SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

Programming languages – CHILL: The ITU-T programming language

CHILL – The ITU-T Programming Language

ITU-T Recommendation Z.200

(Previously CCITT Recommendation)
**ITU-T Z-SERIES RECOMMENDATIONS**

**LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS**

<table>
<thead>
<tr>
<th>FORMAL DESCRIPTION TECHNIQUES (FDT)</th>
<th>Z.100–Z.109</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification and Description Language (SDL)</td>
<td>Z.100–Z.109</td>
</tr>
<tr>
<td>Application of Formal Description Techniques</td>
<td>Z.110–Z.119</td>
</tr>
<tr>
<td>Message Sequence Chart</td>
<td>Z.120–Z.129</td>
</tr>
</tbody>
</table>

**PROGRAMMING LANGUAGES**

<table>
<thead>
<tr>
<th>CHILL: The ITU-T programming language</th>
<th>Z.200–Z.209</th>
</tr>
</thead>
</table>

**MAN-MACHINE LANGUAGE**

<table>
<thead>
<tr>
<th>General principles</th>
<th>Z.300–Z.309</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic syntax and dialogue procedures</td>
<td>Z.310–Z.319</td>
</tr>
<tr>
<td>Extended MML for visual display terminals</td>
<td>Z.320–Z.329</td>
</tr>
<tr>
<td>Specification of the man-machine interface</td>
<td>Z.330–Z.399</td>
</tr>
</tbody>
</table>

**QUALITY OF TELECOMMUNICATION SOFTWARE**

| Z.400–Z.499 |

**METHODS FOR VALIDATION AND TESTING**

| Z.500–Z.599 |

For further details, please refer to the list of ITU-T Recommendations.
Summary
This Recommendation | International Standard defines the ITU-T programming language CHILL. CHILL is a strongly typed, block structured and object-oriented language designed primarily for the implementation of large and complex embedded systems.

CHILL was designed to provide reliability and run time efficiency, at the same time sufficient flexibility and powerfulness to encompass the required range of applications. CHILL also provides facilities that encourage piecewise and modular development of large systems.

Source
ITU-T Recommendation Z.200 was prepared by ITU-T Study Group 10 (1997-2000) and approved on 19 November 1999. An identical text is also published as ISO | IEC 9496.
FOREWORD

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunication. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the ITU. The 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 Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 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.

INTELLECTUAL PROPERTY RIGHTS

The ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. The 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, the ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

© ITU 2001

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the ITU.
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>General</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Language survey</td>
<td>1</td>
</tr>
<tr>
<td>1.3</td>
<td>Modes and classes</td>
<td>2</td>
</tr>
<tr>
<td>1.4</td>
<td>Locations and their accesses</td>
<td>3</td>
</tr>
<tr>
<td>1.5</td>
<td>Values and their operations</td>
<td>3</td>
</tr>
<tr>
<td>1.6</td>
<td>Actions</td>
<td>4</td>
</tr>
<tr>
<td>1.7</td>
<td>Input and output</td>
<td>4</td>
</tr>
<tr>
<td>1.8</td>
<td>Exception handling</td>
<td>4</td>
</tr>
<tr>
<td>1.9</td>
<td>Time supervision</td>
<td>5</td>
</tr>
<tr>
<td>1.10</td>
<td>Program structure</td>
<td>5</td>
</tr>
<tr>
<td>1.11</td>
<td>Concurrent execution</td>
<td>5</td>
</tr>
<tr>
<td>1.12</td>
<td>General semantic properties</td>
<td>6</td>
</tr>
<tr>
<td>1.13</td>
<td>Implementation options</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Preliminaries</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>The metalanguage</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Vocabulary</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>The use of spaces</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Comments</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Format effectors</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>Compiler directives</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Names and their defining occurrences</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Modes and classes</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>General</td>
<td>12</td>
</tr>
<tr>
<td>3.2</td>
<td>Mode definitions</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>Mode classification</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>Discrete modes</td>
<td>17</td>
</tr>
<tr>
<td>3.5</td>
<td>Real modes</td>
<td>20</td>
</tr>
<tr>
<td>3.6</td>
<td>Powerset modes</td>
<td>22</td>
</tr>
<tr>
<td>3.7</td>
<td>Reference modes</td>
<td>22</td>
</tr>
<tr>
<td>3.8</td>
<td>Procedure modes</td>
<td>23</td>
</tr>
<tr>
<td>3.9</td>
<td>Instance modes</td>
<td>24</td>
</tr>
<tr>
<td>3.10</td>
<td>Synchronization modes</td>
<td>25</td>
</tr>
<tr>
<td>3.11</td>
<td>Input-Output Modes</td>
<td>26</td>
</tr>
<tr>
<td>3.12</td>
<td>Timing modes</td>
<td>28</td>
</tr>
<tr>
<td>3.13</td>
<td>Composite modes</td>
<td>29</td>
</tr>
<tr>
<td>3.14</td>
<td>Dynamic modes</td>
<td>37</td>
</tr>
<tr>
<td>3.15</td>
<td>Moreta Modes</td>
<td>38</td>
</tr>
<tr>
<td>4</td>
<td>Locations and their accesses</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Declarations</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Locations</td>
<td>47</td>
</tr>
<tr>
<td>5</td>
<td>Values and their operations</td>
<td>54</td>
</tr>
<tr>
<td>5.1</td>
<td>Synonym definitions</td>
<td>54</td>
</tr>
<tr>
<td>5.2</td>
<td>Primitive value</td>
<td>55</td>
</tr>
<tr>
<td>5.3</td>
<td>Values and expressions</td>
<td>70</td>
</tr>
</tbody>
</table>
## Appendix IV – Program examples

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV.1 Operations on integers</td>
<td>179</td>
</tr>
<tr>
<td>IV.2 Same operations on fractions</td>
<td>179</td>
</tr>
<tr>
<td>IV.3 Same operations on complex numbers</td>
<td>180</td>
</tr>
<tr>
<td>IV.4 General order arithmetic</td>
<td>180</td>
</tr>
<tr>
<td>IV.5 Adding bit by bit and checking the result</td>
<td>180</td>
</tr>
<tr>
<td>IV.6 Playing with dates</td>
<td>181</td>
</tr>
<tr>
<td>IV.7 Roman numerals</td>
<td>182</td>
</tr>
<tr>
<td>IV.8 Counting letters in a character string of arbitrary length</td>
<td>183</td>
</tr>
<tr>
<td>IV.9 Prime numbers</td>
<td>184</td>
</tr>
<tr>
<td>IV.10 Implementing stacks in two different ways, transparent to the user</td>
<td>184</td>
</tr>
<tr>
<td>IV.11 Fragment for playing chess</td>
<td>185</td>
</tr>
<tr>
<td>IV.12 Building and manipulating a circularly linked list</td>
<td>188</td>
</tr>
<tr>
<td>IV.13 A region for managing competing accesses to a resource</td>
<td>189</td>
</tr>
<tr>
<td>IV.14 Queuing calls to a switchboard</td>
<td>190</td>
</tr>
<tr>
<td>IV.15 Allocating and deallocating a set of resources</td>
<td>190</td>
</tr>
<tr>
<td>IV.16 Allocating and deallocating a set of resources using buffers</td>
<td>192</td>
</tr>
<tr>
<td>IV.17 String scanner1</td>
<td>194</td>
</tr>
<tr>
<td>IV.18 String scanner2</td>
<td>195</td>
</tr>
<tr>
<td>IV.19 Removing an item from a double linked list</td>
<td>196</td>
</tr>
<tr>
<td>IV.20 Update a record of a file</td>
<td>196</td>
</tr>
<tr>
<td>IV.21 Merge two sorted files</td>
<td>197</td>
</tr>
<tr>
<td>IV.22 Read a file with variable length records</td>
<td>198</td>
</tr>
<tr>
<td>IV.23 The use of spec modules</td>
<td>199</td>
</tr>
<tr>
<td>IV.24 Example of a context</td>
<td>199</td>
</tr>
<tr>
<td>IV.25 The use of prefixing and remote modules</td>
<td>199</td>
</tr>
<tr>
<td>Appendix V – Decommitted features</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>V.1 Free directive</td>
<td>206</td>
</tr>
<tr>
<td>V.2 Integer modes syntax</td>
<td>206</td>
</tr>
<tr>
<td>V.3 Set modes with holes</td>
<td>206</td>
</tr>
<tr>
<td>V.4 Procedure modes syntax</td>
<td>206</td>
</tr>
<tr>
<td>V.5 String modes syntax</td>
<td>207</td>
</tr>
<tr>
<td>V.6 Array modes syntax</td>
<td>207</td>
</tr>
<tr>
<td>V.7 Level structure notation</td>
<td>207</td>
</tr>
<tr>
<td>V.8 Map reference names</td>
<td>207</td>
</tr>
<tr>
<td>V.9 Based declarations</td>
<td>207</td>
</tr>
<tr>
<td>V.10 Character string literals</td>
<td>207</td>
</tr>
<tr>
<td>V.11 Receive expressions</td>
<td>207</td>
</tr>
<tr>
<td>V.12 Addr notation</td>
<td>207</td>
</tr>
<tr>
<td>V.13 Assignment syntax</td>
<td>207</td>
</tr>
<tr>
<td>V.14 Case action syntax</td>
<td>207</td>
</tr>
<tr>
<td>V.15 Do for action syntax</td>
<td>207</td>
</tr>
<tr>
<td>V.16 Explicit loop counters</td>
<td>208</td>
</tr>
<tr>
<td>V.17 Call action syntax</td>
<td>208</td>
</tr>
<tr>
<td>V.18 RECURSEFAIL exception</td>
<td>208</td>
</tr>
<tr>
<td>V.19 Start action syntax</td>
<td>208</td>
</tr>
<tr>
<td>V.20 Explicit value receive names</td>
<td>208</td>
</tr>
<tr>
<td>V.21 Blocks</td>
<td>208</td>
</tr>
<tr>
<td>V.22 Entry statement</td>
<td>208</td>
</tr>
<tr>
<td>V.23 Register names</td>
<td>208</td>
</tr>
<tr>
<td>V.24 Recursive attribute</td>
<td>208</td>
</tr>
<tr>
<td>V.25 Quasi cause statements and quasi handlers</td>
<td>209</td>
</tr>
<tr>
<td>V.26 Syntax of quasi statements</td>
<td>209</td>
</tr>
<tr>
<td>V.27 Weakly visible names and visibility statements</td>
<td>209</td>
</tr>
<tr>
<td>V.28 Weakly visible names and visibility statements</td>
<td>209</td>
</tr>
<tr>
<td>V.29 Pervasiveness</td>
<td>209</td>
</tr>
<tr>
<td>V.30 Seizing by modulation name</td>
<td>209</td>
</tr>
<tr>
<td>V.31 Predefined simple name strings</td>
<td>209</td>
</tr>
</tbody>
</table>

Appendix VI – Index of production rules | 210
This Recommendation | International Standard defines the ITU-T programming language CHILL. When CHILL was first defined in 1980 "CHILL" stood for CCITT High Level Language.

The following subclauses of this clause introduce some of the motivations behind the language design and provide an overview of the language features.

For information concerning the variety of introductory and training material on this subject, the reader is referred to the Manuals, "Introduction to CHILL" and "CHILL user's manual".

An alternative definition of CHILL, in a strict mathematical form (based on the VDM notation), is available in the Manual entitled "Formal definition of CHILL".

### 1.1 General

CHILL is a strongly typed, block structured language designed primarily for the implementation of large and complex embedded systems.

CHILL was designed to:

- enhance reliability and run time efficiency by means of extensive compile-time checking;
- be sufficiently flexible and powerful to encompass the required range of applications and to exploit a variety of hardware;
- provide facilities that encourage piecewise and modular development of large systems;
- cater for real-time applications by providing built-in concurrency and time supervision primitives;
- permit the generation of highly efficient object code;
- be easy to learn and use.

The expressive power inherent in the language design allows engineers to select the appropriate constructs from a rich set of facilities such that the resulting implementation can match the original specification more precisely.

Because CHILL is careful to distinguish between static and dynamic objects, nearly all the semantic checking can be achieved at compile time. This has obvious run time benefits. Violation of CHILL dynamic rules results in run-time exceptions which can be intercepted by an appropriate exception handler (however, generation of such implicit checks is optional, unless a user defined handler is explicitly specified).

CHILL permits programs to be written in a machine independent manner. The language itself is machine independent; however, particular compilation systems may require the provision of specific implementation defined objects. It should be noted that programs containing such objects will not, in general, be portable.

### 1.2 Language survey

A CHILL program consists essentially of three parts:

- a description of objects;
- a description of actions which are to be performed upon the objects;
- a description of the program structure.
Objects are described by data statements (declaration and definition statements), actions are described by action statements and the program structure is described by program structuring statements.

The manipulatable objects of CHILL are values and locations where values can be stored. The actions define the operations to be performed upon the objects and the order in which values are stored into and retrieved from locations. The program structure determines the lifetime and visibility of objects.

CHILL provides for extensive static checking of the use of objects in a given context.

In the following subclauses, a summary of the various CHILL concepts is given. Each subclause is an introduction to a clause with the same title, describing the concept in detail.

1.3 Modes and classes

A location has a mode attached to it. The mode of a location defines the set of values which may reside in that location and other properties associated with it (note that not all properties of a location are determinable by its mode alone). Properties of locations are: size, internal structure, read-onliness, referability, etc. Properties of values are: internal representation, ordering, applicable operations, etc.

A value has a class attached to it. The class of a value determines the modes of the locations that may contain the value.

CHILL provides the following categories of modes:

- discrete modes: integer, character, boolean, set (enumerations) modes and ranges thereof;
- real modes: floating point modes and ranges thereof;
- powerset modes: sets of elements of some discrete mode;
- reference modes: bound references, free references and rows used as references to locations;
- composite modes: string, array and structure modes;
- procedure modes: procedures considered as manipulatable data objects;
- instance modes: identifications for processes;
- synchronization modes: event and buffer modes for process synchronization and communication;
- input-output modes: association, access and text modes for input-output operations;
- timing modes: duration and absolute time modes for time supervision;
- moreta modes: module, region and task modes for object orientation with single inheritance.

CHILL provides denotations for a set of standard modes. Program defined modes can be introduced by means of mode definitions. Some language constructs have a so-called dynamic mode attached. A dynamic mode is a mode of which some properties can be determined only dynamically. Dynamic modes are always parameterized modes with run-time parameters. A mode that is not dynamic is called a static mode.

With moreta modes CHILL supports object oriented programming in a very versatile manner. There are three kinds of modes for objects:

- module modes: the values of these modes behave very much like modules and resemble therefore mostly the objects in classical object oriented programming (e.g. Smalltalk, C++, Eiffel, Java);
- region modes: the values of these modes behave very much like regions. Such objects are usually not found in classical object oriented programming;
- task modes: the values of these modes have essentially the same structure as regions but have their own thread of control, and communication between them and other objects is done asynchronously.

Classes have no denotation in CHILL. They are introduced in the metalanguage only to describe static and dynamic context conditions.
1.4 Locations and their accesses

Locations are places where values can be stored or from which values can be obtained. In order to store or obtain a value, a location has to be accessed.

Declaration statements define names to be used for accessing a location. There are:

1) location declarations;
2) loc-identity declarations.

The first one creates locations and establishes access names to the newly created locations. The latter one establishes new access names for locations created elsewhere.

Apart from location declarations, new locations can be created by means of a GETSTACK or ALLOCATE built-in routine call yielding reference values (see below) to the newly created location.

A location may be referable. This means that a corresponding reference value exists for the location. This reference value is obtained as the result of the referencing operation, applied to the referable location. By dereferencing a reference value, the referred location is obtained. CHILL requires certain locations to be referable and others to be not referable, but for other locations it is left to the implementation to decide whether or not they are referable. Referability must be a statically determinable property of locations.

A location may have a read-only mode, which means that it can only be accessed to obtain a value and not to store a new value into it (except when initializing).

A location may be composite, which means that it has sub-locations which can be accessed separately. A sub-location is not necessarily referable. A location containing at least one read-only sub-location is said to have the read-only property. The accessing methods delivering sub-locations (or sub-values) are indexing and slicing for strings and for arrays, and selection for structures.

A location has a mode attached. If this mode is dynamic, the location is called a dynamic mode location.

The following properties of a location, although statically determinable, are not part of the mode:

referability: whether or not a reference value exists for the location;
storage class: whether or not it is statically allocated;
regionality: whether or not the location is declared within a region.

1.5 Values and their operations

Values are basic objects on which specific operations are defined. A value is either a (CHILL) defined value or an undefined value (in the CHILL sense). The usage of an undefined value in specified contexts results in an undefined situation (in the CHILL sense) and the program is considered to be incorrect.

CHILL allows locations to be used in contexts where values are required. In this case, the location is accessed to obtain the value contained in it.

A value has a class attached. Strong values are values that besides their class also have a mode attached. In that case the value is always one of the values defined by the mode. The class is used for compatibility checking and the mode for describing properties of the value. Some contexts require those properties to be known and a strong value will then be required.

A value may be literal, in which case it denotes an implementation independent discrete value, known at compile time. A value may be constant, in which case it always delivers the same value, i.e. it need only be evaluated once. When the context requires a literal or constant value, the value is assumed to be evaluated before run-time and therefore cannot generate a run-time exception. A value may be intra-regional, in which case it can refer somehow to locations declared within a region. A value may be composite, i.e. contain sub-values.

Synonym definition statements establish new names to denote constant values.
1.6 Actions

Actions constitute the algorithmic part of a CHILL program.

The assignment action stores a (computed) value into one or more locations. The procedure call invokes a procedure, a built-in routine call invokes a built-in routine (a built-in routine is a procedure whose definition need not be written in CHILL and whose parameter and result mechanism may be more general). To return from and/or establish the result of a procedure call, the return and result actions are used.

To control the sequential action flow, CHILL provides the following flow of control actions:

- **if action**: for a two-way branch;
- **case action**: for a multiple branch. The selection of the branch may be based upon several values, similarly to a decision table;
- **do action**: for iteration or bracketing;
- **exit action**: for leaving a bracketed action or a module in a structured manner;
- **cause action**: to cause a specific exception;
- **goto action**: for unconditional transfer to a labelled program point.

Action and data statements can be grouped together to form a module or begin-end block, which form a (compound) action.

To control the concurrent action flow, CHILL provides the start, stop, delay, continue, send, delay case, and receive case actions, and receive and start expressions.

1.7 Input and output

The input and output facilities of CHILL provide the means to communicate with a variety of devices in the outside world.

The input-output reference model knows three states. In the free state there is no interaction with the outside world. Through an **ASSOCIATE** operation, the file handling state is entered. In the file handling state there are locations of association mode, which denote outside world objects. It is possible via built-in routines to read and modify the language defined attributes of associations, i.e. **existing**, **readable**, **writeable**, **indexable**, **sequencible** and **variable**. File creation and deletion are also done in the file handling state.

Through the **CONNECT** operation, a location of access mode is connected to a location of an association mode, and the data transfer state is entered. The **CONNECT** operation allows positioning of a **base** index in a file. In the data transfer state various attributes of locations of access mode can be inspected and the data transfer operations **READRECORD** and **WRITERECORD** can be applied.

Through the text transfer operations, CHILL values can be represented in a human-readable form which can be transferred to or from a file or a CHILL location.

1.8 Exception handling

The dynamic semantic conditions of CHILL are those (non context-free) conditions that, in general, cannot be statically determined. (It is left to the implementation to decide whether or not to generate code to test the dynamic conditions at run time, unless an appropriate handler is explicitly specified.) The violation of a dynamic semantic rule causes a runtime exception; however, if an implementation can determine statically that a dynamic condition will be violated, it may reject the program.

Exceptions can also be caused by the execution of a cause action or, conditionally, by the execution of an assert action. When, at a given program point, an exception occurs, control is transferred to the associated handler for that exception, if one is specified. Whether or not a handler is specified for an exception at a given point can be statically determined. If no explicit handler is specified, control may be transferred to an implementation defined exception handler.

Exceptions have a name, which is either a CHILL defined exception name, an implementation defined exception name, or a program defined exception name. Note that when a handler is specified for an exception name, the associated dynamic condition must be checked.
1.9 Time supervision

Time supervision facilities of CHILL provide the means to react to the elapsing of time in the external world. A process becomes timeoutable when it reaches a well-defined point in the execution of certain actions. At this point it may be interrupted. When this happens, control is transferred to an appropriate handler.

Programs may detect the elapsing of a period of time or may synchronize to an absolute point of time or at precise intervals without cumulated drifts. Built-in routines for time are provided to convert absolute time values and duration values into integer values, to suspend a process until a time supervision expires.

1.10 Program structure

The program structuring statements are the begin-end block, module, procedure, process, region and moreta mode. The program structuring statements provide the means of controlling the lifetime of locations and the visibility of names.

The lifetime of a location is the time during which a location exists within the program. Locations can be explicitly declared (in a location declaration) or generated (GETSTACK or ALLOCATE built-in routine call), or they can be implicitly declared or generated as the result of the use of language constructs.

A name is said to be visible at a certain point in the program if it may be used at that point. The scope of a name encompasses all the points where it is visible, i.e. where the denoted object is identified by that name.

Begin-end blocks determine both visibility of names and lifetime of locations.

Modules are provided to restrict the visibility of names to protect against unauthorized usage. By means of visibility statements, it is possible to exercise control over the visibility of names in various program parts.

A procedure is a (possibly parameterized) sub-program that may be invoked (called) at different places within a program. It may return a value (value procedure) or a location (location procedure), or deliver no result. In the latter case the procedure can only be called in a procedure call action.

Processes, task locations, regions and region locations provide the means by which a structure of concurrent executions can be achieved.

Generic templates provide the means by which generic modules, regions, procedures, processes and moreta modes can be constructed. These templates can be parameterized by SYN constants, modes and procedures. Generic instantiation statements are used to obtain (nongeneric) modules, regions, procedures, processes and moreta modes which are called generic instances. A generic instance is obtained from a generic template T by replacing in T the formal generic parameters with the corresponding actual generic parameters.

A complete CHILL program is a list of program units that is considered to be surrounded by an (imaginary) process definition. This outermost process is started by the system under whose control the program is executed. A program unit can be a module, a region, a moreta synmode definition statement, a moreta newmode definition statement or a generic template.

Constructs are provided to facilitate various ways of piecewise development of programs. A spec module and spec region are used to define the static properties of a program piece, a context is used to define the static properties of seized names. In addition it is possible to specify that the text of a program piece is to be found somewhere else through the remote facility.

1.11 Concurrent execution

CHILL allows for the concurrent execution of program units. A thread (process or task) is the unit of concurrent execution. The evaluation of a start expression causes the creation of a new process of the indicated process definition. The process is then considered to be executed concurrently with the starting thread. CHILL allows for one or more processes with the same or different definition to be active at one time. The stop action, executed by a process or a task, causes its termination.
A thread is always in one of two states; it can be active or delayed. The transition from active to delayed is called the delaying of the thread; the transition from delayed to active is called the re-activation of the thread. The execution of delaying actions on events, or receiving actions on buffers or signals, or sending actions on buffers, or call action to a component procedure of a region location, or call action to a component procedure of a task location in case there is not enough storage to perform can cause the executing thread to become delayed. The execution of a continue action on events, or sending actions on buffers or signals, or receiving actions on buffers, or release of a region location, or at the beginning of the execution of an externally called component procedure of a task location can cause a delayed thread to become active again.

Buffers and events are locations with restricted use. The operations send, receive and receive case are defined on buffers; the operations delay, delay case and continue are defined on events. Buffers are a means of synchronizing and transmitting information between processes. Events are used only for synchronization. Signals are defined in signal definition statements. They denote functions for composing and decomposing lists of values transmitted between processes. Send actions and receive case actions provide for communication of a list of values and for synchronization.

A region or region location is a special kind of module. Its use is to provide for mutually exclusive access to data structures that are shared by several threads.

1.12 General semantic properties

The semantic (non context-free) conditions of CHILL are the mode and class compatibility conditions (mode checking) and the visibility conditions (scope checking). The mode rules determine how names may be used; the scope rules determine where names may be used.

The mode rules are formulated in terms of compatibility requirements between modes, between classes and between modes and classes. The compatibility requirements between modes and classes and between classes themselves are defined in terms of equivalence relations between modes. If dynamic modes are involved, mode checking is partly dynamic.

The scope rules determine the visibility of names through the program structure and explicit visibility statements. The explicit visibility statements influence the scope of the mentioned names. Names introduced in a program have a place where they are defined or declared. This place is called the defining occurrence of the name. The places where the name is used are called applied occurrences of the name. The name binding rules associate a unique defining occurrence with each applied occurrence of the name.

1.13 Implementation options

CHILL allows for implementation defined integer modes, implementation defined built-in routines, implementation defined process names, implementation defined exception handlers and implementation defined exception names.

An implementation defined integer mode must be denoted by an implementation defined mode name. This name is considered to be defined in a newmode definition statement that is not specified in CHILL. Extending the existing CHILL-defined arithmetic operations to the implementation defined integer modes is allowed within the framework of the CHILL syntactic and semantic rules. Examples of implementation defined integer modes are long integers, and short integers.

A built-in routine is a procedure whose definition need not be written in CHILL and that may have a more general parameter passing and result transmission scheme than CHILL procedures.

