introduced for `Float` (rather than define a syntype of `Real`) to define arithmetic operators (such as `"+"` between each of the C integer types and `Float`. An operator such as `"+"` is defined between `Float` and each of the following: `Real`, `Integer`, `Signed_char`, `Unsigned_char`, `Signed_short`, `Unsigned_short`, `Signed_int`, `Unsigned_int`, `Signed_long`, `Unsigned_long`, `Signed_long_long` and `Unsigned_long_long`. The corresponding operator is also defined in each case where the sorts of the parameters are interchanged so the second parameter sort is `Float` and for the case where both parameter sorts are `Float`. The result sort of each arithmetic operator is `Real`. For each such operator name there are therefore 25 signatures.

Comparison operators ("==","!=","<",">","<=",">=") are defined between the pairs of sorts as above, but also for both Boolean and Integer results. For each such operator name there are therefore 50 signatures.

Assignment expression methods ("=","*=","/=","+=","-=") are defined where the right-hand side is a Float, Integer Real or C integer type and the result is the Real value of the updated Float variable.

The operators between C integer types and Float are defined with the type for the integer rather than below, to shorten the description of Float.

Similar to integers, it is assumed that the result of all operators (except comparisons or "&" for pointers) involving floating numbers is a Real. Also, there are no literals for Float: the constructor is the operator

```
to_Float ( Real) -> Float;
```

To shorten the description of Float below, for each operation only the <operation definition> is given from which the <operation signature> is constructed (see the *Model* part of clause 12.1.7).

```
value type Float {
     literals unordered; /* "<", ">","<=",">=", first, last, pred, succ,
        and num are not implicitly defined
     operators and methods signature lists are constructed from the definitions
        of the operations below */
     operator to_Real (f Float) -> Real external;
     operator to_Float(r Real) ->Float external; /* constructor */
     /* the casting operators are not formally defined.
        It is assumed that:
           to_Real (to_Float(r)) has the value r, and
           to_Float (to_Real(f)) has the value f. */
     operator Integer_to_Float(i Integer) ->Float {return to_Float(float(i))}
     operator "&" ( f Float )-> Void_star external; /* 'address of' operator.*/
     operator "+"(f Float)-> Real{return  to_Real(f)} /*unary plus */
     operator "-"(f Float)-> Real{return -to_Real(f)} /*unary minus*/
     /* addition */
     operator "+"(f1 Float, f2 Float)-> Real{return to_Real(f1) + to_Real(f2)}
     operator "+"(f Float, r Real)-> Real{return to_Real(f) + r}
     operator "+"(r Real, f Float)-> Real{return to_Real(f) + r}
     operator "+"(f Float, i Integer)-> Real{return to_Real(f) + float(i)}
     operator "+"(i Integer, f Float)-> Real{return to_Real(f) + float(i)}
     /* subtraction */
     operator "-"(f1 Float, f2 Float)-> Real{return to_Real(f1) - to_Real(f2)}
     operator "-"(f Float, r Real)-> Real{return to_Real(f) - r}
     operator "-"(r Real, f Float)-> Real{return r - to_Real(f) }
     operator "-"(f Float, i Integer)-> Real{return to_Real(f) - float(i)}
     operator "-"(i Integer, f Float)-> Real{return float(i) - to_Real(f)}
     /* multiplication */
     operator "*"(f1 Float, f2 Float)-> Real{return to_Real(f1) * to_Real(f2)}
     operator "*"(f Float, r Real)-> Real{return to_Real(f) * r}
     operator "*"(r Real, f Float)-> Real{return to_Real(f) * r}
     operator "*"(f Float, i Integer)-> Real{return to_Real(f) * float(i)}
     operator "*"(i Integer, f Float)-> Real{return to_Real(f) * float(i)}
     /* division */
     operator "/"(f1 Float, f2 Float)-> Real{return to_Real(f1) / to_Real(f2)}
     operator "/"(f Float, r Real)-> Real{return to_Real(f) / r}
     operator "/"(r Real, f Float)-> Real{return r / to_Real(f)}
     operator "/"(f Float, i Integer)-> Real{return to_Real(f) / float(i)}
     operator "/"(i Integer, f Float)-> Real{return to_Real(i) / float(f)}
     /* equal to - Boolean result */
     operator "=="(f1 Float,f2 Float)-> Boolean{return to_Real(f1)=to_Real(f2)}
     operator "=="(f Float, r Real)-> Boolean{return to_Real(f) = r}
     operator "=="(r Real, f Float)-> Boolean{return r = to_Real(f)}
     operator "=="(f Float, i Integer)-> Boolean{return to_Real(f) = float(i)}
     operator "=="(i Integer, f Float)-> Boolean{return float(i) = to_Real(f)}
     /* not equal to - Boolean result */
```

```
      operator "!="(f1 Float,f2 Float)->Boolean{return to_Real(f1)/=to_Real(f2)}
      operator "!="(f Float, r Real)-> Boolean{return to_Real(f) /= r}
      operator "!="(r Real, f Float)-> Boolean{return r /= to_Real(f)}
      operator "!="(f Float, i Integer)-> Boolean{return to_Real(f) /= float(i)}
      operator "!="(i Integer, f Float)-> Boolean{return float(i) /= to_Real(f)}
      /* less than - Boolean result */
      operator "<"(f1 Float,f2 Float)-> Boolean{return to_Real(f1)<to_Real(f2)}
      operator "<"(f Float, r Real)-> Boolean{return to_Real(f) < r}
      operator "<"(r Real, f Float)-> Boolean{return r < to_Real(f)}
      operator "<"(f Float, i Integer)-> Boolean{return to_Real(f) < float(i)}
      operator "<"(i Integer, f Float)-> Boolean{return float(i) < to_Real(f)}
      /* greater than - Boolean result */
      operator ">"(f1 Float,f2 Float)-> Boolean{return to_Real(f1)>to_Real(f2)}
      operator ">"(f Float, r Real)-> Boolean{return to_Real(f) > r}
      operator ">"(r Real, f Float)-> Boolean{return r > to_Real(f)}
      operator ">"(f Float, i Integer)-> Boolean{return to_Real(f) > float(i)}
      operator ">"(i Integer, f Float)-> Boolean{return float(i) > to_Real(f)}
      /* less than or equal - Boolean result */
      operator"<="(f1 Float,f2 Float)-> Boolean{return to_Real(f1)<=to_Real(f2)}
      operator"<="(f Float, r Real)-> Boolean{return to_Real(f) <= r}
      operator"<="(r Real, f Float)-> Boolean{return r <= to_Real(f)}
      operator"<="(f Float, i Integer)-> Boolean{return to_Real(f) <= float(i)}
      operator"<="(i Integer, f Float)-> Boolean{return float(i) <= to_Real(f)}
      /* greater than or equal - Boolean result */
      operator">="(f1 Float,f2 Float)-> Boolean{return to_Real(f1)>=to_Real(f2)}
      operator">="(f Float, r Real)-> Boolean{return to_Real(f) >= r}
      operator">="(r Real, f Float)-> Boolean{return r >= to_Real(f)}
      operator">="(f Float, i Integer)-> Boolean{return to_Real(f) >= float(i)}
      operator">="(i Integer, f Float)-> Boolean{return float(i) >= to_Real(f)}
      /* equal to - Integer result */
      operator "=="(f1 Float,f2 Float)-> Integer