A built-in process name is a process name whose definition need not be written in CHILL and that may have a more general parameter passing scheme than CHILL processes. A CHILL process may cooperate with built-in processes or start such processes.

An implementation defined exception handler is a handler appended to a process definition. If this handler receives control after the occurrence of an exception, the implementation decides which actions are to be taken. An implementation defined exception is caused if an implementation defined dynamic condition is violated.
2 Preliminaries

2.1 The metalanguage

The CHILL description consists of two parts:

- the description of the context-free syntax;
- the description of the semantic conditions.

2.1.1 The context-free syntax description

The context-free syntax is described using an extension of the Backus-Naur Form. Syntactic categories are indicated by one or more English words, written in slanted characters, enclosed between angular brackets (< and >). This indicator is called a non-terminal symbol. For each non-terminal symbol, a production rule is given in an appropriate syntax section. A production rule for a non-terminal symbol consists of the non-terminal symbol at the left-hand side of the symbol ::=, and one or more constructs, consisting of non-terminal and/or terminal symbols at the right-hand side. These constructs are separated by a vertical bar ( | ) to denote alternative productions for the non-terminal symbol.

Sometimes the non-terminal symbol includes an underlined part. This underlined part does not form part of the context-free description but defines a semantic category (see 2.1.2).

Syntactic elements may be grouped together by using curly brackets ({ and }). Repetition of curly bracketed groups is indicated by an asterisk (*) or plus (+). An asterisk indicates that the group is optional and can be further repeated any number of times; a plus indicates that the group must be present and can be further repeated any number of times. For example, \{A\}^* stands for any sequence of A’s, including zero, while \{A\}^+ stands for any sequence of at least one A. If syntactic elements are grouped using square brackets ([ and ]), then the group is optional. A curly or square bracketed group may contain one or more vertical bars, indicating alternative syntactic elements.

A distinction is made between strict syntax, for which the semantic conditions are given directly, and derived syntax. The derived syntax is considered to be an extension of the strict syntax and the semantics for the derived syntax is indirectly explained in terms of the associated strict syntax.

It is to be noted that the context-free syntax description is chosen to suit the semantic description in this Recommendation | International Standard and is not made to suit any particular parsing algorithm (e.g. there are some context-free ambiguities introduced in the interest of clarity). The ambiguities are resolved using the semantic category of the syntactic elements.

2.1.2 The semantic description

Each syntactic category (non-terminal symbol) is described in sub-sections semantics, static properties, dynamic properties, static conditions and dynamic conditions.

The section semantics describes the concepts denoted by the syntactic categories (i.e. their meaning and behaviour).

The section static properties defines statically determinable semantic properties of the syntactic category. These properties are used in the formulation of static and/or dynamic conditions in the sections where the syntactic category is used.

The section dynamic properties defines the properties of the syntactic category, which are known only dynamically.

The section static conditions describes the context-dependent, statically checkable conditions which must be fulfilled when the syntactic category is used. Some static conditions are expressed in the syntax by means of an underlined part in the non-terminal symbol (see 2.1.1). This use requires the non-terminal to be of a specific semantic category. For example, boolean expression is identical to <expression> in the context-free sense, but semantically it requires the expression to be of a boolean class.

The section dynamic conditions describes the context-dependent conditions that must be fulfilled during execution. In some cases, conditions are static if no dynamic modes are involved. In those cases, the condition is mentioned under static conditions and referred to under dynamic conditions. In other cases, dynamic conditions can be checked statically; an implementation may treat this as a violation of a static condition.
In the semantic description, different fonts are used in the following ways: slanted font (without < and >) is used to indicate syntactic objects; corresponding terms in roman font indicate corresponding semantic objects (e.g. a location denotes a location). Bolding is used to name semantic properties; sometimes a property can be expressed syntactically as well as semantically (e.g. the sentence "the expression is constant" means the same as "the expression is a constant expression").

Unless otherwise specified, the semantics, properties and conditions described in the sub-section of a syntactic category hold regardless of the context in which other sections that syntactic category may appear.

The properties of a syntactic category \( A \) that has a production rule of the form \( A ::= B \), where \( B \) is a syntactic category, are the same as \( B \) unless otherwise specified.

In this Recommendation | International Standard, virtual names are introduced to describe modes, locations and values which do not occur explicitly in the program text. In such cases the name is preceded by an ampersand (\&) symbol. These names are introduced for descriptive purposes only.

2.1.3 The examples

For most syntax sections, there is a section examples giving one or more examples of the defined syntactic categories. These examples are extracted from a set of program examples contained in Appendix IV. References indicate via which syntax rule each example is produced and from which example it is taken.

For example, 6.20 \((d+5)/5\) \((1.2)\) indicates an example of the terminal string \((d+5)/5\), produced via rule \((1.2)\) of the appropriate syntax section, taken from program example no. 6 line 20.

2.1.4 The binding rules in the metalanguage

Sometimes the semantic description mentions CHILL special simple name strings (see Appendix III). These special simple name strings are always used with their CHILL meaning and are therefore not influenced by the binding rules of an actual CHILL program.

2.2 Vocabulary

Programs are represented using the CHILL character set (see Appendix I) except for wide character literals, wide character string literals and comments. The representation of a CHILL program is not specified, which means that it is also possible to use a multi-byte character representation. The CHILL alphabet is represented by the syntactic category <character>, from which any character that is in the CHILL character set can be derived as a terminal production. The characters of UCS-2 level 1 are represented by the syntactic category <wide character>, from which any character that is in the UCS-2 level 1 set can be derived as a terminal production.

The lexical elements of CHILL are:

- special symbols;
- simple name strings;
- literals.

The special symbols are listed in Appendix II. They can be formed by a single character or by character combinations.

Simple name strings are formed according to the following syntax:

**syntax:**

\[
\begin{align*}
<\text{simple name string}> & ::= <\text{letter}> \{ <\text{letter}> | <\text{digit}> | _ \}^* \quad (1) \\
<\text{letter}> & ::= A | B | C | D | E | F | G | H | I | J | K | L | M \quad (2) \\
& | N | O | P | Q | R | S | T | U | V | W | X | Y | Z \\
& | a | b | c | d | e | f | g | h | i | j | k | l | m \quad (2.1) \\
& | n | o | p | q | r | s | t | u | v | w | x | y | z \\
<\text{digit}> & ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \quad (3)
\end{align*}
\]
semantics: The underline character (_) forms part of the simple name string; e.g. the simple name string life_time is different from the simple name string lifetime. Lower case and upper case letters are different, e.g. Status and status are two different simple name strings.

The language has a number of special simple name strings with predetermined meanings (see Appendix III). Some of them are reserved, i.e. they cannot be used for other purposes.

The special simple name strings in a piece must either all be in upper case representation or all be in lower case representation. The reserved simple name strings are only reserved in the chosen representation (e.g. if the lower case fashion is chosen, row is reserved, ROW is not).

static conditions: A simple name string may not be one of the reserved simple name strings (see Appendix III.1).

2.3 The use of spaces

A sequence of one or more spaces is allowed before and after each lexical element. Such a sequence is called a delimiter. Lexical elements are also terminated by the first character that cannot be part of the lexical element. For instance, IFBTHEN will be considered a simple name string and not as the beginning of an action IF B THEN, // will be considered as the concatenation symbol (/) followed by an asterisk (*) and not as a divide symbol (/) followed by a comment opening bracket (/(*).

2.4 Comments

syntax:

<comment> ::= (1)
    <bracketed comment> (1.1)
    | <line-end comment> (1.2)

<bracketed comment> ::= (2)
    /* <character string> */ (2.1)

<line-end comment> ::= (3)
    -- <character string> <end-of-line> (3.1)

<character string> ::= (4)
    { <character> }* (4.1)
    | { <wide character> }* (4.2)

NOTE – End-of-line denotes the end of the line in which the comment occurs.

semantics: A comment conveys information to the reader of a program. It has no influence on the program semantics.

A comment may be inserted at all places where spaces are allowed as delimiters.

A bracketed comment is terminated by the first occurrence of the special sequence: */. A line-end comment is terminated by the first occurrence of the end of the line.

examples:

4.1 /* from collected algorithms from CACM no. 93 */ (2.1)

2.5 Format effectors

The format effectors BS (Backspace), CR (Carriage return), FF (Form feed), HT (Horizontal tabulation), LF (Line feed), VT (Vertical tabulation) of the CHILL character set (see Appendix I, positions FE0 to FE5) and the end-of-line are not mentioned in the CHILL context-free syntax description. When used, they have the same delimiting effect as a space. Spaces and format effectors may not occur within lexical elements (except character string literals).
2.6 Compiler directives

syntax:

<directive clause> ::= <directive> { , <directive> }* <>
(1)
<directive> ::= <implementation directive>
(2.1)

semantics: A directive clause conveys information to the compiler. This information is specified in an implementation defined format.

An implementation directive must not influence the program semantics, i.e. a program with implementation directives is correct, in the CHILL sense, if and only if it is correct without these directives.

A directive clause is terminated by the first occurrence of the directive ending symbol (<>). A directive may contain any character of the character set (see Appendix I).

static properties: A directive clause may be inserted at any place where spaces are allowed as delimiters. It has the same delimiting effect as a space. The names used in a directive clause follow an implementation defined name binding scheme which does not influence the CHILL name binding rules (see 12.2).

2.7 Names and their defining occurrences

syntax:

<name> ::= <name string>
(1)
  | <qualified name>
(1.2)
  | <moreta component name>
(1.3)

<name string> ::= <simple name string>
(2)
  | <prefixed name string>
(2.2)

<prefixed name string> ::= <prefix> ! <simple name string>
(3)

<prefix> ::= <simple prefix> { ! <simple prefix> }*
(4)

<simple prefix> ::= <simple name string>
(5)

<defining occurrence> ::= <simple name string>
(6)

<defining occurrence list> ::= <defining occurrence> { , <defining occurrence> }*
(7)

<set element name> ::= <simple name string>
(8)

<set element name defining occurrence> ::= <simple name string>
(9)

[field name] ::= <simple name string>
(10)

[field name defining occurrence] ::= <simple name string>
(11)

[field name defining occurrence list] ::= <field name defining occurrence> { , <field name defining occurrence> }*
(12)
<exception name> ::= (13)
  <simple name string> (13.1)
  | <prefixed name string> (13.2)
<text reference name> ::= (14)
  <simple name string> (14.1)
  | <prefixed name string> (14.2)
<component name> ::= (15)
  <simple name string> (15.1)
<component name defining occurrence> ::= (16)
  <simple name string> (16.1)
<qualified name> ::= (17)
  <simple name string> ! <component name> (17.1)
<moreta component name> ::= (18)
  <moreta location> . { <simple name string> | <qualified name> } (18.1)

**semantics:** Names in a program denote objects. Given an occurrence of a name (formally: an occurrence of a terminal production of name) in a program, the binding rules of 12.2 provide defining occurrences (formally: occurrences of terminal productions of defining occurrence) to which that (occurrence of) name is bound. The name then denotes the object defined or declared by the defining occurrences. (There can be more than one defining occurrence for a name in the case of names with quasi defining occurrences and in the case of names of components of moreta modes.)

Defining occurrences are said to define the name. A name is said to be an applied occurrence of the name created by the defining occurrence to which it is bound. The name has its rightmost simple name string equal to that of the name.

Similarly, field names are bound to field name defining occurrences and denote the fields (of a structure mode) defined by those field name defining occurrences. Moreta component names are bound to component defining occurrences and denote the components (of a moreta mode) defined by those component name defining occurrences.

Exception names are used to identify exception handlers according to the rules stated in clause 8.

Text reference names are used to identify descriptions of pieces of source text in an implementation defined way, subject to the rules in 10.10.1.

When a name is bound to more than one defining occurrence, each of the defining occurrences to which the name is bound defines or declares the same object (see 10.10 and 12.2.2 for precise rules).

**Qualified names** are used to identify components of moreta modes.

**definition of notation:** Given a name string NS, and a string of characters P, which is either a prefix or is empty, the result of prefixing NS with P, written P ! NS, is defined as follows:

- if P is empty, then P ! NS is NS;
- otherwise P ! NS is the name string obtained by concatenating all the characters in P, a prefixing operator and all the characters in NS.

For example, if P is "q / r" and NS is "s / n" then P ! NS is "q / r ! s ! n".

**static properties:** Each simple name string has a canonical name string attached which is the simple name string itself. A name string has a canonical name string attached which is:

- if the name string is a simple name string, then the canonical name string of that simple name string;
- if the name string is a prefixed name string, then the concatenation in left to right order of all simple name strings in the name string, separated by prefixing operators, i.e. interspersed spaces, comments and format effectors (if any) are left out.
In the rest of this Recommendation | International Standard:

- the name string of a name, exception name or text reference name is used to denote the canonical name string of the name string in that name, exception name or text reference name, respectively;

- the name string of a defining occurrence, field name, field name defining occurrence, moreta component name or moreta component defining occurrence is used to denote the canonical name string of the simple name string in that defining occurrence, field name, field name defining occurrence, moreta component name or moreta component defining occurrence, respectively.

The binding rules are such that:

- names with a simple name string are bound to defining occurrences with the same name string;

- names with a prefixed name string are bound to defining occurrences with the same name string as the rightmost simple name string in the prefixed name string of the name;

- field names are bound to field name defining occurrences with the same name string as the field names.

- moreta component names are bound to moreta component name defining occurrences with the same name string as the moreta component names.

A name inherits all the static properties attached to the name defined by the defining occurrence to which it is bound. A field name inherits all static properties attached to the field name defined by the field name defining occurrence to which it is bound. A moreta component name inherits all static properties attached to the moreta component name defined by the moreta component name defining occurrence to which it is bound.

**static conditions:** The simple name string denoted in a qualified name and followed by ! must be a moreta mode name.

If a qualified name of the form "M ! component name" occurs outside the definition of the moreta mode M, then the component name must be the name of a SYN, a SYNMODE, or a NEWMODE component of M.

3 Modes and classes

3.1 General

A location has a mode attached to it; a value has a class attached to it. The mode attached to a location defines the set of values that may be contained in the location, the access methods of the location and the allowed operations on the values. The class attached to a value is a means of determining the modes of the locations that may contain the value. Some values are strong. A strong value has a class and a mode attached. Strong values are required in those value contexts where mode information is needed.

3.1.1 Modes

CHILL has static modes (i.e. modes for which all properties are statically determinable) and dynamic modes (i.e. modes for which some properties are only known at run time). Dynamic modes are always parameterized modes with run-time parameters.

Static modes are terminal productions of the syntactic category mode.

Modes are also parameterized by values not explicitly denoted in the program text.

3.1.2 Classes

Classes have no denotation in CHILL.

The following kinds of classes exist and any value in a CHILL program has a class of one of these kinds:
For a mode M there exists the M-value class. All values with such a class and only those values are strong and the mode attached to the value is M.

- For a mode M there exists the M-derived class.
- For any mode M there exists the M-reference class.
- The null class.
- The all class.

The last two classes are constant classes, i.e. they do not depend on a mode M. A class is said to be dynamic if and only if it is an M-value class, an M-derived class, or an M-reference class, where M is a dynamic mode.

### 3.1.3 Properties of, and relations between, modes and classes

Modes in CHILL have properties. These may be hereditary or non-hereditary properties. A hereditary property is inherited from a defining mode to a mode name defined by it. Below a summary is given of the properties that apply to all modes (except for the first, they are all defined in 12.1):

- A mode has a novelty (defined in 3.2.2, 3.2.3 and 3.3).
- A mode can have the read-only property.
- A mode can be parameterizable.
- A mode can have the referencing property.
- A mode can have the tagged parameterized property.
- A mode can have the non-value property.

Classes in CHILL may have the following properties (defined in 12.1):

- A class can have a root mode.
- One or more classes may have a resulting class.

Operations in CHILL are determined by the modes and classes of locations and values. This is expressed by the mode checking rules which are defined in 12.1 as a number of relations between modes and classes. There exists the following relations:

- Two modes can be similar.
- Two modes can be v-equivalent.
- Two modes can be equivalent.
- Two modes can be l-equivalent.
- Two modes can be alike.
- Two modes can be novelty bound.
- Two modes can be read-compatible.
- Two modes can be dynamic read-compatible.
- Two modes can be dynamic equivalent.
- A mode can be restrictable to a mode.
- A mode can be compatible with a class.
- A class can be compatible with a class.

### 3.2 Mode definitions

#### 3.2.1 General

**syntax:**

\[
\begin{align*}
<\text{mode definition}> & := \\
<\text{defining occurrence list}> = <\text{defining mode}> & (1) \\
<\text{defining occurrence list}> & := \\
<\text{defining mode}> & := \ \\
<\text{mode}> & (2)
\end{align*}
\]
derived syntax: A mode definition where the defining occurrence list consists of more than one defining occurrence is derived from several mode definitions, one for each defining occurrence, separated by commas, with the same defining mode. For example:

\[
\text{NEWMODE \(dollar, pound = INT;\)}
\]
is derived from:

\[
\text{NEWMODE \(dollar = INT, pound = INT;\)}
\]

semantics: A mode definition defines a name that denotes the specified mode. Mode definitions occur in synmode and newmode definition statements. A synmode is synonymous with its defining mode. A newmode is not synonymous with its defining mode. The difference is defined in terms of the property novelty, that is used in the mode checking (see 12.1).

static properties: A defining occurrence in a mode definition defines a mode name.

Predefined mode names, implementation defined integer mode names and implementation defined floating point mode names (if any, see 3.4.2 and 3.5.1) are also mode names.

A mode name has a defining mode which is the defining mode in the mode definition which defines it. (For predefined and implementation defined mode names this defining mode is a virtual mode.) The hereditary properties of a mode name are those of its defining mode.

A set of recursive definitions is a set of mode definitions or synonym definitions (see 5.1) such that the defining mode in each mode definition or constant value or mode in each synonym definition is, or directly contains, a mode name or a synonym name defined by a definition in the set.

A set of recursive mode definitions is a set of recursive definitions having only mode definitions.

Any mode being or containing a mode name defined in a set of recursive mode definitions is said to denote a recursive mode. A path in a set of recursive mode definitions is a list of mode names, each name indexed with a marker such that:

- all names in the path have a different definition;
- for each name, its successor is or directly occurs in its defining mode (the successor of the last name is the first name);
- the marker indicates uniquely the position of the name in the defining mode of its predecessor (the predecessor of the first name is the last name).

[Example: \(\text{NEWMODE } M = \text{STRUCT } (i M, n \text{ REF } M);\) contains two paths: \(\{M_i\}\) and \(\{M_n\}\).]

A path is safe if and only if at least one of its names is contained in a reference mode, a row mode, or a procedure mode at the marked place.

static conditions: For any set of recursive mode definitions, all its paths must be safe. (The first path of the example above is not safe.)

examples:

1.15 \(\text{operand_mode} = \text{INT}\)  \hspace{1cm} (1.1)

3.3 \(\text{complex} = \text{STRUCT } (\text{re,im FLOAT})\) \hspace{1cm} (1.1)

3.2.2 Synmode definitions

syntax:

\[
<\text{synmode definition statement}>::= (1.1)
\]

\[
\text{SYNMODE} <\text{mode definition}> \{, <\text{mode definition}> \}^* ; \hspace{1cm} (1.1)
\]

\[
| <\text{remote program unit}> \hspace{1cm} (1.2)
\]

semantics: A synmode definition statement defines mode names which are synonymous with their defining mode.
**static properties**: A defining occurrence in a mode definition in a synmode definition statement defines a synmode name (which is also a mode name). A synmode name is said to be synonymous with a mode M (conversely, M is said to be synonymous with the synmode name) if and only if:

- either M is the defining mode of the synmode name;
- or the defining mode of the synmode name is itself a synmode name synonymous with M.

Two mode names A and B are synonymous if and only if:

- either A and B are the same name;
- or A is the defining mode of B and B is a synmode name;
- or B is the defining mode of A and A is a synmode name;
- or the defining mode name of A is synonymous to B and A is a synmode name;
- or the defining mode name of B is synonymous to A and B is a synmode name.

The novelty of a synmode name is that of its defining mode.

If the defining mode is a discrete range mode or a floating point range mode, then the parent mode of the synmode name is that of its defining mode. If the defining mode is a varying string mode, then the component mode of the synmode name is that of its defining mode.

**examples**:

6.3  SYNMODE month = SET (jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec);  
     (1.1)

3.2.3  Newmode definitions

**syntax**:

\[
\text{<newmode definition statement>} ::= \\
\quad \text{NEWMODE} <\text{mode definition}> \{, <\text{mode definition}>\}^*; \\
\quad |\quad \text{<remote program unit>} \\
\]  

**semantics**: A newmode definition statement defines mode names which are not synonymous with their defining mode.

**static properties**: A defining occurrence in a mode definition in a newmode definition statement defines a newmode name (which is also a mode name).

The novelty of the newmode name is the defining occurrence which defines it. If the defining mode of the newmode name is a discrete range mode or a floating point range mode, then the virtual mode &name is introduced as the parent mode of the newmode name. The defining mode of &name is the parent mode of the discrete range mode or the one of the floating point range mode, and the novelty of &name is that of the newmode name.

If the defining mode is a varying string mode, then the virtual mode &name is introduced as the component mode of the newmode name. The defining mode of &name is the component mode of the varying string mode, and the novelty of &name is that of the newmode name.

If the defining occurrence of the mode definition is a quasi defining occurrence, then the novelty is a quasi novelty, otherwise it is a real novelty.

**static conditions**: If the novelty is a quasi novelty, then at most one real novelty must be novelty bound to it.

**examples**:

11.6  NEWMODE line = INT (1:8);  
     (1.1)  

11.12 NEWMODE board = ARRAY (line) ARRAY (column) square;  
      (1.1)
3.3 Mode classification

syntax:

```
<mode> ::= (1)
  [ READ ] <non-composite mode> (1.1)
  | [ READ ] <composite mode> (1.2)
  | <formal generic mode indication> (1.3)

<non-composite mode> ::= (2)
  <discrete mode> (2.1)
  | <real mode> (2.2)
  | <powerset mode> (2.3)
  | <reference mode> (2.4)
  | <procedure mode> (2.5)
  | <instance mode> (2.6)
  | <synchronization mode> (2.7)
  | <input-output mode> (2.8)
  | <timing mode> (2.9)
```

semantics: A mode defines a set of values and the operations which are allowed on the values. A mode may be a read-only mode, indicating that a location of that mode may not be accessed to store a value. A mode has a novelty, indicating whether it was introduced via a newmode definition statement or not.

static properties: A mode has the following hereditary properties:

- It is a read-only mode if it is an explicit or an implicit read-only mode.
- It is an explicit read-only mode if READ is specified or it is a parameterized array mode, a parameterized string mode or a parameterized structure mode, where the origin array mode name, origin string mode name or origin variant structure mode name, respectively, in it is a read-only mode.
- It is an implicit read-only mode if it is not an explicit read-only mode and if:
  - it is the element mode of a read-only string mode or a read-only array mode (see 3.13.2 and 3.13.3);
  - it is a field mode of a read-only structure mode or it is the mode of a tag field of a parameterized structure mode (see 3.13.4).

A mode has the same properties as the non-composite mode or composite mode in it. In the following sections, the properties are defined for predefined mode names and for modes that are not mode names; the properties of mode names are defined in 3.2. Read-only modes have the same properties as their corresponding non-read-only modes except for the read-only property (see 12.1.1.1).

A mode has the following non-hereditary properties:

- A novelty that is either nil or the defining occurrence in a mode definition in a newmode definition statement. The novelty of a mode which is not a mode name (nor READ mode name) is defined as follows:
  - if it is a parameterized string mode, a parameterized array mode or a parameterized structure mode, its novelty is that of its origin string mode, origin array mode or origin variant structure mode, respectively;
  - if it is a discrete range mode or a floating point range mode, its novelty is that of its parent mode;
  - otherwise its novelty is nil.

The novelty of a mode that is a mode name (READ mode name) is defined in 3.2.2 and 3.2.3.

- A size that is the value delivered by SIZE (&M), where &M is a virtual synmode name synonymous with the mode.
3.4 Discrete modes

3.4.1 General

syntax:

```
<discrete mode> ::= (I)
    <integer mode> (1.1)
    | <boolean mode> (1.2)
    | <character mode> (1.3)
    | <set mode> (1.4)
    | <discrete range mode> (1.5)
```

semantics: A discrete mode defines sets and subsets of totally-ordered values.

3.4.2 Integer modes

syntax:

```
<integer mode> ::= (I)
    <integer mode name> (1.1)
```

predefined names: The name INT is predefined as an integer mode name.

semantics: An integer mode defines a set of signed integer values between implementation defined bounds over which the usual ordering and arithmetic operations are defined (see 5.3). An implementation may define other integer modes with different bounds (e.g. LONG_INT, SHORT_INT, UNSIGNED_INT) that may also be used as parent modes for ranges (see 13.2). The &INT mode is introduced as the virtual mode that contains all the values of all predefined integer modes defined by the implementation. The internal representation of an integer value is the integer value itself. Note that &INT is not a predefined mode (although it may have the same bounds as those of a predefined integer mode).

static properties: An integer mode has the following hereditary properties:

- An upper bound and a lower bound which are the literals denoting respectively the highest and lowest value defined by the integer mode. They are implementation defined.
- A number of values which is upper bound – lower bound + 1.

examples:

```
1.5 INT (1.1)
```

3.4.3 Boolean modes

syntax:

```
<boolean mode> ::= (I)
    <boolean mode name> (1.1)
```

predefined names: The name BOOL is predefined as a boolean mode name.

semantics: A boolean mode defines the logical truth values (TRUE and FALSE), with the usual boolean operations (see 5.3). The internal representations of FALSE and TRUE are the integer values 0 and 1, respectively. This representation defines the ordering of the values.

static properties: A boolean mode has the following hereditary properties:

- An upper bound which is TRUE, and a lower bound which is FALSE.
- A number of values which is 2.

examples:

```
5.4 BOOL (1.1)
```
3.4.4 Character modes

syntax:

```
<character mode> ::= (1)
  <character mode name> (1.1)
```

predefined names: The names CHAR and WCHAR are predefined as character mode names.

semantics: A character mode defines the character values as described by the CHILL character set (see Appendix I) in case of CHAR or by ISO/IEC 10646-1 in case of WCHAR. These alphabets define the ordering of the characters and the integer values which are their internal representations.

static properties: A character mode has the following hereditary properties:

- An upper bound and a lower bound which are the character literals denoting respectively the highest and lowest value defined by CHAR or WCHAR respectively.
- A number of values which is 256 in case of CHAR, and which is given in ISO/IEC 10646-1 in case of WCHAR.

examples:

8.4 CHAR (1.1)

3.4.5 Set modes

syntax:

```
<set mode> ::= (1)
  SET ( <set list> ) (1.1)
| <set mode name> (1.2)

<set list> ::= (2)
  <numbered set list> (2.1)
| <unnumbered set list> (2.2)

<numbered set list> ::= (3)
  <numbered set element> { , <numbered set element>}* (3.1)

<numbered set element> ::= (4)
  <set element name defining occurrence> = <integer literal expression> (4.1)

<numbered set element> ::= (5)
  <set element> { , <set element>}* (5.1)

<set element> ::= (6)
  <set element name defining occurrence> (6.1)
```

semantics: A set mode defines a set of named and unnamed values. The named values are denoted by the names defined by defining occurrences in the set list; the unnamed values are the other values. The internal representation of the named values is the integer value associated with them. This representation defines the ordering of the values.

The maximum number of values of a set mode is implementation defined.

static properties: A defining occurrence in a set list defines a set element name. A set element name has a set mode attached, which is the set mode.

A set mode has the following hereditary properties:

- A set of set element names which is the set of names defined by defining occurrences in its set list.
- Each set element name of a set mode has an internal representation value attached which is, in the case of a numbered set element, the value delivered by the integer literal expression in it; otherwise one of the values 0, 1, 2, etc., according to its position in the unnumbered set list. For example in: SET (a, b), a has representation value 0, and b has representation value 1 attached.
- An upper bound and a lower bound which are its set element names with the highest and lowest representation values, respectively.
• A **number of values** which is the highest of the values attached to the **set element** names plus 1.

• It is a **numbered** set mode if the **set list** in it is a **numbered set list**; otherwise it is an **unnumbered** set mode.

**static conditions:** For each pair of **integer literal expressions** \( e_1 \), \( e_2 \) in the **set list** \( \text{NUM}(e_1) \) and \( \text{NUM}(e_2) \) must deliver different non-negative results.

**examples:**

11.7  **SET** (occupied, free)  \( (1.1) \)

6.3  **month**  \( (1.2) \)

**3.4.6 Discrete range modes**

**syntax:**

\[
\begin{align*}
\text{<discrete range mode>} & \::= \\
\quad & \text{<discrete mode name> \ ( <literal range> )} \\
\quad & \text{RANGE \ ( <literal range> )} \\
\quad & \text{BIN \ ( <integer literal expression> )} \\
\quad & \text{<discrete range mode name>}
\end{align*}
\]

\[
\begin{align*}
\text{<literal range>} & \::= \\
\quad & \text{<lower bound> : <upper bound>}
\end{align*}
\]

\[
\begin{align*}
\text{<lower bound>} & \::= \\
\quad & \text{<discrete literal expression>}
\end{align*}
\]

\[
\begin{align*}
\text{<upper bound>} & \::= \\
\quad & \text{<discrete literal expression>}
\end{align*}
\]

**derived syntax:** The notation **BIN** \((n)\) is derived from **RANGE** \((0 : 2^n - 1)\), e.g. **BIN** \((2+1)\) stands for **RANGE** \((0 : 7)\).

**semantics:** A discrete range mode defines the set of values ranging between the bounds specified (bounds included) by the **literal range**. The range is taken from a specific **parent** mode that determines the operations on and ordering of the range values.

**static properties:** A discrete range mode has the following non-hereditary property: it has a **parent** mode, defined as follows:

• If the discrete range mode is of the form:

\[
\text{<discrete mode name> \ ( <literal range> )}
\]

then if the **discrete mode name** is not a discrete range mode, the **parent** mode is the **discrete mode name**; otherwise it is the **parent** mode of the **discrete mode name**.

• If the discrete range mode is of the form:

\[
\text{RANGE \ ( <literal range> )}
\]

then the **parent** mode depends on the **resulting class** of the classes of the **upper bound** and **lower bound** in the **literal range**:

– if it is an M-derived class, where M is an integer mode, then the **parent** mode is a **predefined** integer mode chosen by the implementation such that it contains the range of values delivered by **literal range**;

– otherwise it is the **root** mode of the **resulting class**.

• If the discrete range mode is a **discrete range mode name** which is a **synmode** name, then its **parent** mode is that of the **defining** mode of the **synmode** name; otherwise it is a **newmode** name and then its **parent** mode is the virtually introduced **parent** mode (see 3.2.3).

A discrete range mode has the following hereditary properties:

• An **upper bound** and a **lower bound** which are the literals denoting the values delivered by **lower bound** and **upper bound**, respectively, in the **literal range**.

• A **number of values** which is the value delivered by \( \text{NUM}(U) - \text{NUM}(L) + 1 \), where \( U \) and \( L \) denote respectively the **upper bound** and **lower bound** of the discrete range mode.

• It is a **numbered** range mode if its **parent** mode is a **numbered** set mode.
static conditions: The classes of upper bound and lower bound must be compatible and both must be compatible with the discrete mode name, if specified.

Lower bound must deliver a value that is less than or equal to the value delivered by upper bound, and both values must belong to the set of values defined by discrete mode name, if specified.

The integer literal expression in case of BIN must deliver a non-negative value.

If the parent mode is an integer mode, there must exist a predefined integer mode that contains the set of values included between the lower bound and the upper bound.

If the discrete range mode is of the form:

RANGE ( <literal range> ) or <discrete mode name> ( <literal range> )

then the evaluation of the 1. lower bound, 2. upper bound, must not depend directly or indirectly on the value of the 1. lower bound, 2. upper bound of the discrete range mode. If the discrete range mode is of the form:

BIN ( <integer literal expression> )

then the evaluation of the integer literal expression must not depend directly or indirectly on the value of the upper bound of the discrete range mode.

examples:

9.5 INT (2:max) (1.1)
11.12 line (1.4)

3.5 Real modes

syntax:

<real mode> ::= 
| <floating point mode> (1.1)
| <floating point range mode> (1.2)

semantics: A real mode specifies a set of numerical values which approximate a continuous range of real numbers.

3.5.1 Floating point modes

syntax:

<floating point mode> ::= 
| <floating point mode name> (1.1)

predefined names: The name FLOAT is predefined as a floating point mode name.

semantics: A floating point mode defines a set of numeric approximations to a range of real values, together with their minimum relative accuracy, between implementation defined bounds, over which the usual ordering and arithmetic operations are defined (see 5.3). This set contains only the values which can be represented by the implementation. An implementation may define other floating point modes with different bounds and/or precision (e.g. LONG_FLOAT, SHORT_FLOAT) that may also be used as parent modes for ranges (see 13.3). The &FLOAT mode is introduced as the virtual mode that contains all the values of all predefined floating point modes defined by the implementation. The internal representation of a floating point value is the floating point value itself. Note that &FLOAT is not a predefined mode (although it may have the same bounds as those of a predefined floating point mode).

static properties: A floating point mode has the following hereditary properties:

- An upper bound and a lower bound which are the literals denoting respectively the highest and lowest value defined by the floating point mode. They are implementation defined.
- A precision which is the maximum number of significant decimal digits defined by the mode.
• A positive lower limit and a negative upper limit which are the literals denoting respectively the smallest positive value and the largest negative value exactly representable in the floating point mode, zero excluded.

dependencies:

examples:

FLOAT

3.5.2 Floating point range modes

syntax:

<floating point range mode> ::= (1)
  <floating point mode name> ( <float value range> ) (1.1)
  | RANGE ( <float value range> [ , <significant digits> ] ) (1.2)
  | <floating point range mode name> (1.3)

<float value range> ::= (2)
  <lower float bound> : <upper float bound> (2.1)

<lower float bound> ::= (3)
  <floating point literal expression> (3.1)

<upper float bound> ::= (4)
  <floating point literal expression> (4.1)

<significant digits> ::= (5)
  <integer literal expression> (5.1)

semantics: A floating point range mode defines the set of values ranging between the bounds specified (bounds included) by float value range with the number of significant digits specified by significant digits. The range is taken from a specific parent mode that determines the operations on and ordering of the range values. For example, RANGE ( –10.0E1 : 10.0E1, 2) denotes the values: –10.0, –9.9, ..., –0.11, –0.1, 0, 0.1, ..., 10.0.

static properties: A floating point range mode has the following non-hereditary property: it has a parent mode, defined as follows:

• If the floating point range mode is of the form:

  <floating point mode name> ( <float value range> )

then if the floating point mode name is not a floating point range mode, the parent mode is the floating point mode name; otherwise it is the parent mode of the floating point mode name.

• If the floating point range mode is of the form:

  RANGE ( <float value range> [ , <significant digits> ] )

then the parent mode depends on the resulting class of the classes of the upper float bound and lower float bound in the literal range:

  – if it is an M-derived class, where M is a floating point mode, then the parent mode is a predefined floating point mode chosen by the implementation such that it contains the range of values delivered by float value range, with the precision defined below;

  – otherwise it is the root mode of the resulting class.

• If the floating point range mode is a floating point range mode name which is a synmode name, then its parent mode is that of the defining mode of the synmode name; otherwise it is a newmode name and then its parent mode is the virtually introduced parent mode (see 3.2.3).

A floating point range mode has the following hereditary properties:

• An upper bound and a lower bound which are the literals denoting the values delivered by lower float bound and upper float bound, respectively, in the float value range.

• A precision which is, if the floating point range mode is of the form:

  RANGE ( <float value range> [ , <significant digits> ] )

  – the value delivered by significant digits if specified;

  – otherwise the greatest precision of the precisions of lower float bound and upper float bound.

Otherwise it is that of the floating point mode name or the floating point range mode name.
**static conditions:** Lower float bound must deliver a value that is less than or equal to the value delivered by upper float bound, and both values must belong to the set of values defined by floating point mode name, if specified.

There must exist a predefined floating point mode that contains both upper bound and lower bound with the specified precision.

The value delivered by significant digit must be greater than zero.

The evaluation of the 1.lower float bound, 2.upper float bound, must not depend directly or indirectly on the value of the 1.lower bound, 2.upper bound of the floating point range mode.

### 3.6 Powerset modes

**syntax:**

\[
\text{<powerset mode>} ::= \\
\text{POWERSET <member mode>} \quad (1.1) \\
| \text{<powerset mode name>} \quad (1.2)
\]

**semantics:** A powerset mode defines values that are sets of values of its member mode. Powerset values range over all subsets of the member mode. The usual set-theoretic operators are defined on powerset values (see 5.3).

The maximum number of values of the member mode is implementation defined.

**static properties:** A powerset mode has the following hereditary property:

- A member mode which is the member mode.

**examples:**

8.4 \text{POWERSET CHAR} \quad (1.1)

9.5 \text{POWERSET INT (2:max)} \quad (1.1)

9.6 \text{number_list} \quad (1.2)

### 3.7 Reference modes

#### 3.7.1 General

**syntax:**

\[
\text{<reference mode>} ::= \\
\text{<bound reference mode>} \quad (1.1) \\
| \text{<free reference mode>} \quad (1.2) \\
| \text{<row mode>} \quad (1.3)
\]

**semantics:** A reference mode defines references (addresses or descriptors) to referable locations. By definition, bound references refer to locations of a given static mode or a set of related moreta modes; free references may refer to locations of any static mode; rows refer to locations of a dynamic mode.

The dereferencing operation is defined on reference values (see 4.2.3, 4.2.4 and 4.2.5), delivering the location that is referenced.

Two reference values are equal if and only if they both refer to the same location, or both do not refer to a location (i.e. they are the value NULL).

#### 3.7.2 Bound reference modes

**syntax:**

\[
\text{<bound reference mode>} ::= \\
\text{REF <referenced mode>} \quad (1.1) \\
| \text{<bound reference mode name>} \quad (1.2)
\]

\[
\text{<referenced mode>} ::= \\
\text{<mode>} \quad (2)
\]
semantics: A bound reference mode defines reference values to locations of the specified referenced mode. If the referenced mode is a non-moreta mode M then the bound reference mode defines reference values to locations of M. If the referenced mode is a moreta mode MM then the bound reference mode defines reference values to locations of MM or any successor of MM.

static properties: A bound reference mode has the following hereditary property:

• A referenced mode which is the referenced mode.

examples:

10.42 REF cell (1.1)

3.7.3 Free reference modes

syntax:

<free reference mode> ::= (1)
<free reference mode name> (1.1)

predefined names: The name PTR is predefined as a free reference mode name.

semantics: A free reference mode defines reference values to locations of any static mode.

examples:

19.8 PTR (1.1)

3.7.4 Row modes

syntax:

<row mode> ::= (1)
ROW <string mode> (1.1)
ROW <array mode> (1.2)
ROW <variant structure mode> (1.3)
<row mode name> (1.4)

semantics: A row mode defines reference values to locations of dynamic mode (which are locations of some parameterized mode with non constant parameters).

A row value may refer to:

• string locations with non constant string length;
• array locations with non constant upper bound;
• parameterized structure locations with non constant parameters.

static properties: A row mode has the following hereditary property:

• A referenced origin mode which is the string mode, the array mode, or the variant structure mode, respectively.

static condition: The variant structure mode must be parameterizable.

examples:

8.6 ROW CHARS (max) (1.1)

3.8 Procedure modes

syntax:

<procedure mode> ::= (1)
PROC ( [ <parameter list> ] ) [ <result spec> ]
[ EXCEPTIONS ( <exception list> ) ] (1.1)
| <procedure mode name> (1.2)
semantics: A procedure mode defines (general) procedure values, i.e. the objects denoted by general procedure names that are names defined in procedure definition statements. Procedure values indicate pieces of code in a dynamic context. Procedure modes allow for manipulating a procedure dynamically, e.g. passing it as a parameter to other procedures, sending it as message value to a buffer, storing it into a location, etc.

Procedure values can be called (see 6.7).

Two procedure values are equal if and only if they denote the same procedure in the same dynamic context, or if they both denote no procedure (i.e. they are the value NULL).

static properties: A procedure mode has the following hereditary properties:

• A list of parameter specs, each consisting of a mode and possibly a parameter attribute. The parameter specs are defined by the parameter list.

• An optional result spec, consisting of a mode and an optional result attribute. The result spec is defined by the result spec.

• A possibly empty list of exception names which are those mentioned in the exception list.

static conditions: All names mentioned in exception list must be different.

If LOC is specified in the parameter spec or in the result spec, the mode in it may have the non-value property.

If DYNAMIC is specified in the parameter spec or in the result spec, the mode in it must be parameterizable.

3.9 Instance modes

syntax:

\[
\text{<instance mode>} ::= \quad (1) \\
\quad \text{<instance mode name>} \quad (1.1)
\]

predefined names: The name INSTANCE is predefined as an instance mode name.

semantics: An instance mode defines values which identify processes. The creation of a new process (see 5.2.15, 6.13 and 11.1) yields a unique instance value as identification for the created process.

Two instance values are equal if and only if they identify the same process, or they both identify no process (i.e. they are the value NULL).

examples:

15.39 INSTANCE (1.1)
3.10 Synchronization modes

3.10.1 General

syntax:

<synchronization mode> ::= (1)
  <event mode> (1.1)
| <buffer mode> (1.2)

semantics: A synchronization mode provides a means for synchronization and communication between processes (see clause 11). There exists no expression in CHILL denoting a value defined by a synchronization mode. As a consequence, there are no operations defined on the values.

3.10.2 Event modes

syntax:

<event mode> ::= (1)
  EVENT [ ( <event length> ) ] (1.1)
| <event mode name> (1.2)
<event length> ::= (2)
  <integer literal expression> (2.1)

semantics: An event mode location provides a means for synchronization between processes. The operations defined on event mode locations are the continue action, the delay action and the delay case action, which are described in 6.15, 6.16 and 6.17, respectively.

The event length specifies the maximum number of processes that may become delayed on an event location; that number is unlimited if no event length is specified.

An event mode location which contains the undefined value is an "empty" event, i.e. no delayed processes are attached to it.

static properties: An event mode has the following hereditary property:
• An optional event length which is the value delivered by event length.

static conditions: The event length must deliver a positive value.

The evaluation of the event length must not depend directly or indirectly on the value of the event length of the event mode.

examples:

14.10 EVENT (1.1)

3.10.3 Buffer modes

syntax:

<buffer mode> ::= (1)
  BUFFER [ ( <buffer length> ) ] buffer element mode (1.1)
| <buffer mode name> (1.2)
<buffer length> ::= (2)
  <integer literal expression> (2.1)
<buffer element mode> ::= (3)
  <mode> (3.1)

semantics: A buffer mode location provides a means for synchronization and communication between processes. The operations defined on buffer locations are the send action and the receive case action, described in 6.18 and 6.19, respectively.

The buffer length specifies the maximum number of values that can be stored in a buffer location; that number is unlimited if no buffer length is specified.

A buffer mode location which contains the undefined value is an "empty" buffer, i.e. no delayed processes are attached to it nor are there messages in the buffer.
static properties: A buffer mode has the following hereditary properties:

- An optional buffer length which is the value delivered by buffer length.
- A buffer element mode which is the buffer element mode.

static conditions: The buffer length must deliver a non-negative value.

The buffer element mode must not have the non-value property.

The evaluation of the buffer length must not depend directly or indirectly on the value of the buffer length of the buffer mode.

eamples:

16.30 BUFFER (1) user_messages
16.34 user_buffers

3.11 Input-Output Modes

3.11.1 General

syntax:

\[
<\text{input-output mode}> ::= \\
\quad <\text{association mode}> (1) \\
\quad | <\text{access mode}> (1.1) \\
\quad | <\text{text mode}> (1.2)
\]

semantics: An input-output mode provides a means for input-output operations as defined in clause 7. There exists no expression in CHILL denoting a value defined by an input-output mode. As a consequence, there are no operations defined on the values.

eamples:

20.17 ASSOCIATION

3.11.2 Association modes

syntax:

\[
<\text{association mode}> ::= \\
\quad <\text{association mode name}> (1)
\]

predefined names: The name ASSOCIATION is predefined as an association mode name.

semantics: An association mode location provides a means for representing a relation to an outside world object. Such a relation is called an association in CHILL; associations can be created by the built-in routine ASSOCIATE and be ended by DISSOCIATE.

An association mode location which contains the undefined value is "empty", i.e. it does not contain an association.

3.11.3 Access modes

syntax:

\[
<\text{access mode}> ::= \\
\quad \text{ACCESS} [ ( <\text{index mode}> ) ][ <\text{record mode}> [ \text{DYNAMIC} ] ] (1.1) \\
\quad | <\text{access mode name}> (1.2) \\
\quad | <\text{record mode}> ::= \\
\quad \quad <\text{mode}> (2.1) \\
\quad | <\text{index mode}> ::= \\
\quad \quad <\text{discrete mode}> (3.1) \\
\quad \quad | <\text{literal range}> (3.2)
\]

derived syntax: The index mode notation literal range is derived from the discrete mode RANGE (literal range).

semantics: An access mode location provides a means for positioning a file and for transferring values from a CHILL program to a file in the outside world, and vice versa.
An access mode may define a record mode; this record mode defines the root mode of the class of the values that can be transferred via a location of that access mode to or from a file. The mode of the transferred value may be dynamic, i.e. the size of the record may vary, when the attribute DYNAMIC is specified in the access mode denotation or when record mode is a varying string mode. In the latter case DYNAMIC need not be specified.

An access mode may also define an index mode; such an index mode defines the size of a "window" to (a part of) the file, from which it is possible to read (or write) records randomly. Such a window can be positioned in an (indexable) file by the connect operation. If no index mode is specified, then it is possible to transfer records only sequentially.

An access mode location which contains the undefined value is "empty", i.e. it is not connected to an association.

**static properties:** An access mode has the following hereditary properties:

- An optional record mode which is the record mode if present. It is a dynamic record mode if DYNAMIC is specified or if record mode is a varying string mode, otherwise it is a static record mode.
- An optional index mode which is the index mode.
- Optional upper bound and lower bound which are the upper bound and lower bound of the index mode, if present.

**static conditions:** The optional record mode must not have the non-value property.

If DYNAMIC is specified, the record mode must be parameterizable and must not be a tagless structure mode.

The index mode must neither be a numbered set mode nor a numbered range mode.

If the index mode is a literal range of the form:

\[
\text{<lower bound>} : \text{<upper bound>}
\]

then, the evaluation of the 1.lower bound, 2.upper bound, must not depend directly or indirectly on the value of the 1.lower bound, 2.upper bound of the access mode.

**examples:**

```
20.18   ACCESS (index_set) record_type       (1.1)
22.20   ACCESS string DYNAMIC               (1.1)
20.18   record_type                         (2.1)
20.18   index_set                           (3.1)
```

**3.11.4 Text modes**

**syntax:**

\[
<\text{text mode}> ::= \\
| <\text{narrow text mode}>  (1) \\
| <\text{wide text mode}>    (1.2)
\]

\[
<\text{narrow text mode}> ::= \\
  TEXT ( <\text{length}> ) [ <\text{index mode}> ] [ DYNAMIC ]    (2.1)
\]

\[
<\text{wide text mode}> ::= \\
  WTEXT ( <\text{length}> ) [ <\text{index mode}> ] [ DYNAMIC ]   (3.1)
\]

\[
<\text{length}> ::= \\
  <\text{integer literal expression}> (4)
\]

**semantics:** A text mode location provides a means for transferring values represented in human-readable form from a CHILL program to a file in the outside world, and vice versa. A text mode location has a text record sub-location and an access sub-location. The text record sub-location is initialized with an empty string.

A text mode has a text length, which defines the maximum length of the records that can be transferred, and possibly an index mode that has the same meaning as for access modes. The actual length attribute of a text mode location is the actual length of its text record.
A text mode location which contains the `undefined` value has a `text record` sub-location that contains the empty string and an `access` sub-location that contains the `undefined` value.

**static properties:** A text mode has the following hereditary properties:

- A `text length` which is the value delivered by `text length`.
- A `text record` mode which is `CHARS (<text length>) VARYING` in case of `TEXT` and which is `WCHARS (<text length>) VARYING` in case of `WTEXT`.
- It has an `access` mode which is `ACCESS [(<index mode>)] CHARS (<text length>) [DYNAMIC]` in case of `TEXT` and which is `WCHARS (<text length>) [DYNAMIC]` in case of `WTEXT` (`<index mode>` and `DYNAMIC` are part of the mode only if they are specified).
- Optional `upper bound` and `lower bound` which are the `upper bound` and `lower bound` of the `index mode`, if present.

**static conditions:** If the `index mode` is a `literal range` of the form:

```
<lower bound> : <upper bound>
```

then, the evaluation of the 1.`lower bound`, 2.`upper bound`, must not depend directly or indirectly on the value of the 1.`lower bound`, 2.`upper bound` of the text mode.

**examples:**

26.8  TEXT (80) DYNAMIC

### 3.12 Timing modes

#### 3.12.1 General

**syntax:**

```
<duration mode> ::= (1)
  <duration mode name> (1.1)
  | <absolute time mode> (1.2)
```

**semantics:** A timing mode provides a means for time supervision of processes as described in clause 9. Timing values are created by a set of built-in routines. The relational operators are defined on timing values.

#### 3.12.2 Duration modes

**syntax:**

```
<duration mode> ::= (1)
  <duration mode name> (1.1)
```

**predefined names:** The name `DURATION` is predefined as a `duration mode` name.

**semantics:** A duration mode defines values which represent periods of time. The set of values defined by the duration mode is implementation defined. An implementation may choose to represent duration values as pairs of precision and value. Duration values are ordered in the intuitive way.

#### 3.12.3 Absolute time modes

**syntax:**

```
<absolute time mode> ::= (1)
  <absolute time mode name> (1.1)
```

**predefined names:** The name `TIME` is predefined as an `absolute time mode` name.

**semantics:** An absolute time mode defines values which represent points in time. The set of values defined by the absolute time mode is implementation defined. Absolute time values are ordered in the intuitive way.
3.13 Composite modes

3.13.1 General

syntax:

\[
<\text{composite mode}> ::= (1) \\
| <\text{string mode}> (1.1) \\
| <\text{array mode}> (1.2) \\
| <\text{structure mode}> (1.3) \\
| <\text{moreta mode}> (1.4)
\]

semantics: A composite mode defines composite values, i.e. values consisting of sub-components which can be accessed or obtained (see 4.2.6-4.2.10 and 5.2.6-5.2.10).