         {return if to_Real(f1)=to_Real(f2)then 1 else 0 fi}
      operator "=="(f Float, r Real)-> Integer{return if to_Real(f) = rthen 1 else 0 fi}
      operator "=="(r Real, f Float)-> Integer{return if r = to_Real(f)then 1 else 0 fi}
      operator "=="(f Float, i Integer)-> Integer
         {return if to_Real(f) = float(i)then 1 else 0 fi}
      operator "=="(i Integer, f Float)-> Integer
         {return if float(i) = to_Real(f)then 1 else 0 fi}
      /* not equal to - Integer result */
      operator "!="(f1 Float,f2 Float)->Integer
         {return if to_Real(f1)/=to_Real(f2)then 1 else 0 fi}
      operator "!="(f Float, r Real)-> Integer{return if to_Real(f) /= rthen 1 else 0
fi}
      operator "!="(r Real, f Float)-> Integer{return if r /= to_Real(f)then 1 else 0
fi}
      operator "!="(f Float, i Integer)-> Integer
         {return if to_Real(f) /= float(i)then 1 else 0 fi}
      operator "!="(i Integer, f Float)-> Integer
         {return if float(i) /= to_Real(f)then 1 else 0 fi}
      /* less than - Integer result */
      operator "<"(f1 Float,f2 Float)-> Integer
         {return if to_Real(f1)<to_Real(f2) then 1 else 0 fi}
      operator "<"(f Float, r Real)-> Integer{return if to_Real(f) < r then 1 else 0 fi}
      operator "<"(r Real, f Float)-> Integer{return if r < to_Real(f) then 1 else 0 fi}
      operator "<"(f Float, i Integer)-> Integer
         {return if to_Real(f) < float(i) then 1 else 0 fi}
      operator "<"(i Integer, f Float)-> Integer
         {return if float(i) < to_Real(f) then 1 else 0 fi}
      /* greater than - Integer result */
      operator ">"(f1 Float,f2 Float)-> Integer
         {return if to_Real(f1)>to_Real(f2) then 1 else 0 fi}
      operator ">"(f Float, r Real)-> Integer{return if to_Real(f) > r then 1 else 0 fi}
      operator ">"(r Real, f Float)-> Integer{return if r > to_Real(f) then 1 else 0 fi}
      operator ">"(f Float, i Integer)-> Integer
         {return if to_Real(f) > float(i) then 1 else 0 fi}
```

```
      operator ">"(i Integer, f Float)-> Integer
         {return if float(i) > to_Real(f) then 1 else 0 fi}
      /* less than or equal - Integer result */
      operator"<="(f1 Float,f2 Float)-> Integer
         {return if to_Real(f1)<=to_Real(f2) then 1 else 0 fi}
      operator"<="(f Float, r Real)-> Integer{return if to_Real(f) <= r then 1 else 0
fi}
      operator"<="(r Real, f Float)-> Integer{return if r <= to_Real(f) then 1 else 0
fi}
      operator"<="(f Float, i Integer)-> Integer
         {return if to_Real(f) <= float(i) then 1 else 0 fi}
      operator"<="(i Integer, f Float)-> Integer
         {return if float(i) <= to_Real(f) then 1 else 0 fi}
      /* greater than or equal - Integer result */
      operator">="(f1 Float,f2 Float)-> Integer
         {return if to_Real(f1)>=to_Real(f2) then 1 else 0 fi}
      operator">="(f Float, r Real)-> Integer{return if to_Real(f) >= r then 1 else 0
fi}
      operator">="(r Real, f Float)-> Integer{return if r >= to_Real(f) then 1 else 0
fi}
      operator">="(f Float, i Integer)-> Integer
         {return if to_Real(f) >= float(i) then 1 else 0 fi}
      operator">="(i Integer, f Float)-> Integer
         {return if float(i) >= to_Real(f) then 1 else 0 fi}
      /* Simple assignment */
      method " ="   (f Float  ) -> Real {this:=f;                  return to_Real(this)}
      method " ="   (r Real   ) -> Real {this:=to_Float(r);        return to_Real(this)}
      method " ="   (i Integer) -> Real {this:=to_Float(float(i)); return to_Real(this)}
      method " ="   (i Signed_char) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method " ="   (i Signed_short) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method " ="   (i Signed_int) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method " ="   (i Signed_long) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method " ="   (i Signed_long_long) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method " ="   (i Unsigned_char) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method " ="   (i Unsigned_short) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method " ="   (i Unsigned_int) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method " ="   (i Unsigned_long) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method " ="   (i Unsigned_long_long) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      /* multiplication assignment */
      method " *="  (f Float)              -> Real {this:=this * f; return to_Real(this)}
      method " *="  (r Real)               -> Real {this:=this * r; return to_Real(this)}
      method " *="  (i Integer)            -> Real {this:=this * i; return to_Real(this)}
      method " *="  (i Signed_char)        -> Real {this:=this * i; return to_Real(this)}
      method " *="  (i Signed_short)       -> Real {this:=this * i; return to_Real(this)}
      method " *="  (i Signed_int)         -> Real {this:=this * i; return to_Real(this)}
      method " *="  (i Signed_long)        -> Real {this:=this * i; return to_Real(this)}
      method " *="  (i Signed_long_long)   -> Real {this:=this * i; return to_Real(this)}
      method " *="  (i Unsigned_char)      -> Real {this:=this * i; return to_Real(this)}