3.13.2 String modes

syntax:

\[
<\text{string mode}> ::= (1) \\
| <\text{string type}> ( <\text{string length}> ) \ [ \text{VARYING} ] (1.1) \\
| <\text{parameterized string mode}> (1.2) \\
| <\text{string mode name}> (1.3)
\]

\[
<\text{parameterized string mode}> ::= (2) \\
| <\text{origin string mode name}> ( <\text{string length}> ) (2.1) \\
| <\text{parameterized string mode name}> (2.2)
\]

\[
<\text{origin string mode name}> ::= (3) \\
| <\text{string mode name}> (3.1)
\]

\[
<\text{string type}> ::= (4) \\
| \text{BOOLS} (4.1) \\
| \text{CHARS} (4.2) \\
| \text{WCHARS} (4.3)
\]

\[
<\text{string length}> ::= (5) \\
| <\text{integer literal expression}> (5.1)
\]

semantics: A fixed string mode defines bit or character string values of a length indicated or implied by the string mode. A varying string mode defines bit or character string values whose actual length ranges from 0 to the string length. The length is known only at runtime from the value of the attribute actual length. For a fixed string mode the actual length is always equal to the string length. Character strings are sequences of character values; bit strings are sequences of boolean values.

String values are either empty or have string elements which are numbered from 0 upward.

The string values of a given string mode are totally-ordered in accordance with the ordering of the component values and the following definition.

Two strings \( s \) and \( t \) are equal if and only if they are empty or have the same length \( l \) and \( s(i) = t(i) \) for all \( 0 \leq i < l \). A string \( s \) precedes \( t \) when either:

- there exists an index \( j \) such that \( s(j) < t(j) \) and \( s(0 : j - 1) = t(0 : j - 1) \), or
- \( \text{LENGTH}(s) < \text{LENGTH}(t) \) and \( s = t(0 \ UP \ \text{LENGTH}(s)) \).

The concatenation operator is defined on string values. The usual logical operators are defined on bit string values and operate between their corresponding elements (see 5.3).

The maximum length of string modes is implementation defined.

static properties: A string mode has the following hereditary properties:

- A string length which is the value delivered by string length.
- An upper bound and a lower bound which are the values delivered by string length – 1 and 0, respectively.
• An element mode which is either \( M \) or READ \( M \), where \( M \) is BOOL, CHAR or WCHAR depending on whether string type specifies BOOls, CHARS or WCHARs, or the element mode of the origin string mode name, respectively. The element mode will be READ \( M \) if and only if the string mode is a read-only mode; in such case it is an implicit read-only mode.

• It is a varying string mode if VARYING is specified or if the origin string mode name denotes a varying string mode; otherwise it is a fixed string mode.

A string mode is parameterized if and only if it is a parameterized string mode.

A parameterized string mode has an origin string mode which is the mode denoted by origin string mode name.

A varying string mode has the following non-hereditary property: it has a component mode, defined as follows:

• If the varying string mode is of the form:
  
  \(<\text{string type}> ( <\text{string length}> ) \text{VARYING}\)

  then it is \(<\text{string type}> ( <\text{string length}> )\).

• If the varying string mode is of the form:

  \(<\text{origin string mode name}> ( <\text{string length}> )\)

  then the component mode is \&\text{name} ( string length ), where \&\text{name} is a virtually introduced \\text{synmode} name synonymous with the component mode of the origin string mode name.

• If the varying string mode is a string mode name which is a synmode name, then its component mode is that of the defining mode of the synmode name; otherwise it is a newmode name and then its component mode is the virtually introduced component mode (see 3.2.3).

\text{static conditions}: The string length must deliver a non-negative value.

The value delivered by the string length directly contained in a parameterized string mode must be less than or equal to the string length of the origin string mode name. This condition applies only to the parameterized string modes that are not introduced virtually.

The evaluation of the string length must not depend directly or indirectly on the value of the string length of the string mode.

\text{examples}:

| 7.51  | CHARS (20) | (1.1) |
| 22.22 | CHARS (20) VARYING | (1.1) |

\text{3.13.3 Array modes}

\text{syntax}:

\text{<array mode> ::=}

\text{ARRAY ( <index mode> | , <index mode> | ) *}

\text{| <element mode> | <element layout> | ) *}

\text{\| <parameterized array mode>}

\text{\| <array mode name>}

\text{\| <origin array mode name> ( <upper index> )}

\text{\| <parameterized array mode name>}

\text{\text{derived syntax}: An array mode with more than one index mode (denoting a multi-dimensional array), is derived syntax for an array mode with an element mode that is an array mode. For example:}

\text{ARRAY (1:20,1:10) INT}
is derived from:

```
ARRAY (RANGE (1:20)) ARRAY (RANGE (1:10)) INT
```

Only if this derived syntax is used, is more than one element layout occurrence allowed. The number of element layout occurrences must be less than or equal to the number of index mode occurrences. In that case, the leftmost element layout is associated with the innermost element mode, etc.

**semantics:** An array mode defines composite values, which are lists of values defined by its element mode. The physical layout of an array location or value can be controlled by element layout specification (see 3.13.5). Two array values are equal if and only if they have the same number of elements and the corresponding element values are equal.

The maximum number of elements of array modes is implementation defined.

**static properties:** An array mode has the following hereditary properties:

- An index mode which is the index mode if it is not a parameterized array mode, otherwise the index mode is the discrete range mode constructed as:

  `&name (lower bound : upper bound)`

  where `&name` is a virtual synmode name synonymous with the index mode of origin array mode name, lower bound is the lower bound of the index mode of the origin array mode name and upper bound is the upper index.

- An upper bound and a lower bound which are the upper bound and the lower bound of its index mode, respectively.

- An element mode which is either M or READ M, where M is the element mode, or the element mode of the origin array mode name, respectively. The element mode will be READ M if and only if M is not a read-only mode and the array mode is a read-only mode. The element mode is an implicit read-only mode if it is READ M.

- An element layout which, if it is a parameterized array mode, is the element layout of its origin array mode name; otherwise it is either the specified element layout, or the implementation default, which is either PACK or NOPACK.

- A number of elements which is the value delivered by:

  `NUM (upper bound) − NUM (lower bound) + 1`

  where upper bound and lower bound are respectively the upper bound and the lower bound of its index mode.

- It is a mapped mode if element layout is specified and is a step.

An array mode is parameterized if and only if it is a parameterized array mode.

A parameterized array mode has an origin array mode which is the mode denoted by origin array mode name.

**static conditions:** The class of upper index must be compatible with the index mode of the origin array mode name and the value delivered by it must lie in the range defined by that index mode.

If the array mode is a parameterized array mode, the evaluation of the upper index must not depend directly or indirectly on the value of the upper bound of the array mode. If the array mode is neither a parameterized array mode nor an array mode name, and if the index mode is a literal range of the form:

```
<lower bound> : <upper bound>
```

then, the evaluation of the 1. lower bound, 2. upper bound, must not depend directly or indirectly on the value of the 1. lower bound, 2. upper bound of the array mode.

**examples:**

```
5.27  ARRAY (1:16) STRUCT (c4, c2, c1 BOOL) (1.1)
11.12  ARRAY (line) ARRAY (column) square (1.1)
11.17  board (1.3)
```
3.13.4 Structure modes

syntax:

\[
<\text{structure mode}> :::=
\]

\[
\text{STRUCT} \left( <\text{field}> \{, <\text{field}>\}^* \right)
\]

\[
| <\text{parameterized structure mode}>
\]

\[
| <\text{structure mode name}>
\]

\[
<\text{field}> :::=
\]

\[
<\text{fixed field}>
\]

\[
| <\text{alternative field}>
\]

\[
<\text{fixed field}> :::=
\]

\[
<\text{field name defining occurrence list} > <\text{mode} > [ <\text{field layout} > ]
\]

\[
<\text{alternative field}> :::=
\]

\[
\text{CASE} \left[ <\text{tag list} > \right] \text{OF}
\]

\[
<\text{variant alternative} > \{, <\text{variant alternative} > \}^*
\]

\[
[ \text{ELSE} \left[ <\text{variant field} > \{, <\text{variant field} > \}^* \right] ] \text{ESAC}
\]

\[
<\text{variant alternative} > :::=
\]

\[
[ <\text{case label specification} > ] \left[ <\text{variant field} > \{, <\text{variant field} > \}^* \right] \]

\[
<\text{tag list}> :::=
\]

\[
<\text{tag field name} > \{, <\text{tag field name} > \}^*
\]

\[
<\text{variant field}> :::=
\]

\[
<\text{field name defining occurrence list} > <\text{mode} > [ <\text{field layout} > ]
\]

\[
<\text{parameterized structure mode} > :::=
\]

\[
<\text{origin variant structure mode name} > ( <\text{literal expression list} > )
\]

\[
| <\text{parameterized structure mode name}>
\]

\[
<\text{origin variant structure mode name} > :::=
\]

\[
<\text{variant structure mode name}>
\]

\[
<\text{literal expression list} > :::=
\]

\[
<\text{discrete literal expression} > \{, <\text{discrete literal expression} > \}^*
\]

\[
\text{derived syntax}: \text{A fixed field occurrence or variant field occurrence, where field name defining occurrence list consists of more than one field name defining occurrence, is derived syntax for several fixed field occurrences or variant field occurrences with one field name defining occurrence respectively, each with the specified mode and optional field layout. In the case of field layout, this field layout must not be pos. For example:}
\]

\[
\text{STRUCT } (I, J \text{ BOOL PACK})
\]

is derived from:

\[
\text{STRUCT } (I \text{ BOOL PACK, J BOOL PACK})
\]

semantics: Structure modes define composite values consisting of a list of values, selectable by a component name. Each value is defined by a mode that is attached to the component name. Structure values may reside in (composite) structure locations, where the component name serves as an access to the sub-location. The components of a structure value or location are called fields and their names field names.

There are fixed structures, variant structures and parameterized structures.

Fixed structures consist only of fixed fields, i.e. fields that are always present and that can be accessed without any dynamic check.

Variant structures have variant fields, i.e. fields that are not always present. For tagged variant structures, the presence of these fields is known only at run time from the value(s) of certain associated fixed field(s) called tag fields. Tag-less variant structures do not have tag fields. Because the composition of a variant structure may change during run time, the size of a variant structure location is based upon the largest choice (worst case) of variant alternatives.
In an alternative field the variant alternative chosen is that for which values give in the case label specification match; if no value match, the variant alternative following ELSE (which will be present) is chosen.

A parameterized structure is determined from a variant structure mode for which the choice of variant alternatives is statically specified by means of literal expressions. The composition is fixed from the point of the creation of the parameterized structure and may not change during run time. The tag fields, if present, are read-only and automatically initialized with the specified values. For a parameterized structure location, a precise amount of storage can be allocated at the point of declaration or generation. Note that dynamic parameterized structure modes also exist; their semantics are defined in 3.14.4.

The layout of a structure location or value can be controlled by means of a field layout specification (see 3.13.5).

Two structure values are equal if and only if the corresponding component values are equal. However, if the structure values are tag-less variant structure values, the result of comparison is implementation defined.

For a mode with the tagged parameterized property the undefined value denotes a value in which tag field sub-values are equal to the corresponding parameter values and all the other ones are equal to the undefined value.

static properties:

**general**: A structure mode has the following hereditary properties:

- It is a fixed structure mode if it is a structure mode that does not directly contain an alternative field occurrence.
- It is a variant structure mode if it is a structure mode and contains at least one alternative field occurrence.
- It is a parameterized structure mode if it is a parameterized structure mode.
- It has a set of field names. This set is defined below for the different cases. A name is said to be a field name if and only if it is defined in a field name defining occurrence list in fixed fields or variant fields in a structure mode.

Each fixed field, variant field and therefore each field name of a structure mode has a field mode attached that is either M or READ M, where M is the mode in the fixed field or variant field. The field mode is READ M if M is not a read-only mode and either the structure mode is a read-only mode, or the field is a tag field of a parameterized structure mode. The field mode is an implicit read-only mode if it is READ M.

A fixed field, variant field and therefore a field name of a given structure mode has a field layout attached to it that is the field layout in the fixed field or variant field, if present; otherwise it is the default field layout, which is either PACK or NOPACK.

- It is a mapped mode if its field names have a field layout that is pos.

**fixed structures**: A fixed structure mode has the following hereditary property:

- A set of field names which is the set of names defined by any field name defining occurrence list in fixed fields. These field names are fixed field names.

**variant structures**: A variant structure mode has the following hereditary properties:

- A set of field names which is the union of the set of names defined by any field name defining occurrence list in fixed fields and the set of names defined by any field name defining occurrence list in alternative fields. Field names defined by a field name defining occurrence list in fixed fields are the fixed field names of the variant structure mode; its other field names are the variant field names.
- A field name of a variant structure mode is a tag field name if and only if it occurs in any tag list of an alternative field. Alternative fields in which no tag lists are specified are tag-less alternative fields.
- A variant structure mode is a tag-less variant structure mode if all its alternative field occurrences are tag-less. Otherwise it is a tagged variant structure mode.
- A variant structure mode is a parameterizable variant structure mode if it is either a tagged variant structure mode or a tag-less variant structure mode where for each of the alternative field occurrences a case label specification is given for all the variant alternative occurrences in it.
A parameterizable variant structure mode has a list of classes attached, determined as follows:

- if it is a tagged variant structure mode, the list of $M_i$–value classes, where $M_i$ are the modes of the tag field names in the order that they are defined in fixed fields;
- if it is a tag-less variant structure mode, the list is built up from the individual resulting lists of classes of each alternative field by concatenating them in the order as the alternative fields occur. The resulting list of classes of an alternative field occurrence is the resulting list of classes of the list of case label specification occurrences in it (see 12.3).

### Parameterized Structures

A parameterized structure mode has the following hereditary properties:

- An origin variant structure mode which is the mode denoted by origin variant structure mode name.
- A set of field names which is the union of the set of fixed field names of its origin variant structure mode and the set of those variant field names of its origin variant structure mode that are defined in variant alternative occurrences that are selected by the list of values defined by literal expression list.
- The set of tag field names of a parameterized structure mode is the set of tag field names of its origin variant structure mode.
- A list of values attached, defined by literal expression list.
- It is a tagged parameterized structure mode if its origin variant structure mode is a tagged variant structure mode; otherwise the parameterized structure mode is tag-less.

For dynamic parameterized structure modes see 3.14.4.

### Static Conditions

**General:** All field names of a structure mode must be different.

If any field has a field layout which is pos, all the fields must have a field layout which must be pos.

**Variant Structures:** A tag field name must be a fixed field name and must be textually defined before all the alternative field occurrences in whose tag list it is mentioned. (As a consequence, a tag field precedes all the variant fields that depend upon it.) The mode of a tag field name must be a discrete mode.

The mode of variant field may have neither the non-value property nor the tagged parameterized property.

In a variant structure mode the alternative field occurrences must be either all tagged or all tag-less. For tagged alternative fields, case label specification must be specified in each variant alternative. For tag-less alternative fields, case label specification may be omitted in all variant alternative occurrences together, or must be specified for each variant alternative occurrence.

If, for a tag-less variant structure mode, any of its alternative fields has case label specification given, all its alternative fields must have case label specification.

For alternative fields, the case selection conditions must be fulfilled (see 12.3), and the same completeness, consistency and compatibility requirements must hold as for the case action (see 6.4). Each of the tag field names of tag list (if present) serves as a case selector with the M-value class, where M is the mode of the tag field name. In the case of tag-less alternative fields, the checks involving the case selector are ignored.

For a parameterizable variant structure mode none of the classes of its attached list of classes may be the all class. (This condition is automatically fulfilled by a tagged variant structure mode.)

### Parameterized Structures

The origin variant structure mode name must be parameterizable.

There must be as many literal expressions in the literal expression list as there are classes in the list of classes of the origin variant structure mode name. The class of each literal expression must be compatible with the corresponding (by position) class of the list of classes. If the latter class is an M-value class, the value delivered by the literal expression must be one of the values defined by M.
examples:

3.3  STRUCT (re, im INT)  

11.7 STRUCT (status SET (occupied, free),
         CASE status OF
         (occupied): p piece,
         (free):
         ESAC)  

2.6  fraction  

11.7  status SET (occupied, free)  

11.8  status  

11.9  p piece  

3.13.5  Layout description for array modes and structure modes

syntax:

\[
\begin{align*}
\langle\text{element layout}\rangle & : = \\
& \text{PACK} | \text{NOPACK} | <\text{step}> \\
\langle\text{field layout}\rangle & : = \\
& \text{PACK} | \text{NOPACK} | <\text{pos}> \\
\langle\text{pos}\rangle & : = \\
& \text{STEP} ( <\text{pos}> [ , <\text{step size}> ] ) \\
\langle\text{word}\rangle & : = \\
& <\text{integer literal expression}> \\
\langle\text{step size}\rangle & : = \\
& <\text{integer literal expression}> \\
\langle\text{start bit}\rangle & : = \\
& <\text{integer literal expression}> \\
\langle\text{end bit}\rangle & : = \\
& <\text{integer literal expression}> \\
\langle\text{length}\rangle & : = \\
& <\text{integer literal expression}>
\end{align*}
\]

semantics: It is possible to control the layout of an array or a structure by giving packing or mapping information in its mode. Packing information is either \text{PACK} or \text{NOPACK}, mapping information is either \text{step} in the case of array modes, or \text{pos} in the case of structure modes. The absence of \text{element layout} or \text{field layout} in an array or structure mode will always be interpreted as packing information, i.e. either as \text{PACK} or as \text{NOPACK}.

If \text{PACK} is specified for elements of an array or fields of a structure, it means that the use of memory space is optimized for the array elements or structure fields, whereas \text{NOPACK} implies that the access time for the array elements or the structure fields is optimized. \text{NOPACK} also implies \text{referable}.

The \text{PACK}, \text{NOPACK} information is applied only for one level, i.e. it is applied to the elements of the array or fields of the structure, not for possible components of the array element or structure field. The layout information is always attached to the nearest mode to which it may apply and which does not already have layout attached. For example, if the default packing is \text{NOPACK}:

\text{STRUCT (f ARRAY (0:1) m PACK)}
is equivalent to:

```
STRUCT (f ARRAY (0:1) m PACK NOPACK)
```

It is also possible to control the precise layout of an array or a structure by specifying positioning information for its components in the mode. This positioning information is given in the following ways:

- For array modes, the positioning information is given for all elements together, in the form of a `step` following the array mode.
- For structure modes, the positioning information is given for each field individually, in the form of a `pos`, following the mode of the field.

Mapping information with `pos` is given in terms of word and bit-offsets. A `pos` of the form:

```
POS (<word>, <start bit>, <length>)
```

defines a bit-offset of

```
NUM (word) * WIDTH + NUM (start bit)
```

and a length of `NUM (length)` bits, where `WIDTH` is the (implementation defined) number of bits in a word, and `word` is an integer literal expression.

When `pos` is specified in field layout it defines that the corresponding field starts at the given bit-offset from the start of each location of the structure mode, and occupies the given length.

A `step` of the form:

```
STEP (<pos>, <step size>)
```

defines a series of bit-offsets `b_i` for `i` taking values 0 to `n–1` where `n` is the number of elements in the array, and

```
b_i = i * NUM (step size)
```

The `j`-th element of the array starts at a bit-offset of `p + b_j` from the start of each location of the array mode, where `p` is the bit-offset specified in `pos`. Each element occupies the length given in `pos`.

### Defaults

The notation:

```
POS (<word>, <start bit> : <end bit>)
```

is semantically equivalent to:

```
POS (<word>, <start bit>, NUM (<end bit>) – NUM (<start bit>) + 1)
```

The notation:

```
POS (<word>, <start bit>)
```

is semantically equivalent to:

```
POS (<word>, <start bit>, BSIZE)
```

where `BSIZE` is the minimum number of bits which is needed to be occupied by the component for which the `pos` is specified.

The notation:

```
POS (<word>)
```

is semantically equivalent to:

```
POS (<word>, 0, BSIZE)
```

The notation:

```
STEP (<pos>)
```

is semantically equivalent to

```
STEP (<pos>, SSIZE)
```

where `SSIZE` is the `<length>` specified in `pos` or derivable from `pos` by the above rules.
**static properties:** For any location of an array mode the element layout of the mode determines the referability of its sub-locations (including sub-arrays, array slices) as follows:

- either all sub-locations are **referable**, or none of them are;
- if the element layout is **NOPACK** all sub-locations are **referable**.

For any location of a structure mode, the referability of the structure field selected by a **field** name is determined by the field layout of the **field** name as follows:

- the **field** name is **referable** if the field layout is **NOPACK**.

**static conditions:** If the **element** mode of a given array mode or the **field** mode of a **field** name of a given structure mode, is itself an array or structure mode, then it must be a **mapped** mode if the given array or structure mode is **mapped**.

\[
\text{NUM (word), NUM (start bit), NUM (end bit), NUM (length) and NUM (step size) } \geq 0;
\]
\[
\text{NUM (start bit) and NUM (end bit)} \leq \text{WIDTH}; \text{NUM (start bit) } \leq \text{NUM (end bit).}
\]

Each implementation defines for each mode a minimum number of bits its values need to occupy; call this the minimum bit occupancy. For discrete modes it is any number of bits not less than log to the base two of the **number of values** of the mode. For array modes it is the offset of the element of the highest index plus its occupied bits. For structure modes it is the offset of the highest bit occupied.

For each **pos** the **length** specified must not be less than the minimum bit occupancy of the mode of the associated field or array components.

For each **mapped** array mode the **step size** must not be less than the **length** given or implied in the **pos**.

**Consistency and feasibility**

**Consistency:** No component of a structure may be specified such that it occupies any bits occupied by another component of the same object except in the case of two **variant field** names defined in the same **alternative field** occurrence; however, in the latter case the **variant field** names may not both be defined in the same **variant alternative** nor both following **ELSE**.

**Feasibility:** There are no language defined feasibility requirements, except for the one that can be deduced from the rule that the referability of a sub-location of any (**referable** or non-**referable**) location is determined only by the (element or field) layout, which is a property of the mode of the location. This places some restrictions on the mapping of components that themselves have **referable** components.

**examples:**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>17.5</td>
<td>PACK</td>
<td>(1.1)</td>
</tr>
<tr>
<td>19.14</td>
<td>POS (1,0:15)</td>
<td>(4.2)</td>
</tr>
</tbody>
</table>

**3.14 Dynamic modes**

**3.14.1 General**

A dynamic mode is a mode of which some properties are known only at run time. Dynamic modes are always parameterized modes with one or more run-time parameters. For description purposes, virtual denotations are introduced in this Recommendation | International Standard. These virtual denotations are preceded by the ampersand symbol (&) to distinguish them from actual notations which appear in a CHILL program text.

**3.14.2 Dynamic string modes**

**virtual denotation:** &<origin string mode name> ( <integer expression> )

**semantics:** A dynamic string mode is a parameterized string mode with non constant length.

**static properties:** Dynamic string modes have the same properties as string modes, except for the properties described below.
dynamic properties:

- A dynamic string mode has a dynamic string length which is the value delivered by integer expression.
- A dynamic string mode has an upper bound and a lower bound which are the values delivered by string length –1 and 0, respectively.

3.14.3 Dynamic array modes

virtual denotation: &<origin array mode name> ( <discrete expression> )

semantics: A dynamic array mode is a parameterized array mode with non constant upper bound.

static properties: Dynamic array modes have the same properties as array modes, except for the properties described below.

dynamic properties:

- A dynamic array mode has a dynamic upper bound which is the value delivered by discrete expression, and a dynamic number of elements which is the value delivered by:

  \[ \text{NUM (discrete expression)} - \text{NUM (lower bound)} + 1 \]

  where lower bound is the lower bound of the origin array mode name.

3.14.4 Dynamic parameterized structure modes

virtual denotation: &<origin variant structure mode name> ( <expression list> )

semantics: A dynamic parameterized structure mode is a parameterized structure mode with non constant parameters.

static properties: The static properties of a dynamic parameterized structure mode are those of a static parameterized structure mode except for the following:

- The set of field names of a dynamic parameterized structure mode is the set of field names of its origin variant structure mode.

dynamic properties:

- A dynamic parameterized structure mode has a list of values attached that is the list of values delivered by the expressions in the expression list.

3.15 Moreta Modes

3.15.1 General

syntax:

- <moreta mode> ::= (1)
  - <module mode> (1.1)
  - | <region mode> (1.2)
  - | <task mode> (1.3)
  - | <generic moreta mode instantiation> (1.4)
  - | <interface mode> (1.5)
  - | <moreta mode name> [ ( <actual parameter list> ) ] (1.6)

semantics:

module mode – A location of module mode has the same properties as a module without an action statement list.

region mode – A location of region mode has the same properties as a region.

task mode – A location of task mode has essentially the same structure as a module mode location without process definitions. The direct access to the components of a location, whose mode is a task mode, is mutually exclusive. A location, whose mode is a task mode, may be executed concurrently with other threads (see 11.1).

generic moreta mode instantiation – A generic moreta mode instantiation is obtained statically by an instantiation of a generic moreta mode template (see 10.11).

interface mode – An interface mode consists of specifications and signatures only.
static conditions:

Moreta modes are not parameterizable.

Moreta modes and generic moreta mode templates cannot be nested. (1.1) – (1.5) are only allowed in symmode and newmode definitions, i.e. anonymous moreta modes are not allowed.

3.15.2 Module Modes

syntax:

<module mode> ::= (1)
   <module mode specification> (1.1)
   | <module mode body> (1.2)

<module mode specification> ::= (2)
   MODULE SPEC [[ ASSIGNABLE | FINAL | ABSTRACT ] |
   [ NOT_ASSIGNABLE | ABSTRACT | FINAL ]]
   <module inheritance clause>
   {<module specification component> | <invariant part>}
   END [<simple name string>] (2.1)

<module mode body> ::= (3)
   MODULE BODY [[ ASSIGNABLE | FINAL | ABSTRACT ] |
   [ NOT_ASSIGNABLE | ABSTRACT | FINAL ]]
   <module inheritance clause>
   {<module body component> | <invariant part>}
   END [ <handler> ] [<simple name string>] (3.1)

<module inheritance clause> ::= (4)
   [<module inheritance>] [<implementation clause>] (4.1)

<module inheritance> ::= (5)
   BASED_ON <module mode name> (5.1)

<implementation clause> ::= (6)
   IMPLEMENTS <interface mode name> {, <interface mode name>} (6.1)

<module specification component> ::= (7)
   | <common module component>
   | <declaration statement> (7.1)
   | <simple guarded procedure signature statement> (7.2)
   | <inline guarded procedure definition statement> (7.3)
   | <process specification statement> (7.4)
   | <signal definition statement> (7.5)
   | <grant statement> (7.6)

<module body component> ::= (8)
   | <common module component> (8.1)
   | <simple guarded procedure definition statement> (8.2)
   | <process definition statement> (8.3)

<common module component> ::= (9)
   | <synonym definition statement> (9.1)
   | <synmode definition statement> (9.2)
   | <newmode definition statement> (9.3)
   | <seize statement> (9.4)

<invariant part> ::= (10)
   INVARIANT <boolean expression> (10.1)

semantics: A module mode defines composite values consisting of a list of components selectable by component names.

Module values may reside in (composite) module locations.

A module mode is defined by giving two separate parts: a module mode specification and a module mode body.
The **specification** part defines the interface of the values of a **module mode**.

The **body** part defines the behaviour of the values of a **module mode**.

The boolean expression of the invariant part must be true before and after any call of a **public** component procedure or a **public** component process.

**static properties:** If the attribute **ASSIGNABLE** is specified, the mode is an **assignable** module mode. An **assignable** module mode can be used in the same way as a mode for which **READ** is not specified (see 3.3).

If the attribute **NOT_ASSIGNABLE** is specified, the mode has the **not_assignable** property, indicating that the location of that mode may not be accessed to store the value and may not be accessed to copy its value.

If neither **ASSIGNABLE** nor **NOT_ASSIGNABLE** is specified the mode is **not_assignable** by default.

If the attribute **ABSTRACT** is specified, the mode is an **abstract** mode.

If a **module inheritance** is given, the mode MD being defined is immediately derived from the mode MB given in the **module inheritance**, and MB is an immediate base mode of MD.

If an **implementation clause** IC is given, the mode MD being defined is immediately derived from the modes given in IC, and these modes are immediate base modes of MD.

The effect of the **module inheritance clause** is that the derived mode behaves as if it contained all components of its immediate base modes except for the constructor and destructor component procedures of these base modes. If any of these base modes is itself a derived mode, this inheritance of components is to be understood in a transitive manner. For visibility see 12.2.

A **module specification component** contained in a **module mode specification** MS or SEIZEd into MS, which is granted by MS, is called a **public** component of the mode of MS.

A **module specification component** contained in a **module mode specification** MS or SEIZEd into MS, which is not granted by MS, is called an **internal** component of the mode of MS.

A **module body component** C contained in a **module mode body** MB or SEIZEd into MB, is called a **private** component of the mode of MB if C is neither a **public** nor an **internal** component of the mode of MB.

An **abstract** module mode has the property **not_assignable**.

**static conditions:** A module mode cannot be used as the mode in a **synonym definition**.

For each **module mode specification**, there must be one **module mode body** with the same name string in the **defining occurrence**.

If specified, the **simple name string** after **END** must be equal to the name string of the defining occurrence of this mode definition. This holds for **module mode specification** and for **module mode body**.

If one of the attributes **ASSIGNABLE, NOT_ASSIGNABLE, ABSTRACT** or **FINAL** is specified in a **module mode specification**, it must also be specified in the corresponding **module mode body**.

If a **module mode specification** contains a **module inheritance clause**, the corresponding **module mode body** must contain the same **module inheritance clause**.

If the attribute **INCOMPLETE** (see 10.4) is specified in a **simple guarded procedure signature** then this procedure has the property **incomplete**.

If the attribute **INCOMPLETE** (see 10.4) is specified in a **simple guarded procedure signature statement** this procedure must be **public**.

For each **simple, complete guarded procedure signature statement** S of a **module mode specification**, the corresponding **module mode body** must contain a corresponding **simple guarded procedure definition statement** D, where the guarded procedure signature of S matches the guarded procedure definition of D (see 12.1.3).

If P is a **simple, incomplete guarded procedure signature of a module mode specification**, the corresponding **module mode body** must not contain a **simple guarded procedure definition matching** P.

For each **process specification** of a **module mode specification**, the corresponding **module mode body** must contain a corresponding **process definition** (see 12.1.3).
If the attribute \texttt{REIMPLEMENT} (see 10.4) is specified in a \textit{simple guarded procedure signature statement} this procedure must be \texttt{public}.

If the attribute \texttt{REIMPLEMENT} (see 10.4) is specified in a \textit{simple guarded procedure signature} \texttt{PD} contained in a \textit{module mode specification} \texttt{M} then the immediate base mode \texttt{MB} of \texttt{M} must contain or have inherited a \texttt{public simple guarded procedure signature} \texttt{PB}, where \texttt{PB} matches \texttt{PD} and \texttt{PB} is neither a constructor nor a destructor and \texttt{PB} is not \texttt{SEIZE}ed.

A \textit{module mode} is an \texttt{abstract} module mode if it contains at least one \texttt{incomplete component procedure} (see 10.4). In this case the attribute \texttt{ABSTRACT} must be specified.

An \texttt{abstract} module mode name can only be used as the module mode name in a \textit{module inheritance} or as a \texttt{referenced mode}.

If a \textit{module mode} \texttt{M} has at least one (sub-)component with \texttt{non-value property}, then \texttt{M} also has the \texttt{non-value property} and the attribute \texttt{ASSIGNABLE} must not be specified (see 12.1.1.5).

If a \textit{module mode} \texttt{M} contains the attribute \texttt{FINAL} \texttt{M} is called a \texttt{final} module mode. A \texttt{final} module mode cannot be used as a base mode in a moreta \textit{inheritance}.

A \texttt{final} module mode must not contain an \texttt{incomplete} component procedure.

### 3.15.3 Region Modes

**syntax:**

\[
\texttt{<region mode>} ::= (1) \\
\texttt{<region mode specification>} | (1.1) \texttt{<region mode body>} (1.2)
\]

\[
\texttt{<region mode specification>} ::= (2) \\
\texttt{REGION SPEC \{ABSTRACT | FINAL\} [<region inheritance>] \{<region specification component>\}* [<invariant part>] \texttt{END} [<simple name string>]}
\]

\[
\texttt{<region mode body>} ::= (3) \\
\texttt{REGION BODY \{ABSTRACT | FINAL\} [<region inheritance clause>] \{<region body component>\}* [<invariant part>] \texttt{END} [<handler>] [<simple name string>]}
\]

\[
\texttt{<region inheritance clause>} ::= (4) \\
\texttt{<region inheritance>] [<implementation clause>]}
\]

\[
\texttt{<region inheritance>} ::= (5) \\
\texttt{BASED\_ON \{<module mode name> | <region mode name>\}}
\]

\[
\texttt{<region specification component>} ::= (6) \\
\texttt{<common module component>} | (6.1) \texttt{<declaration statement>} (6.2) \\
| (6.3) \texttt{<simple guarded procedure signature statement>} \\
| (6.4) \texttt{<signal definition statement>} \\
| (6.5) \texttt{<grant statement>}
\]

\[
\texttt{<region body component>} ::= (7) \\
\texttt{<common module component>} | (7.1) \texttt{<simple guarded procedure definition statement>}
\]

**semantics:** A \textit{region mode} defines composite values consisting of a list of components selectable by component names.

Region values may reside in (composite) region locations.

A \textit{region mode} is defined by giving two separate parts: a \textit{region mode specification} and a \textit{region mode body}.

The \texttt{specification} part defines the interface of the values of the \textit{region mode}.

The \texttt{body} part defines the behaviour of the values of the \textit{region mode}.

The \texttt{boolean expression} of the \texttt{invariant part} must be true before and after any call of a \texttt{public} component procedure.
static properties: A region mode has always the not assignable property.

If the attribute ABSTRACT is specified, the mode is an abstract mode.

If a region inheritance is given, the mode MD being defined is immediately derived from the mode MB given in the region inheritance, and MB is an immediate base mode of MD.

If an implementation clause IC is given, the mode MD being defined is immediately derived from the modes given in IC, and these modes are immediate base modes of MD.

The effect of the region inheritance clause is that the derived mode behaves as if it contained all components of its immediate base modes except for the constructor and destructor component procedures of these base modes. If any of these base modes is itself a derived mode, this inheritance of components is to be understood in a transitive manner. For visibility see 12.2.

A region specification component contained in a region mode specification MS or SEIZED into MS, which is granted by MS, is called a public component of the mode of MS.

A region specification component contained in a region mode specification MS or SEIZED into MS, which is not granted by MS, is called an internal component of the mode of MS.

A region body component C contained in a region mode body MB or SEIZED into MB, is called a private component of the mode of MB if C is neither a public nor an internal component of the mode of MB.

static conditions: A region mode cannot be used as the mode in a synonym definition.

For each region mode specification, there must be one region mode body with the same name string in the defining occurrence.

If specified, the simple name string after END must be equal to the name string of the defining occurrence of this mode definition. This holds for region mode specification and for region mode body.

If the attribute ABSTRACT or FINAL is specified in a region mode specification, it must also be specified in the corresponding region mode body.

If a region mode specification contains a region inheritance clause, the corresponding region mode body must contain the same region inheritance clause.

If the attribute INCOMPLETE (see 10.4) is specified in a simple guarded procedure signature then this procedure has the property incomplete.

If the attribute INCOMPLETE (see 10.4) is specified in a simple guarded procedure signature statement this procedure must be public.

For each simple, complete guarded procedure signature statement S of a region mode specification, the corresponding region mode body must contain a corresponding simple guarded procedure definition statement D (see 12.1.3), where the guarded procedure signature of S matches the guarded procedure definition of D.

If P is a simple, incomplete guarded procedure signature of a region mode specification, the corresponding region mode body must not contain a simple guarded procedure definition matching P.

If the attribute REIMPLEMENT (see 10.4) is specified in a simple guarded procedure signature statement this procedure must be public.

If the attribute REIMPLEMENT (see 10.4) is specified in a simple guarded procedure signature PD contained in a region mode specification M then the immediate base mode MB of M must contain or have inherited a public simple guarded procedure signature PB, where PB matches PD and PB is neither a constructor nor a destructor and PB is not SEIZED.

A region mode is an abstract region mode if it contains at least one incomplete component procedure (see 10.4). In this case the attribute ABSTRACT must be specified.

An abstract region mode name can only be used as the region mode name in a region inheritance or as a referenced mode.

A region mode specification must not grant any location.

If the base mode of a region mode is a module mode M then M must have the not assignable property, must not grant any location and must not contain any inline guarded component procedure or any component process.
If a region mode \( M \) contains the attribute \text{FINAL} \( M \) is called a final region mode. A final region mode cannot be used as a base mode in a region inheritance.

A final region mode must not contain an incomplete component procedure.

### 3.15.4 Task Modes

**syntax:**

\[
\begin{align*}
\text{<task mode> ::= } & \quad (1) \\
\text{<task mode specification> | <task mode body>} & \quad (1.1)
\end{align*}
\]

\[
\begin{align*}
\text{<task mode specification> ::= } & \quad (2) \\
\text{TASK SPEC [ABSTRACT | FINAL] [<task inheritance clause>] *[<task specification component]> \ END [<simple name string>]}
\end{align*}
\]

\[
\begin{align*}
\text{<task mode body> ::= } & \quad (3) \\
\text{TASK BODY [ABSTRACT | FINAL] [<task inheritance clause>] *[<task body component>] \ END [<simple name string>]}
\end{align*}
\]

\[
\begin{align*}
\text{<task inheritance clause> ::= } & \quad (4) \\
\text{[<task inheritance>] [<implementation clause>]}
\end{align*}
\]

\[
\begin{align*}
\text{<task inheritance> ::= } & \quad (5) \\
\text{BASED ON [<module mode name> | <task mode name>]} \quad (5.1)
\end{align*}
\]

\[
\begin{align*}
\text{<task specification component> ::= } & \quad (6) \\
\text{<region specification component>}
\end{align*}
\]

\[
\begin{align*}
\text{<task body component> ::= } & \quad (7) \\
\text{<region body component>}
\end{align*}
\]

**semantics:** A task mode defines composite values consisting of a list of components selectable by component names. Task values may reside in (composite) task locations.

A task mode is defined by giving two separate parts: a task mode specification and a task mode body.

The specification part defines the interface of the values of the task mode.

The body part defines the behaviour of the values of the task mode.

The boolean expression of the invariant part must be true before and after any call of a public component procedure.

**static properties:** A task mode has the notAssignable property.

If the attribute ABSTRACT is specified, the mode is an abstract mode.

If a task inheritance is given, the mode MD being defined is immediately derived from the mode MB given in the task inheritance, and MB is an immediate base mode of MD.

If an implementation clause IC is given, the mode MD being defined is immediately derived from the modes given in IC, and these modes are immediate base modes of MD.

The effect of the task inheritance clause is that the derived mode behaves as if it contained all components of its immediate base modes except for the constructor and destructor component procedures of these base modes. If any of these base modes is itself a derived mode, this inheritance of components is to be understood in a transitive manner. For visibility see 12.2.

A task specification component contained in a task mode specification \( M_S \) or SEIZEd into \( M_S \), which is granted by \( M_S \), is called a public component of the mode of \( M_S \).

A task specification component contained in a task mode specification \( M_S \) or SEIZEd into \( M_S \), which is not granted by \( M_S \), is called an internal component of the mode of \( M_S \).
A task body component \( C \) contained in a task mode body \( M_B \) or SEIZed into \( M_B \), is called a **private** component of the mode of \( M_B \) if \( C \) is neither a **public** nor an **internal** component of the mode of \( M_B \).

**Static conditions**: A task mode cannot be used as the mode in a synonym definition.

For each task mode specification, there must be one task mode body with the same name string in the defining occurrence.

If specified, the simple name string after **END** must be equal to the name string of the defining occurrence of this mode definition. This holds for task mode specification and for task mode body.

If the attribute **ABSTRACT** or **FINAL** is specified in a task mode specification, it must also be specified in the corresponding task mode body.

If a task mode specification contains a task inheritance clause, the corresponding task mode body must contain the same task inheritance clause.

All public component procedures of a task mode must only have IN parameters and must not have a result spec.

If the attribute **INCOMPLETE** (see 10.4) is specified in a simple guarded procedure signature then this procedure has the property **incomplete**.

If the attribute **INCOMPLETE** (see 10.4) is specified in a simple guarded procedure signature statement this procedure must be **public**.

For each simple, complete guarded procedure signature statement \( S \) of a task mode specification, the corresponding task mode body must contain a corresponding simple guarded procedure definition statement \( D \) (see 12.1.3), where the guarded procedure signature of \( S \) matches the guarded procedure definition of \( D \).

If \( P \) is a simple, incomplete guarded procedure signature of a task mode specification, the corresponding task mode body must not contain a simple guarded procedure definition matching \( P \).

If the attribute **REIMPLEMENT** (see 10.4) is specified in a simple guarded procedure signature statement this procedure must be **public**.

If the attribute **REIMPLEMENT** (see 10.4) is specified in a simple guarded procedure signature \( PD \) contained in a task mode specification \( M \) then the immediate base mode of \( M \) must contain or have inherited a public simple guarded procedure signature \( PB \), where \( PB \) matches \( PD \) and \( PB \) is neither a constructor nor a destructor and \( PB \) is not SEIZed.

A task mode is an abstract task mode if it contains at least one incomplete component procedure (see 10.4). In this case the attribute **ABSTRACT** must be specified.

An abstract task mode name can only be used as the task mode name in a task inheritance or as a referenced mode.

A task mode specification must not grant any location.

If an immediate base mode of a task mode is a module mode \( M \) then \( M \) must have the **not assignable** property, must not grant any location, must not contain any inline guarded component procedure or any component process, and must contain only public procedures which fulfill the restrictions of public component procedures of task modes.

If a task mode \( M \) contains the attribute **FINAL**, \( M \) is called a final task mode. A final task mode cannot be used as a base mode in a task inheritance.

A final task mode must not contain an incomplete component procedure.

### 3.15.5 Interface Modes

**Syntax**:

\[
<\text{interface mode}> ::= \text{INTERFACE} [<\text{interface inheritance}>] \{<\text{interface component}>*\} \text{END} [<\text{simple name string}>] (1)
\]

\[
<\text{interface inheritance}> ::= \text{BASED_ON} <\text{interface mode name}> \{, <\text{interface mode name}> \}*(2)
\]
semantics: An interface mode defines a moreta mode which can only be used as a base mode in the definition of other moreta modes and as the referenced mode of a bound reference mode.

static properties: If interface inheritance II is given, the mode MD being defined is immediately derived from the modes given in II, and these modes are immediate base modes of MD.

The effect of the interface inheritance is that the derived mode behaves as if it contained all components of its immediate base modes. If any of these base modes is itself a derived mode, this inheritance of components is to be understood in a transitive manner. For visibility see 12.2.

All interface components (including the SEIZEd ones) are implicitly GRANTed and therefore all are called public components.

An interface mode is an abstract mode.

static conditions: An interface mode cannot be used as the mode in a synonym definition.

If specified, the simple name string after END must be equal to the name string of the defining occurrence of this mode definition.

The attribute INCOMPLETE (see 10.4) must be specified in all simple guarded procedure signatures; therefore, all procedures have the property incomplete.

The attribute REIMPLEMENT (see 10.4) must not be specified in a simple guarded procedure signature statement of an interface component.

4 Locations and their accesses

4.1 Declarations

4.1.1 General

syntax:

```
<declaration statement> ::= DCL <declaration> { , <declaration> }* ; (1.1)
```

```
<declaration> ::= <location declaration> (2.1)
| <loc-identity declaration> (2.2)
```

semantics: A declaration statement declares one or more names to be an access to a location.

examples:

```
6.9  DCL j INT := julian_day_number,
    d, m, y INT; (1.1)

11.36 starting_square LOC := b(m.lin_1)(m.col_1) (2.2)
```

4.1.2 Location declarations

syntax:

```
<location declaration> ::= <defining occurrence list> <mode> [ STATIC ] [ <initialization> ] (1.1)
```

```
<initialization> ::= <reach-bound initialization> (2.1)
| <lifetime-bound initialization> (2.2)
| <moreta-bound initialization> (2.3)
```
semantics: A location declaration creates as many locations as there are defining occurrences specified in the defining occurrence list.

With reach-bound initialization, the value is evaluated each time the reach in which the declaration is placed is entered (see 10.2) and the delivered value is assigned to the location(s). Before the value is evaluated the location(s) contain the undefined value.

With lifetime-bound initialization, the value yielded by the constant value is assigned to the location(s) only once at the beginning of the lifetime of the location(s) (see 10.2 and 10.9).

If the mode is a moreta mode, first all initializations in the components are performed in textual order. If a (possibly empty) parameter list is specified, the corresponding constructor of the mode is applied to the newly created location. If the mode is a task mode, the task belonging to the newly created location is started.

Specifying no initialization is semantically equivalent to the specification of a lifetime-bound initialization with the undefined value (see 5.3.1).

The meaning of the undefined value as initialization for a location which has attached a mode with the tagged parameterized property or the non-value property is as follows:

- tagged parameterized property: the created tag field sub-location(s) are initialized with their corresponding parameter value.

- non-value property:
  - the created event and/or buffer (sub-)location(s) are initialized to "empty", i.e. no delayed processes are attached to the event or buffer nor are there messages in the buffer;
  - the created region and/or task (sub-)location(s) are initialized to "empty", i.e. no delayed threads are attached to them;
  - the created association (sub-)location(s) are initialized to "empty", i.e. they do not contain an association;
  - the created access (sub-)location(s) are initialized to "empty", i.e. they are not connected to an association;
  - the created text (sub-)location(s) have a text record sub-location which is initialized with an empty string and an access sub-location which is initialized with "empty", i.e. it is not connected to an association.

- The semantics of STATIC and handler can be found in 10.9 and clause 8, respectively.

If the lifetime of a moreta location L ends and the mode of the location contains a destructor, then this destructor is applied to L (see 10.2).

static properties: A defining occurrence in a location declaration defines a location name. The mode attached to the location name is the mode specified in the location declaration. A location name is referable.

static conditions: The class of the value or constant value must be compatible with the mode and the delivered value should be one of the values defined by the mode, or the undefined value.

If the mode has the read-only property, initialization must be specified. If the mode has the non-value property, reach-bound initialization must not be specified.

If initialization is specified, the value must be regionally safe for the location (see 11.2.2).

dynamic conditions: In the case of reach-bound initialization, the assignment conditions of value with respect to the mode apply (see 6.2).
4.1.3 Loc-identity declarations

syntax:

<loc-identity declaration> ::= (1)
  <defining occurrence list> <mode> LOC [ DYNAMIC ]
  <assignment symbol> <location> [ <handler> ] (1.1)

semantics: A loc-identity declaration creates as many access names to the specified location as there are defining occurrences specified in the defining occurrence list. The mode of the location may be dynamic only if DYNAMIC is specified.

If the location is evaluated dynamically, this evaluation is done each time the reach in which the loc-identity declaration is placed is entered. In this case, a declared name denotes an undefined location prior to the first evaluation during the lifetime of the access denoted by the declared name (see 10.2 and 10.9).

static properties: A defining occurrence in a loc-identity declaration defines a loc-identity name. The mode attached to a loc-identity name is, if DYNAMIC is not specified, the mode specified in the loc-identity declaration; otherwise it is the dynamically parameterized version of it that has the same parameters as the mode of the location.

It is not allowed to create a location of a moreta mode with the DYNAMIC property.

A loc-identity name is referable if and only if the specified location is referable.

static conditions: If DYNAMIC is specified in the loc-identity declaration, the mode must be parameterizable. The specified mode must be dynamic read-compatible with the mode of the location if DYNAMIC is specified and read-compatible with the mode of the location otherwise.

The location must not be a string element or string slice in which the mode of the string location is a varying string mode.

dynamic conditions: The RANGEFAIL or TAGFAIL exception occurs if DYNAMIC is specified, and the above-mentioned dynamic read-compatible check fails.

examples:

11.36 starting square LOC := b(m.lin_1)(m.col_1) (1.1)
semantics: A location is an object that can contain values. Locations have to be accessed to store or obtain a value.

static properties: A location has the following properties:

- A mode, as defined in the appropriate sections. This mode is either static or dynamic.
- It is static or not (see 10.9).
- It is intra-regional or extra-regional (see 11.2.2).
- It is referable or not. The language definition requires certain locations to be referable and others to be not referable as defined in the appropriate sections. An implementation may extend referability to other locations except when explicitly disallowed.

4.2.2 Access names

syntax:

\[
<\text{access name}> ::= \begin{cases} 
<\text{location name}> & (1) \\
<\text{loc-identity name}> & (1.1) \\
<\text{location enumeration name}> & (1.2) \\
<\text{location do-with name}> & (1.3) \\
<\text{location do-with name}> & (1.4) 
\end{cases}
\]

semantics: An access name delivers a location. An access name is one of the following:

- a location name, i.e. a name explicitly declared in a location declaration or implicitly declared in a formal parameter without the LOC attribute;
- a loc-identity name, i.e. a name explicitly declared in a loc-identity declaration or implicitly declared in a formal parameter with the LOC attribute;
- a location enumeration name, i.e. a loop counter in a location enumeration;
- a location do-with name, i.e. a field name used as direct access in the do action with a with part.

If the location denoted by a location do-with name is a variant field of a tag-less variant structure location, the semantics are implementation defined.

static properties: The (possibly dynamic) mode attached to an access name is the mode of the location name, loc-identity name, location enumeration name or location do-with name, respectively.

An access name is referable if and only if it is a location name, a referable loc-identity name, a referable location enumeration name, or a referable location do-with name.

dynamic conditions: When accessing via a loc-identity name, it must not denote an undefined location.