      method " *="  (i Unsigned_short)     -> Real {this:=this * i; return to_Real(this)}
      method " *="  (i Unsigned_int)       -> Real {this:=this * i; return to_Real(this)}
      method " *="  (i Unsigned_long)      -> Real {this:=this * i; return to_Real(this)}
      method " *="  (i Unsigned_long_long) -> Real {this:=this * i; return to_Real(this)}
      /* division assignment */
      method " /="  (f Float)              -> Real {this:=this / f; return to_Real(this)}
      method " /="  (r Real)               -> Real {this:=this / r; return to_Real(this)}
      method " /="  (i Integer)            -> Real {this:=this / i; return to_Real(this)}
      method " /="  (i Signed_char)        -> Real {this:=this / i; return to_Real(this)}
      method " /="  (i Signed_short)       -> Real {this:=this / i; return to_Real(this)}
      method " /="  (i Signed_int)         -> Real {this:=this / i; return to_Real(this)}
```

```
     method " /=" (i Signed_long)        -> Real {this:=this / i; return to_Real(this)}
     method " /=" (i Signed_long_long)   -> Real {this:=this / i; return to_Real(this)}
     method " /=" (i Unsigned_char)      -> Real {this:=this / i; return to_Real(this)}
     method " /=" (i Unsigned_short)     -> Real {this:=this / i; return to_Real(this)}
     method " /=" (i Unsigned_int)       -> Real {this:=this / i; return to_Real(this)}
     method " /=" (i Unsigned_long)      -> Real {this:=this / i; return to_Real(this)}
     method " /=" (i Unsigned_long_long) -> Real {this:=this / i; return to_Real(this)}
     /* addition assignment */
     method " +=" (f Float)              -> Real {this:=this + f; return to_Real(this)}
     method " +=" (r Real)               -> Real {this:=this + r; return to_Real(this)}
     method " +=" (i Integer)            -> Real {this:=this + i; return to_Real(this)}
     method " +=" (i Signed_char)        -> Real {this:=this + i; return to_Real(this)}
     method " +=" (i Signed_short)       -> Real {this:=this + i; return to_Real(this)}
     method " +=" (i Signed_int)         -> Real {this:=this + i; return to_Real(this)}
     method " +=" (i Signed_long)        -> Real {this:=this + i; return to_Real(this)}
     method " +=" (i Signed_long_long)   -> Real {this:=this + i; return to_Real(this)}
     method " +=" (i Unsigned_char)      -> Real {this:=this + i; return to_Real(this)}
     method " +=" (i Unsigned_short)     -> Real {this:=this + i; return to_Real(this)}
     method " +=" (i Unsigned_int)       -> Real {this:=this + i; return to_Real(this)}
     method " +=" (i Unsigned_long)      -> Real {this:=this + i; return to_Real(this)}
     method " +=" (i Unsigned_long_long) -> Real {this:=this + i; return to_Real(this)}
     /* subtraction assignment */
     method " -=" (f Float)              -> Real {this:=this - f; return to_Real(this)}
     method " -=" (r Real)               -> Real {this:=this - r; return to_Real(this)}
     method " -=" (i Integer)            -> Real {this:=this - i; return to_Real(this)}
     method " -=" (i Signed_char)        -> Real {this:=this - i; return to_Real(this)}
     method " -=" (i Signed_short)       -> Real {this:=this - i; return to_Real(this)}
     method " -=" (i Signed_int)         -> Real {this:=this - i; return to_Real(this)}
     method " -=" (i Signed_long)        -> Real {this:=this - i; return to_Real(this)}
     method " -=" (i Signed_long_long)   -> Real {this:=this - i; return to_Real(this)}
     method " -=" (i Unsigned_char)      -> Real {this:=this - i; return to_Real(this)}
     method " -=" (i Unsigned_short)     -> Real {this:=this - i; return to_Real(this)}
     method " -=" (i Unsigned_int)       -> Real {this:=this - i; return to_Real(this)}
     method " -=" (i Unsigned_long)      -> Real {this:=this - i; return to_Real(this)}
     method " -=" (i Unsigned_long_long) -> Real {this:=this - i; return to_Real(this)}
     /* Assignment methods with names " %=" (remainder), " <<=" (shift left),
        " >>=" (shift right), " &=" (logical and>, " ^=" (logical xor),
        and " |=" (logical or) are not defined for Float */
     default to_Float(0.0);
} /* end value type Float*/
value type Star_Float
     inherits Star_type < Float, sizeof_Float >
     {  Star_Float_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Float = Star_void_to_Star_type);
synonym sizeof_Star_Float Integer = sizeof_Star_void;
/*
```

C.1.6.10 Parameterized type for arrays Cvector

Cvector is used to define the types of arrays.

```
*/
value type Cvector < type Itemsort; synonym MaxIndex Integer >
     inherits Array< Indexsort, Itemsort >
     {syntype Indexsort = Integer constants 0:MaxIndex-1;}
/* end value type Cvector
```

C.1.6.11 Package end

```
*/
endpackage C_Predefined;
/* */
```

Bibliography

[b-ISO/IEC 9899] ISO/IEC 9899:2011, *Information technology – Programming languages – C.*

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Terminals and subjective and objective assessment methods
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects and next-generation networks
Series Z	**Languages and general software aspects for telecommunication systems**

Printed in Switzerland
Geneva, 2013