When accessing via a loc-identity name a location which is a variant field, the variant field access conditions for the location must be satisfied (see 4.2.10). Accessing via a location do-with name causes a TAGFAIL exception if the denoted location is a variant field and the variant field access conditions for the location are not satisfied.

examples:

<table>
<thead>
<tr>
<th>Number</th>
<th>Access Name</th>
<th>Reference</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.12</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.39</td>
<td>starting</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.35</td>
<td>each</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.10</td>
<td>c1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.2.3 Dereferenced bound references

syntax:

\[
<\text{dereferenced bound reference}> ::= \begin{cases} 
<\text{bound reference primitive value}> \rightarrow [ <\text{mode name}> ] & (1) 
\end{cases}
\]

semantics: A dereferenced bound reference delivers the location that is referenced by the bound reference value.
static properties: The mode attached to a dereferenced bound reference is the mode name if specified, otherwise the referenced mode of the mode of the bound reference primitive value. A dereferenced bound reference is referable.

static conditions: The bound reference primitive value must be strong. If the optional mode name is specified, it must be read-compatible with the referenced mode of the mode of the bound reference primitive value.

dynamic conditions: The lifetime of the referenced location must not have ended.

The EMPTY exception occurs if the bound reference primitive value delivers the value NULL.

If the referenced location is a variant field, the variant field access conditions for the location must be satisfied (see 4.2.10).

examples:

10.54 \textit{p} \rightarrow 

4.2.4 Dereferenced free references

syntax:

\[
\texttt{<dereferenced free reference>} ::= \texttt{<free reference primitive value>} \rightarrow \texttt{<mode name>} 
\]

semantics: A dereferenced free reference delivers the location that is referenced by the free reference value.

static properties: The mode attached to a dereferenced free reference is the mode name. A dereferenced free reference is referable.

static conditions: The free reference primitive value must be strong.

dynamic conditions: The lifetime of the referenced location must not have ended.

The EMPTY exception occurs if the free reference primitive value delivers the value NULL.

The mode name must be read-compatible with the mode of the referenced location.

If the referenced location is a variant field, the variant field access conditions for the location must be satisfied (see 4.2.10).

4.2.5 Dereferenced rows

syntax:

\[
\texttt{<dereferenced row>} ::= \texttt{<row primitive value>} \rightarrow 
\]

semantics: A dereferenced row delivers the location that is referenced by the row value.

static properties: The dynamic mode attached to a dereferenced row is constructed as follows:

\[
\&\texttt{<origin mode name>} (, \texttt{<parameter>} | \texttt{<parameter>})* 
\]

where \&\texttt{origin mode name} is a virtual synmode name synonymous with the referenced origin mode of the mode of the row primitive value and where the parameters are, depending on the referenced origin mode:

- the dynamic string length, in the case of a string mode;
- the dynamic upper bound, in the case of an array mode;
- the list of values associated with the mode of the parameterized structure location, in the case of a variant structure mode.

A dereferenced row is referable.

static conditions: The row primitive value must be strong.

dynamic conditions: The lifetime of the referenced location must not have ended.

The EMPTY exception occurs if the row primitive value delivers NULL.
If the referenced location is a **variant** field, the variant field access conditions for the location must be satisfied (see 4.2.10).

**examples:**

8.11 *input* \(\rightarrow\)  

4.2.6 **String elements**

**syntax:**

\[
\begin{align*}
<string element> & ::= & (1) \\
<string location> & ( <start element> ) & (1.1) \\
<start element> & ::= & (2) \\
<integer expression> & (2.1) \\
\end{align*}
\]

**semantics:** A string element delivers a (sub-)location which is the element of the specified string location indicated by *start element*.

**static properties:** The mode attached to the *string element* is the *element* mode of the mode of the *string location*.

If the mode of the *string location* is a *varying* string mode, then the *string element* is not **referable**.

**dynamic conditions:** The **RANGEFAIL** exception occurs if the following relation does not hold:

\[
0 \leq \text{NUM (start element)} \leq L - 1
\]

Where \(L\) is the **actual length** of the *string location*.

**examples:**

18.16 *string* \(\rightarrow\) (i)  

4.2.7 **String slices**

**syntax:**

\[
\begin{align*}
<string slice> & ::= & (1) \\
<string location> & ( <left element> : <right element> ) & (1.1) \\
| & <string location> & ( <start element> \text{ UP <slice size>} ) & (1.2) \\
<left element> & ::= & (2) \\
<integer expression> & (2.1) \\
<right element> & ::= & (3) \\
<integer expression> & (3.1) \\
<slice size> & ::= & (4) \\
<integer expression> & (4.1) \\
\end{align*}
\]

**semantics:** A string slice delivers a (possibly dynamic) string location that is the part of the specified string location indicated by *left element* and *right element* or *start element* and *slice size*. The (possibly dynamic) length of the string slice is determined from the specified expressions.

A *string slice* in which the *right element* delivers a value which is less than that delivered by the *left element* or in which *slice size* delivers a non positive value denotes an empty string.

**static properties:** The (possibly dynamic) mode attached to a *string slice* is a **parameterized** string mode constructed as:

\[
\&\text{name (string size)}
\]

where \&name is a virtual **symmode** name **synonymous** with the (possibly dynamic) mode of the *string location* if it is a **fixed** string mode, otherwise with the **component** mode, and where *string size* is either:

\[
\text{NUM (right element)} - \text{NUM (left element)} + 1
\]

or

\[
\text{NUM (slice size)}.
\]
However, if an empty string is denoted, string size is 0. The mode attached to a string slice is static if string size is literal, i.e. left element and right element are literal or slice size is literal; otherwise the mode is dynamic.

If the mode of the string location is a varying string mode, then the string slice is not referable.

**static conditions:** The following relations must hold:

\[ 0 \leq \text{NUM} \text{ (left element)} \leq L - 1 \]
\[ 0 \leq \text{NUM} \text{ (right element)} \leq L - 1 \]
\[ 0 \leq \text{NUM} \text{ (start element)} \leq L - 1 \]
\[ \text{NUM} \text{ (start element)} + \text{NUM} \text{ (slice size)} \leq L \]

where \( L \) is the actual length of the string location. If \( L \) and the value all integer expressions are known statically, the relations can be checked statically.

**dynamic conditions:** The RANGEFAIL exception occurs if a dynamic part of the check of the relations above fails.

**examples:**

18.26 blanks (count : 9) \hspace{1cm} (1.1)
18.23 string ->(scanstart UP 10) \hspace{1cm} (1.2)

### 4.2.8 Array elements

**syntax:**

\[
\langle \text{array element} \rangle ::= \\
\langle \text{array location} \rangle \ ( \langle \text{expression list} \rangle ) \]

\[
\langle \text{expression list} \rangle ::= \\
\langle \text{expression} \rangle \ \{ \ , \langle \text{expression} \rangle \ \}^{*} \]

**derived syntax:** The notation: \( ( \langle \text{expression list} \rangle ) \) is derived syntax for:

\( ( \langle \text{expression} \rangle ) \ \{ ( \langle \text{expression} \rangle ) \ \}^{*} \)

where there are as many parenthesized expressions as there are expressions in the expression list. Thus an array element in the strict syntax has only one (index) expression.

**semantics:** An array element delivers a (sub-)location which is the element of the specified array location indicated by expression.

**static properties:** The mode attached to the array element is the element mode of the mode of the array location.

An array element is referable if the element layout of the mode of the array location is NOPACK.

**static conditions:** The class of the expression must be compatible with the index mode of the mode of the array location.

**dynamic conditions:** The RANGEFAIL exception occurs if the following relation does not hold:

\[ L \leq \text{expression} \leq U \]

where \( L \) and \( U \) are the lower bound and the (possibly dynamic) upper bound of the mode of the array location, respectively.

**examples:**

11.36 \( b(m.lin_1)(m.col_1) \) \hspace{1cm} (1.1)
4.2.9 Array slices

syntax:

<array slice> ::= (1)
<array location> ( <lower element> : <upper element> ) (1.1)
| <array location> ( <first element> UP <slice size> ) (1.2)

<lower element> ::= (2)
<expression> (2.1)

<upper element> ::= (3)
<expression> (3.1)

<first element> ::= (4)
<expression> (4.1)

semantics: An array slice delivers a (possibly dynamic) array location which is the part of the specified array location indicated by lower element and upper element or first element and slice size. The lower bound of the array slice is equal to the lower bound of the specified array; the (possibly dynamic) upper bound is determined from the specified expressions.

static properties: The (possibly dynamic) mode attached to an array slice is a parameterized array mode constructed as:

&name (upper index)

where &name is a virtual symmode name synonymous with the (possibly dynamic) mode of the array location and upper index is either an expression whose class is compatible with the classes of lower element and upper element and delivers a value such that:

\[ \text{NUM}(\text{upper index}) = \text{NUM}(L) + \text{NUM}(\text{upper element}) - \text{NUM}(\text{lower element}) \]

or is an expression whose class is compatible with the class of first element and delivers a value such that:

\[ \text{NUM}(\text{upper index}) = \text{NUM}(L) + \text{NUM}(\text{slice size}) - 1 \]

where \(L\) is the lower bound of the mode of the array location.

The mode attached to an array slice is static if upper index is literal, i.e. lower element and upper element are both literal or slice size is literal; otherwise the mode is dynamic.

An array slice is referable if the element layout of the mode of the array location is NOPACK.

static conditions: The classes of lower element and upper element or the class of first element must be compatible with the index mode of the array location.

The following relations must hold:

\[ L \leq \text{NUM}(\text{lower element}) \leq \text{NUM}(\text{upper element}) \leq U \]
\[ 1 \leq \text{NUM}(\text{slice size}) \leq \text{NUM}(U) - \text{NUM}(L) + 1 \]
\[ \text{NUM}(L) \leq \text{NUM}(\text{first element}) \leq \text{NUM}(\text{first element}) + \text{NUM}(\text{slice size}) - 1 \leq \text{NUM}(U) \]

where \(L\) and \(U\) are respectively the lower bound and upper bound of the mode of the array location. If \(U\) and the value of all expressions are known statically, the relations can be checked statically.

dynamic conditions: The RANGEFAIL exception occurs if a dynamic part of the check of the relations above fails.

examples:

17.27 res (0 : count – 1) (1.1)
4.2.10 Structure fields

syntax:

<structure field> ::=<structure location> . <field name> (1)

semantics: A structure field delivers a (sub-)location which is the field of the specified structure location indicated by field name. If the structure location has a tag-less variant structure mode and the field name is a variant field name, the semantics are implementation defined.

static properties: The mode of the structure field is the mode of the field name.

A structure field is referable if the field layout of the field name is NOPACK.

static conditions: The field name must be a name from the set of field names of the mode of the structure location.

dynamic conditions: A location must not denote:

• a tagged variant structure mode location in which the associated tag field value(s) indicate(s) that the field does not exist;

• a dynamic parameterized structure mode location in which the associated list of values indicates that the field does not exist.

The above mentioned conditions are called the variant field access conditions for the location. The TAGFAIL exception occurs if they are not satisfied for the structure location.

examples:

10.57 last ->.info (1.1)

4.2.11 Location procedure calls

syntax:

<location procedure call> ::=<location procedure call> (1)

semantics: A location procedure call delivers the location returned from the procedure.

static properties: The mode attached to a location procedure call is the mode of the result spec of the location procedure call if DYNAMIC is not specified in it; otherwise it is the dynamically parameterized version of it that has the same parameters as the mode of the delivered location.

The location procedure call is referable if NONREF is not specified in the result spec of the location procedure call.

dynamic conditions: The location procedure call must not deliver an undefined location and the lifetime of the delivered location must not have ended.

4.2.12 Location built-in routine calls

syntax:

<location built-in routine call> ::=<location built-in routine call> (1)

semantics: A location built-in routine call delivers the location returned from the built-in routine call.

static properties: The mode attached to the location built-in routine call is the mode of the result spec of the location built-in routine call.

dynamic conditions: The location built-in routine call must not deliver an undefined location and the lifetime of the delivered location must not have ended.
4.2.13 Location conversions

**syntax:**

\[
\langle \text{location conversion} \rangle ::= (1) \\
\langle \text{mode name} \rangle \# ( \langle \text{static mode location} \rangle )
\]  

(1.1)

**semantics:** A location conversion delivers the location denoted by \textit{static mode location}. However, it overrides the CHILL mode checking and compatibility rules and explicitly attaches a mode to the location without any change in the internal representation.

The precise dynamic semantics of a location conversion are implementation defined.

**static properties:** The mode of a \textit{location conversion} is the \textit{mode name}.

A location conversion is \textit{referable}.

**static conditions:** The \textit{static mode location} must be \textit{referable}.

The following relation must hold:

\[
\text{SIZE} (\text{mode name}) = \text{SIZE} (\text{static mode location})
\]

4.2.14 Predefined moreta location

**syntax:**

\[
\langle \text{predefined moreta location} \rangle ::= (1) \\
\text{SELF}
\]  

(1.1)

**semantics:** In a component procedure and/or process \textit{P} of a \textit{moreta mode}, \textit{SELF} denotes that moreta location \textit{ML} to which \textit{P} is currently being applied. The mode of \textit{SELF} is the mode of \textit{ML}.

**static conditions:** The use of \textit{SELF} is allowed only inside the definition of a moreta mode.

5 Values and their operations

5.1 Synonym definitions

**syntax:**

\[
\langle \text{synonym definition statement} \rangle ::= (1) \\
\text{SYN} \langle \text{synonym definition} \rangle \{, \langle \text{synonym definition} \rangle\}^* ; \\
\langle \text{synonym definition} \rangle ::= (2) \\
\langle \text{defining occurrence list} \rangle [ \langle \text{mode} \rangle ] = \langle \text{constant value} \rangle
\]  

(2.1)

**derived syntax:** A synonym definition, where \textit{defining occurrence list} consists of more than one \textit{defining occurrence}, is derived from several \textit{synonym definition} occurrences, one for each \textit{defining occurrence} with the same \textit{constant value} and \textit{mode}, if present. E.g. \textit{SYN i}, \textit{j = 3}; is derived from \textit{SYN i = 3}, \textit{j = 3};.

**semantics:** A synonym definition defines a name that denotes the specified \textit{constant value}.

**static properties:** A \textit{defining occurrence} in a synonym definition defines a \textit{synonym} name.

The class of the \textit{synonym} name is, if a \textit{mode} is specified, the M-value class, where M is the \textit{mode}, otherwise the class of the \textit{constant value}.

A \textit{synonym} name is \textit{undefined} if and only if the \textit{constant value} is an \textit{undefined} value (see 5.3.1).

A \textit{synonym} name is \textit{literal} if and only if the \textit{constant value} is \textit{literal}.

**static conditions:** If a \textit{mode} is specified, it must be \textit{compatible} with the class of the \textit{constant value} and the value delivered by the \textit{constant value} must be one of the values defined by the \textit{mode}.
The evaluation of the constant value must not depend directly or indirectly on the constant value of the synonym name.

examples:

1.17 \[ \text{SYN} \text{ neutral}\_\text{for}\_\text{add} = 0, \]
\[ \text{neutral}\_\text{for}\_\text{mult} = 1; \] (1.1)

2.18 \[ \text{neutral}\_\text{for}\_\text{add} \text{ fraction} = [0,1] \] (2.1)

5.2 Primitive value

5.2.1 General

syntax:

\[
\text{\textless primitive value\textgreater } ::= \\
\text{\textless location contents\textgreater } \\
\text{\textless value name\textgreater } \\
\text{\textless literal\textgreater } \\
\text{\textless tuple\textgreater } \\
\text{\textless value string element\textgreater } \\
\text{\textless value string slice\textgreater } \\
\text{\textless value array element\textgreater } \\
\text{\textless value array slice\textgreater } \\
\text{\textless value structure field\textgreater } \\
\text{\textless expression conversion\textgreater } \\
\text{\textless representation conversion\textgreater } \\
\text{\textless value procedure call\textgreater } \\
\text{\textless value built-in routine call\textgreater } \\
\text{\textless start expression\textgreater } \\
\text{\textless zero-adic operator\textgreater } \\
\text{\textless parenthesized expression\textgreater }
\]

semantics: A primitive value is the basic constituent of an expression. Some primitive values have a dynamic class, i.e. a class based on a dynamic mode. For these primitive values the compatibility checks can only be completed at run time. Check failure will then result in the TAGFAIL or RANGEFAIL exception.

static properties: The class of the primitive value is the class of the location contents, value name, etc., respectively.

A primitive value is constant if and only if it is a constant value name, a literal, a constant tuple, a constant expression conversion, a constant representation conversion, a constant value built-in routine call or a constant parenthesized expression.

A primitive value is literal if and only if it is a value name that is literal, a discrete literal, or a value built-in routine call that is literal.

5.2.2 Location contents

syntax:

\[
\text{\textless location contents\textgreater } ::= \\
\text{location}\]

semantics: A location contents delivers the value contained in the specified location. The location is accessed to obtain the stored value.

static properties: The class of the location contents is the M-value class, where M is the (possibly dynamic) mode of the location.

static conditions: The mode of the location must not have the non-value property.
**ISO-IEC 9496 : 2002(E)**

**dynamic conditions**: The delivered value must not be **undefined**.

**examples:**

3.7  

c2.im  

(1.1)

**5.2.3 Value names**

**syntax:**

```plaintext
<value name> ::= (1)
    synonym name> (1.1)
    | <value enumeration name> (1.2)
    | <value do-with name> (1.3)
    | <value receive name> (1.4)
    | <general procedure name> (1.5)
```

**semantics**: A value name delivers a value. A value name is one of the following:

- a **synonym** name, i.e. a name defined in a synonym definition statement;
- a **value enumeration** name, i.e. a name defined by a *loop counter* in a *value enumeration*;
- a **value do-with** name, i.e. a *field* name introduced as value name in the *do action* with a *with part*;
- a **value receive** name, i.e. a name introduced in a *receive case action*;
- a **general procedure** name (see 10.4).

If the value denoted by a **value do-with** name is a variant field of a tag-less variant structure value, the semantics are implementation defined.

**static properties**: The class of a *value name* is the class of the **synonym name**, **value enumeration name**, **value do-with name**, **value receive name** or the M-derived class, where M is the mode of the *general procedure name*, respectively.

A *value name* is **literal** if and only if it is a **synonym name** that is **literal**.

A *value name* is **constant** if it is a **synonym name** or a **general procedure name** denoting a *procedure* name which has attached a *procedure definition* which is not surrounded by a block.

**static conditions**: The **synonym name** must not be **undefined**.

**dynamic conditions**: Evaluating a **value do-with name** causes a **TAGFAIL** exception if the denoted value is a **variant** field and the variant field access conditions for the value are not satisfied.

**examples:**

10.12  

max  

(1.1)

8.8  

i  

(1.2)

15.54  

this_counter  

(1.4)

**5.2.4 Literals**

**5.2.4.1 General**

**syntax:**

```plaintext
literal> ::= (1)
    integer literal> (1.1)
    | <floating point literal> (1.2)
    | <boolean literal> (1.3)
    | <character literal> (1.4)
    | <set literal> (1.5)
    | <emptiness literal> (1.6)
    | <character string literal> (1.7)
    | <bit string literal> (1.8)
```

**semantics**: A literal delivers a **constant** value.
static properties: The class of the literal is the class of the integer literal, boolean literal, etc., respectively. A literal is discrete if it is either an integer literal, a boolean literal, a character literal or a set literal.

The letter together with the following apostrophe which starts an integer literal, boolean literal, bit string literal, wide character literal or wide character string literal (i.e. B', D', H', O', W', b', d', h', o', w') is a literal qualification.

5.2.4.2 Integer literals

syntax:

\[
<\text{integer literal}> ::= \quad (1) \\
\quad \text{unsigned integer literal} (1.1) \\
\quad \text{signed integer literal} (1.2)
\]

\[
<\text{unsigned integer literal}> ::= \quad (2) \\
\quad <\text{decimal integer literal}> (2.1) \\
\quad <\text{binary integer literal}> (2.2) \\
\quad <\text{octal integer literal}> (2.3) \\
\quad <\text{hexadecimal integer literal}> (2.4)
\]

\[
<\text{signed integer literal}> ::= \quad (3) \\
\quad -<\text{unsigned integer literal}> (3.1)
\]

\[
<\text{decimal integer literal}> ::= \quad (4) \\
\quad [ \{ D | d \} ' ] <\text{digit sequence}> (4.1)
\]

\[
<\text{binary integer literal}> ::= \quad (5) \\
\quad \{ B | b \} ' \{ 0 | 1 | _ \} + (5.1)
\]

\[
<\text{octal integer literal}> ::= \quad (6) \\
\quad \{ O | o \} ' \{ \text{octal digit} | _ \} + (6.1)
\]

\[
<\text{hexadecimal integer literal}> ::= \quad (7) \\
\quad \{ H | h > \} ' \{ \text{hexadecimal digit} | _ \} + (7.1)
\]

\[
<\text{hexadecimal digit}> ::= \quad (8) \\
\quad <\text{digit}> | A | B | C | D | E | F | a | b | c | d | e | f (8.1)
\]

\[
<\text{octal digit}> ::= \quad (9) \\
\quad 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 (9.1)
\]

\[
<\text{digit sequence}> ::= \quad (10) \\
\quad \{ <\text{digit}> | _ \} + (10.1)
\]

semantics: An integer literal delivers an integer value. The usual decimal (base 10) notation is provided as well as binary (base 2), octal (base 8) and hexadecimal (base 16). The underline character ( _ ) is not significant, i.e. it serves only for readability and it does not influence the denoted value.

A signed integer literal delivers a value which is the additive inverse of that delivered by the unsigned integer literal in it.

static properties: The class of an integer literal is the &INT-derived class. An integer literal is constant and literal.

static conditions: The string following the apostrophe ( ’ ) and the digit sequence must not consist solely of underline characters.

The value delivered by integer literal must be one of the values defined by the &INT mode.

examples:

6.11 1_721_119 (2.1) 
D’1_721_119 (2.1) 
B’101011_110100 (2.2) 
O’53_64 (2.3) 
H’AF4 (2.4)
5.2.4.3 Floating point literals

syntax:

\[
\text{\begin{align*}
<\text{floating point literal}> & ::= \\
& \text{unsigned floating point literal} \\
& | \text{signed floating point literal}
\end{align*}}
\]

\[
\text{\begin{align*}
<\text{unsigned floating point literal}> & ::= \\
& <\text{digit sequence}> . [ <\text{digit sequence}> ] [ <\text{exponent}> ] \\
& [ <\text{digit sequence}> ] . <\text{digit sequence}> [ <\text{exponent}> ]
\end{align*}}
\]

\[
\text{\begin{align*}
<\text{signed floating point literal}> & ::= \\
& -<\text{unsigned floating point literal}>
\end{align*}}
\]

\[
\text{\begin{align*}
<\text{exponent}> & ::= \\
& E <\text{digit sequence}> \\
& E - <\text{digit sequence}>
\end{align*}}
\]

**derived syntax:** A floating point literal in which 1. a digit sequence, 2. an exponent is missing is derived syntax for a literal in which 1. the digit sequence is 0, 2. the exponent is E1.

**semantics:** A floating point literal delivers a floating point value, expressed as a decimal number in scientific notation.

A signed floating point literal delivers a value which is the additive inverse of that delivered by the unsigned floating point literal in it.

If the floating point literal lies between the upper bound and lower bound of one of the predefined floating point modes of the implementation but is not exactly representable, the floating point literal value is approximated to the value delivered by an implicit representation conversion to the predefined floating point mode chosen by the implementation for representing the floating point literal.

**static properties:** The class of a floating point literal is the &FLOAT-derived class. A floating point literal is constant and literal.

The precision of a floating point literal is the sum of the number of significant decimal digits delivered by the two digit sequences that form its mantissa.

**static conditions:** The value delivered by floating point literal must be one of the values defined by the &FLOAT mode.

**examples:**

\[
\begin{align*}
10.0E1 & \quad (1.1) \\
-365.0E-5 & \quad (1.1)
\end{align*}
\]

5.2.4.4 Boolean literals

syntax:

\[
\text{\begin{align*}
<\text{boolean literal}> & ::= \\
& \text{boolean literal name}
\end{align*}}
\]

**predefined names:** The names FALSE and TRUE are predefined as boolean literal names.

**semantics:** A boolean literal delivers a boolean value.

**static properties:** The class of a boolean literal is the BOOL-derived class. A boolean literal is constant and literal.

**examples:**

5.42 FALSE \quad (1.1)
5.2.4.5 Character literals

syntax:

\[
\text{<character literal> ::= (1)} \\
\text{| narrow character literal> (1.1)} \\
\text{| <wide character literal> (1.2)} \\
\text{<narrow character literal> ::= (2)} \\
\text{\' { <character> | <control sequence> } \'} (2.1) \\
\text{<wide character literal> ::= (3)} \\
\text{\{ W | w \'} { <character> | <control sequence> } \'} (3.1) \\
\text{<control sequence> ::= (4)} \\
\text{^ (<integer literal expression> | , <integer literal expression> )* (4.1)} \\
\text{| ^ <non-special character> (4.2)} \\
\text{| ^^ (4.3)}
\]

semantics: A character literal delivers a character value.

Apart from the printable representation, the control sequence representation may be used. A control sequence in which the circumflex character (\(^\wedge\)) is followed by an open parenthesis denotes the sequence of characters whose representations are the integer literal expression in it; otherwise if it is followed by another circumflex character it denotes itself, otherwise it denotes the character whose representation is obtained by logically negating the b7 of the internal representation of the non-special character in it (see 12.4.4 and Appendix I).

static properties: The class of a narrow character literal is the CHAR-derived class. The class of a wide character literal is the WCHAR-derived class. A character literal is constant and literal.

static conditions: A control sequence in a character literal must denote only one character.

The value delivered by an integer literal expression in a control sequence must belong to the range of values defined by the representations of the characters in the CHILL character set (see Appendix I) in case of narrow character literal or to the set of values defined by the representations of characters in the set of characters of ISO/IEC 10646-1 in case of wide character literal.

examples:

7.9 'M' (2.1)

5.2.4.6 Set literals

syntax:

\[
\text{<set literal> ::= (1)} \\
\text{[ <mode name> . ] <set element name> (1.1)}
\]

semantics: A set literal delivers a set value. A set literal is a name defined in a set mode.

static properties: The class of a set literal is the M-value class, where M is the mode name, if specified. Otherwise, M depends upon the context where the set literal occurs, according to the following list:

- if the set literal is used in a place where a tuple without the mode name can be used, then M is derived following the same rules defined for the tuple (see 5.2.5);
- if the set literal is used as a value in a tuple, then M is the mode of that value;
- if the set literal is used in a literal range to define a discrete range mode of the form:

\[
\text{<discrete mode name> ( <literal range> )}
\]

then M is the discrete mode name;
- if the set literal is the usage expression, the where expression, the index expression or the write expression in a built-in routine for input output (see 7.4), then M is respectively USAGE, WHERE, the index mode of the access location or of the text location, the record mode of the access location;
- if the set literal is used in a conditional expression, then M is derived in the same way as for the expression in which it is contained;
• if the set literal is the upper index in a parameterized array mode, then M is the corresponding index mode of the origin array mode;
• if the set literal is an expression in a parameterized structure mode, then M is the root mode of the corresponding tag field name in the origin variant structure mode;
• if the set literal is used in an array element or array slice, then M is the corresponding index mode in the array mode;
• if the set literal is used in a case label, then M is derived from the mode of the corresponding tag field name (for structure mode), from the mode of the corresponding selector in the case selector list (for case action or conditional expression), or from the index mode (for tuple);
• if the set literal is used as the lower bound or the upper bound and a discrete mode name is specified in the literal range in which it is contained, then M is the discrete mode name.

A set literal is constant and literal.

static conditions: The optional mode name may be omitted only in the contexts specified above.

The set element name must belong to the set of set element names of M.

examples:
6.51 dec (1.1)
11.78 king (1.1)

5.2.4.7 Emptiness literal

syntax:

<emptiness literal> ::= (1)
<emptiness literal name> (1.1)

predefined names: The name NULL is predefined as an emptiness literal name.

semantics: The emptiness literal delivers either the empty reference value, i.e. a value which does not refer to a location, the empty procedure value, i.e. a value which does not indicate a procedure, or the empty instance value, i.e. a value which does not identify a process.

static properties: The class of the emptiness literal is the null class. An emptiness literal is constant.

examples:
10.43 NULL (1.1)

5.2.4.8 Character string literals

syntax:

<character string literal> ::= (1)
<character string literal> (1.1)
| <wide character string literal> (1.2)

<narrow character string literal> ::= (2)
" { <non-reserved character> | <quote> | <control sequence> } * " (2.1)

<wide character string literal> ::= (3)
{ W' | w' } " { <non-reserved wide character> | <quote> | <control sequence> } * " (3.1)

<quote> ::= (4) (4.1)

semantics: A character string literal delivers a character string value that may be of length 0. It is a list of values for the elements of the string; the values are given for the elements in increasing order of their index from left to right. To represent the character quote (") within a character string literal, it has to be written twice ("").

static properties: The string length of a character string literal is the number of non-reserved character, quote and characters denoted by control sequence occurrences.
The class of a character string literal is the CHARS \(n\)-derived class, where \(n\) is the string length of the narrow character string literal. The class of a character string literal is the WCHARS \(n\)-derived class, where \(n\) is the string length of the wide character string literal. A character string literal is constant.

**examples:**

8.20 "A-B<ZAA9K"  \(\text{(2.1)}\)

5.2.4.9 Bit string literals

**syntax:**

\[
\text{<bit string literal> ::= } \\
\text{<binary bit string literal> } \\
\text{<octal bit string literal> } \\
\text{<hexadecimal bit string literal>} \\
\text{<binary bit string literal> ::= } \\
\{ \text{B | b} \}'\{0 | 1 | _\}^* ' \\
\text{<octal bit string literal> ::= } \\
\{ \text{O | o} \}'\{\text{octal digit} | _\}^* ' \\
\text{<hexadecimal bit string literal> ::= } \\
\{ \text{H | h} \}'\{\text{hexadecimal digit} | _\}^* '
\]

**semantics:** A bit string literal delivers a bit string value that may be of length 0. Binary, octal or hexadecimal notations may be used. The underline character ( _) is insignificant, i.e. it serves only for readability and does not influence the indicated value.

A bit string literal is a list of values for the elements of the string; the values are given for the elements in increasing order of their index from left to right.

**static properties:** The string length of a bit string literal is either the number of 0 and 1 occurrences in a binary bit string literal, three times the number of octal digit occurrences in an octal bit string literal or four times the number of hexadecimal digit occurrences in a hexadecimal bit string literal.

The class of a bit string literal is the BOOLS \(n\)-derived class, where \(n\) is the string length of the bit string literal. A bit string literal is constant.

**examples:**

\('B'101011_110100' \text{ (1.1)}\)

\('O'53_64' \text{ (1.2)}\)

\('H'AF4' \text{ (1.3)}\)

5.2.5 Tuples

**syntax:**

\[
\text{<tuple> ::= } \\
\{ \text{<mode name}> \} : \{ \text{<powerset tuple>} | \text{<array tuple>} | \text{<structure tuple>} \} : \}
\text{<powerset tuple> ::= } \\
\{ \{ \text{<expression> | <range>} \} , \{ \text{<expression> | <range>} \} ^* \}
\text{<range> ::= } \\
\text{<expression> : <expression>}
\text{<array tuple> ::= } \\
\text{<unlabelled array tuple> | <labelled array tuple>}
\text{<unlabelled array tuple> ::= } \\
\{ \{ \text{<value> | <value>} \} *
\]

\(\text{\textit{examples:}}\)

\(\text{\textit{\textbf{\textit{\textbf{\textit{\textbf{B'101011_110100'}}}}}} (1.1)\)}

\(\text{\textit{\textbf{\textit{\textbf{\textbf{O'53_64'}}}} (1.2)\)}

\(\text{\textit{\textbf{\textit{\textbf{H'AF4'}}}} (1.3)\)}
ISO-IEC 9496 : 2002(E)

<labelled array tuple> ::= (case label list : <value> { , <case label list> : <value>})*
(6)

<structure tuple> ::= (unlabelled structure tuple) |
                     <labelled structure tuple>
(7)

<unlabelled structure tuple> ::= <value> { , <value>}*
(8)

<labelled structure tuple> ::= <field name list> : <value> { , <field name list> : <value>}*
(9)

<label name list> ::= . <field name> { , . <field name>}*
(10)

derived syntax: The tuple opening and closing brackets, [ and ], are derived syntax for (: and :), respectively. This is not indicated in the syntax to avoid confusion with the use of square brackets as meta symbols.

semantics: A tuple delivers either a powerset value, an array value or a structure value.

If it is a powerset value, it consists of a list of expressions and/or ranges denoting those member values which are in the powerset value. A range denotes those values which lie between or are one of the values delivered by the expressions in the range. If the second expression delivers a value which is less than the value delivered by the first expression, the range is empty, i.e. it denotes no values. The powerset tuple may denote the empty powerset value.

If it is an array value, it is a (possibly labelled) list of values for the elements of the array; in the unlabelled array tuple, the values are given for the elements in increasing order of their index; in the labelled array tuple, the values are given for the elements whose indices are specified in the case label list labelling the value. It can be used as a shorthand for large array tuples where many values are the same. The label ELSE denotes all the index values not mentioned explicitly. The label * denotes all index values (for further details, see 12.3).

If it is a structure value, it is a (possibly labelled) set of values for the fields of the structure. In the unlabelled structure tuple, the values are given for the fields in the same order as they are specified in the attached structure mode. In the labelled structure tuple, the values are given for the fields whose field names are specified in the field name list for the value.

The order of evaluation of the expressions and values in a tuple is undefined and they may be considered as being evaluated in any order.

static properties: The class of a tuple is the M-value class, where M is the mode name, if specified. Otherwise M depends upon the context where the tuple occurs, according to the following list:

• if the tuple is the value or constant value in an initialization in a location declaration, then M is the mode in the location declaration;

• if the tuple is the right-hand side value in a single assignment action, then M is the (possibly dynamic) mode of the left-hand side location;

• if the tuple is the constant value in a synonym definition with a specified mode, then M is that mode;

• if the tuple is used in an operand-2 and one of the operands is strong, then M is the mode of the strong operand;

• if the tuple is an actual parameter in a procedure call or in a start expression where DYNAMIC is not specified in the corresponding parameter spec, then M is the mode in the corresponding parameter spec;

• if the tuple is the value in a return action or a result action, then M is the mode of the result spec of the procedure name of the result action or return action (see 6.8);

• if the tuple is a value in a send action, then it is the associated mode specified in the signal definition of the signal name or the buffer element mode of the mode of the buffer location;

• if the tuple is an expression in an array tuple, then M is the element mode of the mode of the array tuple;
• if the tuple is an expression in an unlabelled structure tuple or a labelled structure tuple where the associated field name list consists of only one field name, then M is the mode of the field in the structure tuple for which the tuple is specified;

• if the tuple is the value in a GETSTACK or ALLOCATE built-in routine call, then M is the mode denoted by mode argument.

A tuple is constant if and only if each value or expression occurring in it is constant.

static conditions: The optional mode name may be omitted only in the contexts specified above. Depending on whether a powerset tuple, array tuple or structure tuple is specified, the following compatibility requirements must be fulfilled:

a) Powerset tuple
   1) The mode of the tuple must be a powerset mode.
   2) The class of each expression must be compatible with the member mode of the mode of the tuple.
   3) For a constant powerset tuple the value delivered by each expression must be one of the values defined by that member mode.

b) Array tuple
   1) The mode of the tuple must be an array mode.
   2) The class of each value must be compatible with the element mode of the mode of the tuple.
   3) In the case of an unlabelled array tuple, there must be as many occurrences of value as the number of elements of the array mode of the tuple.
   4) In the case of a labelled array tuple, the case selection conditions must hold for the list of case label list occurrences (see 12.3). The resulting class of the list must be compatible with the index mode of the mode of the tuple. The list of case label specifications must be complete.
   5) In the case of a labelled array tuple, the values explicitly indicated by each case label in a case label list must be values defined by the index mode of the tuple.
   6) In an unlabelled array tuple, at least one value occurrence must be an expression.
   7) For a constant array tuple, where the element mode of the mode of the tuple is a discrete mode, each specified value must deliver a value defined by that element mode, unless it is an undefined value.

c) Structure tuple
   1) The mode of the tuple must be a structure mode.
   2) This mode must not be a structure mode which has field names which are invisible (see 12.2.5).

In the case of an unlabelled structure tuple:

– If the mode of the tuple is neither a variant structure mode nor a parameterized structure mode, then:
   3) There must be as many occurrences of value as there are field names in the list of field names of the mode of the tuple.
   4) The class of each value must be compatible with the mode of the corresponding (by position) field name of the mode of the tuple.

– If the mode of the tuple is a tagged variant structure mode or a tagged parameterized structure mode, then:
   5) Each value specified for a tag field must be a discrete literal expression.
   6) There must be as many occurrences of value as there are field names indicated as existing by the value(s) delivered by the discrete literal expression occurrences specified for the tag fields.
   7) The class of each value must be compatible with the mode of the corresponding field name.

– If the mode of the tuple is a tag-less variant structure mode or a tag-less parameterized structure mode, then:
   8) No unlabelled structure tuple is allowed.
In the case of a labelled structure tuple:

- If the mode of the tuple is neither a **variant** structure mode nor a **parameterized** structure mode, then:
  
  9) Each **field** name of the list of field names of the mode of the tuple must be mentioned once and only once in the tuple.

  10) The class of each **value** must be **compatible** with the mode of every **field** name specified in the **field name list** labelling that **value**. The modes of all **field** names in the **field name list** must be **equivalent**.

- If the mode of the tuple is a tagged variant structure mode or a tagged parameterized structure mode, then:

  11) Each **value** that is specified for a **tag** field must be a **discrete literal expression**.

  12) Each **field** name that denotes a fixed field or a field indicated as existing by the **discrete literal expression** occurrences specified for the **tag** fields must be mentioned once and only once in the tuple.

  13) The class of each **value** must be **compatible** with the mode of any **field** name specified in the **field name list** labelling that **value**.

- If the mode of the tuple is a tag-less variant structure mode or a tag-less parameterized structure mode, then:

  14) Each **field** name must be mentioned at most once in the tuple. All the **fixed field** names must be mentioned. **Field** names mentioned in the tuple, which are defined in the same alternative field, must all be defined in the same variant alternative or all be defined after **ELSE**. All **field** names of an alternative field in each variant alternative or all **field** names defined after **ELSE** must be mentioned.

  15) The class of each **value** must be **compatible** with the mode of any **field** name specified in the **field name list** labelling that **value**.

  16) If the mode of the tuple is a tagged parameterized structure mode, the list of values delivered by the **discrete literal expression** occurrences specified for the **tag** fields must be the same as the list of values of the mode of the tuple.

  17) For a **constant structure tuple**, each **value** specified for a field with a discrete mode must deliver a value defined by the **field** mode, unless it is an **undefined** value.

  18) At least one **value** occurrence must be an **expression**.

No tuple may have two **value** occurrences in it, such that one is **extra-regional** and the other is **intra-regional** (see 11.2.2).

**Dynamic conditions**: The assignment conditions of any value with respect to the **member** mode, **element** mode or associated **field** mode, in the case of powerset tuple, array tuple or structure tuple, respectively (see 6.2) apply (refer to conditions a2, b2, c4, c7, c10, c13 and c15).

If the tuple has a dynamic array mode, the **RANGEFAIL** exception occurs if any of the conditions b3 or b5 are not satisfied.

If the tuple has a dynamic parameterized structure mode, the **TAGFAIL** exception occurs if any of the conditions c14 or c16 are not satisfied.

The value delivered by a tuple must not be **undefined**.

**Examples**:

9.6  number_list [ ]^  (1.1)

9.7  [2: max]  (2.1)

8.26  ['A':3, 'B','K','Z':1, (ELSE):0]  (6.1)

17.5  [('*'):']'  (6.1)

12.35  (:NULL,NULL,536:)  (7.1)

11.18  [.status:occupied,.p:[white,rook]]  (9.1)
5.2.6 Value string elements

syntax:

<value string element> ::= (1)
<primitive string value> ( <start element> ) (1.1)

NOTE – If the string primitive value is a string location, the syntactic construct is ambiguous and will be interpreted as a string element (see 4.2.6).

semantics: A value string element delivers a value which is the element of the specified string value indicated by start element.

static properties: The class of the value string element is the M-value class, where M is the element mode of the mode of the string primitive value.

A value string element is constant if and only if string primitive value and start element are constant.

dynamic conditions: The value delivered by a value string element must not be undefined.

The RANGEFAIL exception occurs if the following relation does not hold:

\[ 0 \leq \text{NUM} (\text{start element}) \leq L - 1 \]

Where \( L \) is the actual length of the string primitive value.

5.2.7 Value string slices

syntax:

<value string slice> ::= (1)
<primitive string value> ( <left element> : <right element> ) (1.1)
| <primitive string value> ( <start element> UP <slice size> ) (1.2)

NOTE – If the string primitive value is a string location, the syntactic construct is ambiguous and will be interpreted as a string slice (see 4.2.7).

semantics: A value string slice delivers a (possibly dynamic) string value which is the part of the specified string value indicated by left element and right element or start element and slice size. The (possibly dynamic) length of the string slice is determined from the specified expressions.

A string slice in which the right element delivers a value which is less than that delivered by the left element or in which slice size delivers a non positive value denotes an empty string.

static properties: The (possibly dynamic) class of a value string slice is the M-value class if the string primitive value is strong and otherwise the M-derived class, where M is a parameterized string mode constructed as:

&name (string size)

where &name is a virtual synmode name synonymous with the (possibly dynamic) root mode of the string primitive value if it is a fixed string mode, otherwise with the component mode, and where string size is either

\[ \text{NUM} (\text{right element}) - \text{NUM} (\text{left element}) + 1 \]

or

\[ \text{NUM} (\text{slice size}) \]

However, if an empty string is denoted, string size is 0. The class of a value string slice is static if string size is literal, i.e. left element and right element are literal or slice size is literal; otherwise the class is dynamic.

A value string slice is constant if and only if string primitive value and string size are constant.
static conditions: The following relations must hold:

\[
0 \leq \text{NUM (left element)} \leq L - 1
\]

\[
0 \leq \text{NUM (right element)} \leq L - 1
\]

\[
0 \leq \text{NUM (start element)} \leq L - 1
\]

\[
\text{NUM (start element)} + \text{NUM (slice size)} \leq L
\]

where \( L \) is the actual length of the string primitive value. If \( L \) and the value all integer expressions are known statically, the relations can be checked statically.

dynamic conditions: The value delivered by a value string slice must not be undefined.

The RANGEFAIL exception occurs if a dynamic part of the check of the relations above fails.

5.2.8 Value array elements

syntax:

\[
<\text{value array element}> ::= \begin{cases} 
<\text{array primitive value}> ( <\text{expression list}> ) & (1) \\
<\text{array primitive value}> ( <\text{first element}> : <\text{upper element}> ) & (1.1) \\
| <\text{array primitive value}> ( <\text{first element}> \text{ UP} <\text{slice size}> ) & (1.2)
\end{cases}
\]

NOTE – If the \text{array primitive value} is an \text{array location} the syntactic construct is ambiguous and will be interpreted as an \text{array element} (see 4.2.8).

derived syntax: See 4.2.8.

semantics: A value array element delivers a value which is the element of the specified array value indicated by expression.

static properties: The class of the value array element is the M-value class, where M is the element mode of the mode of the array primitive value.

A value array element is constant if and only if array primitive value and expression are constant.

static conditions: The class of the expression must be compatible with the index mode of the mode of the array primitive value.

dynamic conditions: The value delivered by a value array element must not be undefined.

The RANGEFAIL exception occurs if the following relation does not hold:

\[
L \leq \text{expression} \leq U
\]

where \( L \) and \( U \) are the lower bound and (possibly dynamic) upper bound of the mode of the array primitive value, respectively.

5.2.9 Value array slices

syntax:

\[
<\text{value array slice}> ::= \begin{cases} 
<\text{array primitive value}> ( <\text{lower element}> : <\text{upper element}> ) & (1) \\
<\text{array primitive value}> ( <\text{first element}> \text{ UP} <\text{slice size}> ) & (1.2)
\end{cases}
\]

NOTE – If the \text{array primitive value} is an \text{array location}, the syntactic construct is ambiguous and will be interpreted as an \text{array slice} (see 4.2.9).

semantics: A value array slice delivers an (possibly dynamic) array value which is the part of the specified array value indicated by lower element and upper element, or first element and slice size. The lower bound of the value array slice is equal to the lower bound of the specified array value; the (possibly dynamic) upper bound is determined from the specified expressions.

static properties: The (possibly dynamic) class of a value array slice is the M-value class, where M is a parameterized array mode constructed as:

\&name (upper index)
where \&name is a virtual synmode name synonymous with the (possibly dynamic) mode of the array primitive value and upper index is either an expression whose class is compatible with the classes of lower element and upper element and delivers a value such that:

\[
NUM (\text{upper index}) = NUM (L) + NUM (\text{upper element}) - NUM (\text{lower element})
\]
or is an expression whose class is compatible with the class of first element and delivers a value such that:

\[
NUM (\text{upper index}) = NUM (L) + NUM (\text{slice size}) - 1
\]

where L is the lower bound of the mode of the array primitive value.

The class of a value array slice is static if upper index is literal, i.e. lower element and upper element both are literal or slice size is literal; otherwise the class is dynamic.

**static conditions:** The classes of lower element and upper element or the class of first element must be compatible with the index mode of the array primitive value.

The following relations must hold:

\[
L \leq NUM (\text{lower element}) \leq NUM (\text{upper element}) \leq U \\
1 \leq NUM (\text{slice size}) \leq NUM (U) - NUM (L) + 1 \\
NUM (L) \leq NUM (\text{first element}) \leq NUM (\text{first element}) + NUM (\text{slice size}) - 1 \leq NUM (U)
\]

where L and U are, respectively, the lower bound and upper bound of the mode of the array primitive value. If U and the value of all expressions are known statically, the relations can be checked statically.

A value array slice is constant if and only if array primitive value and upper index are constant.

**dynamic conditions:** The value delivered by a value array slice must not be undefined.

The RANGEFAIL exception occurs if a dynamic part of the check of the relations above fails.

### 5.2.10 Value structure fields

**syntax:**

```
<value structure field> ::= (1)
  <structure primitive value> . <field name> (1.1)
```

NOTE – If the structure primitive value is a structure location, the syntactic construct is ambiguous and will be interpreted as a structure field (see 4.2.10).

**semantics:** A value structure field delivers a value which is the field of the specified structure value indicated by field name. If the structure primitive value has a tag-less variant structure mode and the field name is a variant field name, the semantics are implementation defined.

**static properties:** The class of value structure field is the M-value class, where M is the mode of the field name.

A value structure field is constant if and only if structure primitive value is constant.

**static conditions:** The field name must be a name from the set of field names of the mode of the structure primitive value.

**dynamic conditions:** The value delivered by a value structure field must not be undefined.

A value must not denote:

- a tagged variant structure mode value in which the associated tag field value(s) indicate(s) that the denoted field does not exist;
- a dynamic parameterized structure mode value in which the associated list of values indicates that the field does not exist.

The above-mentioned conditions are called the variant field access conditions for the value (note that the conditions do not include the occurrence of an exception). The TAGFAIL exception occurs if they are not satisfied for the structure primitive value.

**examples:**

```
11.140 b (lin)(col).status
```

(1.1)
5.2.11 Expression conversion

syntax:

\[
\text{<expression conversion> ::= (1)}
\]

\[
\text{<mode name> # ( <expression> ) (1.1)}
\]

NOTE – If the expression is a static mode location, the syntactic construct is ambiguous and will be interpreted as a location conversion (see 4.2.13).

semantics: An expression conversion overrides the CHILL mode checking and compatibility rules. It explicitly attaches a mode to the expression without any change in the internal representation.

static properties: The class of the expression conversion is the M-value class, where M is the mode name. An expression conversion is constant if and only if the expression is constant.

static conditions: The mode name must not have the non-value property. The size of the root mode of the expression and the size of mode name must be equal.

5.2.12 Representation conversion

syntax:

\[
\text{<representation conversion> ::= (1)}
\]

\[
\text{<mode name> ( <expression> ) (1.1)}
\]

semantics: A representation conversion overrides the CHILL mode checking and compatibility rules. It explicitly attaches a mode to the expression and may change the internal representation of the value delivered by the expression itself. If the mode of the mode name is a discrete mode and the class of the value delivered by the expression is discrete, then the value delivered by the representation conversion is such that:

\[
\text{NUM(mode name(expression)) = NUM(expression)}
\]

A representation conversion in which mode name and the root mode of the class of the expression are respectively:

- an integer mode and a floating point mode;
- a floating point mode and an integer mode;
- a floating point mode and another floating point mode with different root modes,

may involve an approximation. If the value delivered by expression is exactly representable in the set of values of mode name, the result of the representation conversion is the value of expression itself, otherwise it is one of the two values belonging to the set of values of mode name that delimit the smallest interval in which the value delivered by expression is contained. A representation conversion in which mode name is an integer mode and the root mode of the class of the expression is a duration mode, delivers an integer value which represents in milliseconds the value delivered by expression.

A representation conversion in which mode name or the root mode of the class of the expression is a structure mode, and the other one is a parameterized structure mode whose origin structure mode is similar with it, delivers a structure value in which the values of the fields are equal to the corresponding ones of the expression, if present. Otherwise the result is implementation defined. Note that for tag-less variant structure values and for tagged variant structure values in which the list of tag values is different from that of the parameterized structure mode the result of the representation conversion is implementation defined.

A representation conversion in which the mode M of the mode name is a reference mode and the class of the expression is the null class, the result of the representation conversion is null, if M is compatible with the class of \((\text{expression} \rightarrow)\) then the result is equal to it, otherwise the result is implementation defined.

Otherwise, the value delivered by the representation conversion is implementation defined and may depend on the internal representation of values.

static properties: The class of the representation conversion is the M-value class, where M is the mode name. A representation conversion is constant if and only if the expression is constant.

static conditions: The mode name must not have the non-value property. An implementation may impose additional static conditions.
dynamic conditions: In the case of an expression that is not constant:

- a RANGEFAIL exception occurs if mode name is a duration mode and the root mode of the class of the expression is an integer mode (or vice versa), and the value delivered by representation conversion does not belong to the set of values defined for mode name;

- an OVERFLOW exception occurs if:
  - the class of the value delivered by expression is discrete and the mode of mode name is a discrete mode which does not define a value with an internal representation equal to NUM (expression);
  - the mode of mode name and the root mode of the class of the expression are, independently, an integer mode or a floating point mode, and the expression delivers a value that does not lie between the bounds of the root mode of mode name;

- an UNDERFLOW exception occurs if the mode name and the root mode of the class of the expression are floating point modes, and the value delivered by expression is greater than the negative lower limit and less than the positive lower limit of the mode name, and is different from zero.

An implementation may impose additional dynamic conditions that, when violated, cause an exception defined by the implementation.

5.2.13 Value procedure calls

syntax:

```
<value procedure call> ::= (1)
  <value procedure call> (1.1)
```

semantics: A value procedure call delivers the value returned from a procedure.

static properties: The class of the value procedure call is the M-value class, where M is the mode of the result spec of the value procedure call.

dynamic conditions: The value procedure call must not deliver an undefined value (see sections 5.3.1 and 6.8).

examples:

6.50 julian_day_number([ 10, dec, 1979]) (1.1)
11.63 ok_bishop(b, m) (1.1)

5.2.14 Value built-in routine calls

syntax:

```
<value built-in routine call> ::= (1)
  <value built-in routine call> (1.1)
```

semantics: A value built-in routine call delivers the value returned by the built-in routine.

static properties: The class attached to the value built-in routine call is the class of the value built-in routine call.

dynamic conditions: The value built-in routine call must not deliver an undefined value (see sections 5.3.1 and 6.8).

5.2.15 Start expressions

syntax:

```
<start expression> ::= (1)
  START <process name> ( [ <actual parameter list> ] ) (1.1)
```

semantics: The evaluation of the start expression creates and activates a new process whose definition is indicated by the process name (see clause 11). The start expression delivers the instance value identifying the created process. Parameter passing is analogous to procedure parameter passing; however, additional actual parameters may be given with an implementation defined meaning.

static properties: The class of the start expression is the INSTANCE-derived class.

static conditions: The number of actual parameter occurrences in the actual parameter list must not be less than the number of formal parameter occurrences in the formal parameter list of the process definition of the process name. If
the number of actual parameters is \( m \) and the number of formal parameters is \( n \) \( (m \geq n) \), the compatibility and **regionality** requirements for the first \( n \) actual parameters are the same as for procedure parameter passing (see 6.7). The static conditions for the rest of the actual parameters are implementation defined.

**dynamic conditions**: For parameter passing, the assignment conditions of any actual value with respect to the mode of its associated formal parameter apply (see 6.7).

The start expression causes the **SPACEFAIL** exception if storage requirements cannot be satisfied.

**examples:**

15.35 START counter() (1.1)

5.2.16 **Zero-adic operator**

**syntax:**

\[
<\text{zero-adic operator}> ::= \\
\text{THIS} \\
\]

**semantics**: The zero-adic operator delivers the unique instance value identifying the process executing it. If it is executed by a task location a **THIS_FAIL** exception occurs.

**static properties**: The class of the **zero-adic operator** is the **INSTANCE**-derived class.

**static conditions**: The **zero-adic operator** **THIS** must not occur inside a task mode definition.

5.2.17 **Parenthesized expression**

**syntax:**

\[
<\text{parenthesized expression}> ::= \\
( <\text{expression}> ) \\
\]

**semantics**: A parenthesized expression delivers the value delivered by the evaluation of the expression.

**static properties**: The class of the **parenthesized expression** is the class of the **expression**.

A parenthesized expression is **constant** (literal) if and only if the expression is **constant** (literal).

**examples:**

5.10 \((a1 \ OR \ b1)\) (1.1)

5.3 **Values and expressions**

5.3.1 **General**

**syntax:**

\[
<\text{value}> ::= \\
<\text{expression}> \\
| <\text{undefined value}> \\
<\text{undefined value}> ::= \\
\star \\
| <\text{undefined synonym name}> \\
\]

**semantics**: A value is either an **undefined** value or a (CHILL defined) value delivered as the result of the evaluation of an expression.

Except where explicitly indicated to the contrary, the order of evaluation of the constituents of an expression and their sub-constituents, etc., is undefined and they may be considered as being evaluated in any order. They need only be evaluated to the point that the value to be delivered is determined uniquely. If the context requires a **constant** or **literal** expression, the evaluation is assumed to be done prior to run time and cannot cause an exception. An implementation will define ranges of allowed values for **literal** and **constant** expressions and may reject a program if such a prior-to-run-time evaluation delivers a value outside the implementation defined bounds.
**static properties**: The class of a value is the class of the expression or undefined value, respectively.

The class of the undefined value is the all class if the undefined value is a *; otherwise the class is the class of the undefined synonym name.

A value is constant if and only if it is an undefined value or an expression which is constant. A value is literal if and only if it is an expression which is literal.

**dynamic properties**: A value is said to be undefined if it is denoted by the undefined value or when explicitly indicated in this Recommendation | International Standard. A composite value is undefined if and only if all its sub-components (i.e. substring values, element values, field values) are undefined.

**examples:**

6.40 \( (146.097*{c})/4+(1.461*{y})/4 + (153*{m}+2)/5+{d}ay+1_721_119 \) (1.1)

**5.3.2 Expressions**

**syntax:**

\[
\begin{align*}
\text{<expression>} &::= \\
&\text{<operand–0>} \quad (1) \\
&\text{| <conditional expression>} \quad (1.1)
\end{align*}
\]

\[
\begin{align*}
\text{<conditional expression>} &::= \\
&\text{IF <boolean expression> <then alternative>} \\
&\text{<else alternative> FI} \quad (2.1)
\end{align*}
\]

\[
\begin{align*}
\text{CASE <case selector list> OF } \{ \text{<value case alternative>} \}+ \\
&\text{ELSE <sub expression> ] ESAC} \quad (2.2)
\end{align*}
\]

\[
\begin{align*}
\text{<then alternative>} &::= \\
&\text{THEN <sub expression>} \quad (3.1)
\end{align*}
\]

\[
\begin{align*}
\text{<else alternative>} &::= \\
&\text{ELSE <sub expression>} \quad (4.1)
\end{align*}
\]

\[
\begin{align*}
&\text{| ELSIF <boolean expression>} \\
&\text{<then alternative> <else alternative>} \quad (4.2)
\end{align*}
\]

\[
\begin{align*}
\text{<sub expression>} &::= \\
&\text{<expression>} \quad (5.1)
\end{align*}
\]

\[
\begin{align*}
\text{<value case alternative>} &::= \\
&\text{<case label specification> : <sub expression> ;} \quad (6.1)
\end{align*}
\]

**semantics**: If IF is specified, the boolean expression is evaluated and if it yields TRUE, the result is the value delivered by the sub expression in the then alternative, otherwise it is the value delivered by the else alternative.

The value delivered by an else alternative is the value of the sub expression if ELSE is specified, otherwise the boolean expression is evaluated and if it yields TRUE, it is the value delivered by the sub expression in the then alternative, otherwise it is the value delivered by the else alternative.

If CASE is specified, the sub expressions in the case selector list are evaluated and if a case label specification matches, the result is the value delivered by the corresponding sub expression, otherwise it is the value delivered by the sub expression following ELSE (which will be present).

Unused sub expressions in a conditional expression are not evaluated.

**static properties**: If an expression is an operand–0, the class of the expression is the class of the operand–0. If it is a conditional expression, the class of the expression is the M-value class, where M is the mode which depends on the context where the conditional expression occurs according to the same rules that define the mode of the class of a tuple without a mode name (see 5.2.5).

An expression is constant (literal) if and only if it is either an operand–0 which is constant (literal), or a conditional expression in which all boolean expression or case selector list in it are constant (literal) and in which all sub expressions in it are constant (literal).

**static conditions**: If an expression is a conditional expression the following conditions apply:

- a conditional expression may occur only in the contexts in which a tuple without a mode name in front of it may occur;
• each sub expression must be compatible with the mode that is derived from the context with the same rules as for tuples. However, the dynamic part of the compatibility relation applies only to the selected sub expression;

• if CASE is specified, the case selection conditions must be fulfilled (see 12.3), and the same completeness, consistency and compatibility requirements must hold as for the case action (see 6.4);

• no conditional expression may have two sub expression occurrences in it such that one is extra-regional and the other is intra-regional (see 11.2.2).

**dynamic conditions**: In the case of a conditional expression, the assignment conditions of the value delivered by the selected sub expression with respect to the mode M derived from the context apply.

### 5.3.3 Operand–0

**syntax:**

\[
<\text{operand–0}> ::= \\
<\text{operand–1}> (1) \\
| <\text{sub operand–0}> \{ \text{OR} | \text{ORIF} | \text{XOR} \} <\text{operand–1}> (1.2)
\]

\[
<\text{sub operand–0}> ::= \\
<\text{operand–0}> (2.1)
\]

**semantics:** If OR, ORIF or XOR is specified, sub operand–0 and operand–1 deliver:

• boolean values, in which case OR and XOR denote the logical operators "inclusive disjunction" and "exclusive disjunction", respectively, delivering a boolean value. If ORIF is specified and operand–0 delivers the boolean value TRUE, then this is the result, otherwise the result is the value delivered by operand–1;

• bit string values, in which case OR and XOR denote the logical operations on corresponding element of the bit strings, delivering a bit string value;

• powerset values, in which case OR denotes the union of both powerset values and XOR denotes the powerset value consisting of those member values which are in only one of the specified powerset values (e.g. \(A \ XOR \ B = A–B \ OR \ B–A\)).

**static properties:** If an operand–0 is an operand–1, the class of operand–0 is the class of operand–1. If OR, ORIF or XOR is specified, the class of operand–0 is the resulting class of the classes of sub operand–0 and operand–1.

An operand–0 is constant (literal) if and only if it is either an operand–1 which is constant (literal), or built up from an operand–0 and an operand–1 which are both constant (literal).

**static conditions:** If OR, ORIF or XOR is specified, the class of sub operand–0 must be compatible with the class of operand–1. If ORIF is specified, both classes must have a boolean root mode, otherwise both classes must have a boolean, powerset or bit string root mode, in which case the actual length of sub operand–0 and operand–1 must be the same. This check is dynamic if one or both modes are (are) dynamic or varying string modes.

**dynamic conditions:** In the case of OR or XOR, a RANGEFAIL exception occurs if one or both operands have a dynamic class and the dynamic part of the above-mentioned compatibility check fails.

**examples:**

10.31 \(i<\text{min}\) (1.1)  
10.31 \(i<\text{min OR i>max}\) (1.2)

### 5.3.4 Operand–1

**syntax:**

\[
<\text{operand–1}> ::= \\
<\text{operand–2}> (1) \\
| <\text{sub operand–1}> \{ \text{AND} | \text{ANDIF} \} <\text{operand–2}> (1.2)
\]

\[
<\text{sub operand–1}> ::= \\
<\text{operand–1}> (2.1)
\]
**semantics:** If **AND** or **ANDIF** is specified, \( sub\ \text{operand–1} \) and \( \text{operand–2} \) deliver:

- boolean values, in which case **AND** denotes the logical "conjunction" operation, delivering a boolean value. If **ANDIF** is specified and \( sub\ \text{operand–1} \) delivers the boolean value \( FALSE \), then this is the result, otherwise the result is the value delivered by \( \text{operand–2} \);
- bit string values, in which case **AND** denotes the logical operation on corresponding element of the bit strings, delivering a bit string value;
- powerset values, in which case **AND** denotes the "intersection" operation of powerset values delivering a powerset value as a result.

**static properties** If an \( \text{operand–1} \) is an \( \text{operand–2} \), the class of \( \text{operand–1} \) is the class of \( \text{operand–2} \).

If **AND** or **ANDIF** is specified, the class of \( \text{operand–1} \) is the **resulting class** of the classes of \( sub\ \text{operand–1} \) and \( \text{operand–2} \). An \( \text{operand–1} \) is **constant** (literal) if and only if it is either an \( \text{operand–2} \) which is **constant** (literal), or built up from an \( \text{operand–1} \) and an \( \text{operand–2} \) which are both **constant** (literal).

**static conditions:** If **AND** or **ANDIF** is specified, the class of \( sub\ \text{operand–1} \) must be **compatible** with the class of \( \text{operand–2} \). If **ANDIF** is specified, both classes must have a boolean **root** mode, otherwise both classes must have a boolean, powerset or bit string **root** mode, in which case the **actual length** of \( sub\ \text{operand–1} \) and \( \text{operand–2} \) must be the same. This check is dynamic if one or both modes is (are) dynamic or varying string modes.

**dynamic conditions:** In the case of **AND**, a **RANGEFAIL** exception occurs if one or both operands have a dynamic class and the dynamic part of the above-mentioned compatibility check fails.

**examples:**

5.10 \((a1 \ OR \ b1)\) \hspace{1cm} (1.1)

5.10 \(\text{NOT} \ k2 \ AND \ (a1 \ OR \ b1)\) \hspace{1cm} (1.2)

5.3.5 **Operand–2**

**syntax:**

\[
<\text{operand–2}> ::=
\begin{align*}
&<\text{operand–3}> \\
| & <\text{sub operand–2}> <\text{operator–3}> <\text{operand–3}>
\end{align*}
\]

\[
<\text{sub operand–2}> ::=
\begin{align*}
&<\text{operand–2}>
\end{align*}
\]

\[
<\text{operator–3}> ::=
\begin{align*}
&<\text{relational operator}> \\
| & <\text{membership operator}> \\
| & <\text{powerset inclusion operator}>
\end{align*}
\]

\[
<\text{relational operator}> ::=
\begin{align*}
&= | /= | > | >= | < | <= \\
&\text{IN} \ (5.1)
\end{align*}
\]

\[
<\text{membership operator}> ::=
\begin{align*}
&\text{IN} \ (5.1) \\
&<= | >= | < | >
\end{align*}
\]

**semantics:** The equality (=) and inequality (/=) operators are defined between all values of a given mode. The other relational operators (less than: <, less than or equal to: \(<=\), greater than: >, greater than or equal to: \(>=\)) are defined between values of a given discrete, timing, string or floating point mode. All the relational operators deliver a boolean value as result.

The membership operator is defined between a member value and a powerset value. The operator delivers \( TRUE \) if the member value is in the specified powerset value, otherwise \( FALSE \).

The powerset inclusion operators are defined between powerset values and they test whether or not a powerset value is contained in: \(<=\), is properly contained in: <, contains: \(>=\) or properly contains: > the other powerset value. A powerset inclusion operator delivers a boolean value as result.
**static properties:** If an operand–2 is an operand–3, the class of operand–2 is the class of operand–3. If an operator–3 is specified, the class of operand–2 is the BOOL-derived class.

An operand–2 is constant (literal) if and only if it is either an operand–3 which is constant (literal) or built up from a sub operand–2 and an operand–3 which are both constant (literal).

**static conditions:** If an operator–3 is specified, the following compatibility requirements between the class of sub operand–2 and the class of operand–3 must be fulfilled:

- if operator–3 is = or /=, both classes must be compatible;
- if operator–3 is a relational operator other than = or /=, both classes must be compatible and must have a discrete, timing, string or floating point root mode;
- if operator–3 is a membership operator, the class of operand–3 must have a powerset root mode and the class of sub operand–2 must be compatible with the member mode of that root mode;
- if operator–3 is a powerset inclusion operator, both classes must be compatible and must have a powerset root mode.

**dynamic conditions:** In the case of a relational operator, a RANGEFAIL or TAGFAIL exception occurs if one or both operands have a dynamic class and the dynamic part of the above-mentioned compatibility check fails. The TAGFAIL exception occurs if and only if a dynamic class is based upon a dynamic parameterized structure mode.

**examples:**

<table>
<thead>
<tr>
<th>10.50</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.50</td>
<td>last=NULL</td>
</tr>
</tbody>
</table>

5.3.6 **Operand–3**

**syntax:**

```
<operand–3> ::= (1)
   <operand–4> (1.1)
   | <sub operand–3> <operator–4> <operand–4> (1.2)

<sub operand–3> ::= (2)
   <operand–3> (2.1)

<operator–4> ::= (3)
   <arithmetic additive operator> (3.1)
   | <string concatenation operator> (3.2)
   | <powerset difference operator> (3.3)

<arithmetic additive operator> ::= (4)
   + | – (4.1)

<string concatenation operator> ::= (5)
   // (5.1)

<powerset difference operator> ::= (6)
   – (6.1)
```

**semantics:** If operator–4 is an arithmetic additive operator, both operands deliver either integer values or floating point values and the resulting integer value or floating point value respectively is the sum (+) or difference (–) of the two values.

If operator–4 is a string concatenation operator, both operands deliver either bit string values or character string values; the resulting value consists of the concatenation of these values. Boolean (character) values are also allowed; they are regarded as bit (character) string values of length 1.

If operator–4 is the powerset difference operator, both operands deliver powerset values and the resulting value is the powerset value consisting of those member values which are in the value delivered by sub operand–3 and not in the value delivered by operand–4.

If the class of operand–3 has a floating point root mode, the result is the floating point value that approximates, using the same criterion used for representation conversion, the result of the exact mathematical operation.
static properties: If an operand–3 is an operand–4, the class of operand–3 is the class of operand–4. If an operand–4 is specified, the class of operand–3 is determined by operand–4 as follows:

- if operand–4 is a string concatenation operator, the class of operand–3 is dependent on the classes of operand–4 and sub operand–3, in which an operand that is a boolean or a character value is regarded as a value whose class is a BOOLS (1)-derived class or CHARS (1)-derived class, respectively:
  - if none of them is strong, the class is the BOOLS (n)-derived class or CHARS (n)-derived class, depending on whether both operands are bit or character strings, where n is the sum of the string lengths of the root modes of both classes;
  - otherwise the class is the &name(n)-value class, where &name is a virtual synmode name synonymous with the root mode of the resulting class of the classes of the operands and n is the sum of the string lengths of the root modes of both classes;
  (this class is dynamic if one or both operands have a dynamic class);
- if operand–4 is an arithmetic additive operator or powerset difference operator, the class of operand–3 is the resulting class of the classes of operand–4 and sub operand–3.

An operand–3 is constant (literal) if and only if it is either an operand–4 which is constant (literal), or built up from an operand–3 and an operand–4 which are both constant (literal) and operand–4 is either the arithmetic additive operator or the powerset difference operator.

If operand–4 is the string concatenation operator, an operand–3 is constant if it is built up from an operand–3 and operand–4 which are both constant.

static conditions: If an operand–4 is specified, the following compatibility requirements must be fulfilled:

- if operand–4 is the arithmetic additive operator, the classes of both operands must be compatible and they must both have either an integer or a floating point root mode. Furthermore if operand–3 is not constant, the root mode of the class of operand–3 must be a predefined integer mode or a predefined floating point mode.
- if operand–4 is the string concatenation operator then:
  - the classes of both operands must be compatible and they must both have a bit string root mode or both have a character string root mode; or
  - the classes of both operands must be compatible with the BOOL mode or both be compatible with the CHAR mode; or
  - the class of one operand must have a bit (character) string root mode and the other must be compatible with the BOOL (CHAR) mode.
- if operand–4 is the powerset difference operator, the classes of both operands must be compatible and both must have a powerset root mode.

dynamic conditions: In the case of an operand–3 that is not constant, if operand–4 is an arithmetic additive operator, an OVERFLOW exception occurs if an addition (+) or a subtraction (–) gives rise to a value that is not one of the values defined by the root mode of the class of operand–3, or one or both operands do not belong to the set of values of the root mode of operand–3.

In the case of an operand–3 that is not constant, an UNDERFLOW exception occurs if the class of operand–3 has a floating point root mode and the exact mathematical addition (+) or subtraction (–) give rise to a value that is greater than the negative upper limit and less than the positive lower limit of the root mode of operand–3, and is different from zero.

examples:

1.6 \( j \) \text{(1.1)}
1.6 \( i+j \) \text{(1.2)}

5.3.7 Operand–4

syntax:

\[
\langle \text{operand–4} \rangle ::= \\
\langle \text{operand–5} \rangle \quad \text{(1)} \\
\mid \langle \text{sub operand–4} \rangle \langle \text{arithmetic multiplicative operator} \rangle \langle \text{operand–5} \rangle \quad \text{(1.2)}
\]

\[
\langle \text{sub operand–4} \rangle ::= \\
\langle \text{operand–4} \rangle \quad \text{(2)}
\]

ITU-T Rec. Z.200 (1999 E)
If the arithmetic multiplicative operator is either the product (\(\ast\)) or the quotient operator (\(\div\)), then both sub operand–4 and operand–5 deliver either integer values or floating point values and the resulting integer value or floating point value respectively is the product or quotient of both values.

If the arithmetic multiplicative operator is either the modulo (MOD) or division remainder (REM) operator, then both sub operand–4 and operand–5 deliver integer values, and the resulting integer value is the modulo or division remainder of both values.

The modulo operation is defined such that \(i \mod j\) delivers the unique integer value \(k\), \(0 \leq k < j\) such that there is an integer value \(n\) such that \(i = n \ast j + k\); \(j\) must be greater than 0.

The quotient operation is defined such that all relations:

\[
\text{ABS} (x/y) = \text{ABS} (x) / \text{ABS} (y) \text{ and }
\]

\[
\text{sign} (x/y) = \text{sign} (x) / \text{sign} (y) \text{ and }
\]

\[
\text{ABS} (x) - (\text{ABS} (x) / \text{ABS} (y)) \ast \text{ABS} (y) = \text{ABS} (x) \mod \text{ABS} (y)
\]

yield \(\text{TRUE}\) for all integer values \(x\) and \(y\), where \(\text{sign} (x) = -1\) if \(x < 0\), otherwise \(\text{sign} (x) = 1\).

The remainder operation is defined such that \(x \mod y = x - (x/y) \ast y\) yields \(\text{TRUE}\) for all integer values \(x\) and \(y\).

If the class of operand–4 has a floating point root mode, the result is the floating point value that approximates, using the same criterion used for representation conversion, the result of the exact mathematical operation.

**static properties:** If operand–4 is an operand–5, the class of operand–4 is the class of operand–5; otherwise the class of operand–4 is the resulting class of the classes of sub operand–4 and operand–5.

An operand–4 is constant (literal) if and only if it is either an operand–5 which is constant (literal), or built up from an operand–4 and an operand–5 which are both constant (literal).

**static conditions:** If an arithmetic multiplicative operator is specified between integer or floating point operands, then the classes of operand–5 and sub operand–4 must be compatible and both must have an integer root mode or a floating point root mode respectively. Furthermore, if operand–4 is not constant, the root mode of the class of operand–4 must be a predefined integer mode or a predefined floating point mode.

**dynamic conditions:** In the case of an operand–4 that is not constant, if an arithmetic multiplicative operator is specified, an OVERFLOW exception occurs if a multiplication (\(\ast\)), a division (\(\div\)), a modulo (MOD), or a remainder (REM) operation gives rise to a value that is not one of the values defined by the root mode of the class of operand–4 or is performed on operand values for which the operator is mathematically not defined, i.e. division or remainder with an operand–5 delivering 0 or a modulo operation with an operand–5 delivering a non-positive integer value, or one or both operands do not belong to the set of values of the root mode of operand–4.

In the case of an operand–4 that is not constant, an UNDERFLOW exception occurs if the class of operand–4 has a floating point root mode and the exact mathematical multiplication (\(\ast\)) or division (\(\div\)) give rise to a value that is greater than the negative upper limit and less than the positive lower limit of the root mode of operand–4, and is different from zero.

**examples:**

| 6.15  | 1_461 |
| 6.15  | (4 * d + 3) / 1_461 |

5.3.8 **Operand–5**

**syntax:**

\[
\text{<operand–5>} ::= \text{<operand–6>}
\]

(1)

\[
\text{<operand–5>} ::= \text{<sub operand–5> <exponentiation operator> <operand–6>}
\]

(1.1)

\[
\text{<operand–5>} ::= \text{<sub operand–5>}
\]

(1.2)

\[
\text{<operand–5>} ::= \text{<operand–5>}
\]

(2)

\[
\text{<operand–5>} ::= \text{<operand–5>}
\]

(2.1)
exponentiation operator ::= (3)

\( r^{4} \) (1.2)

5.3.9 Operand–6

syntax:

<operand–6> ::= (1)

\[ [ <monadic operator> ] <operand–7> \] (1.1)
\| <signed integer literal> (1.2)
\| <signed floating point literal> (1.3)

<monadic operator> ::= (2)

\[ – | NOT \] (2.1)
\| <string repetition operator> (2.2)

<string repetition operator> ::= (3)

\[ ( <integer literal expression> ) \] (3.1)

NOTE – If the monadic operator is the change sign operator (–) and the operand–7 is an unsigned integer literal or an unsigned floating point literal, the syntactic construct is ambiguous and will be interpreted as a signed integer literal or a signed floating point literal respectively.

semantics: If the monadic operator is a change-sign operator (–), operand–7 delivers an integer value or a floating point value and the resulting integer value or floating point value is the previous integer value or floating point value with its sign changed.

If the monadic operator is NOT, operand–7 delivers a boolean value, a bit string value, or a powerset value. In the first two cases the logical negation of the boolean value or of the elements of the bit string value is delivered. In the latter case, the set complement value, i.e. the set of those member values which are not in the operand powerset value, is delivered.

If the monadic operator is a string repetition operator, operand–7 is a character string literal or a bit string literal. If the integer literal expression delivers 0, the result is the empty string value; otherwise the result is the string value formed by...
concatenating the string with itself as many times as specified by the value delivered by the \textit{integer literal expression} minus 1.

\textbf{static properties}: If operand–6 is an operand–7, the class of operand–6 is the class of operand–7.

If a monadic operator is specified, the class of operand–6 is:

- if the monadic operator is \texttt{--} or \texttt{NOT} then the \textbf{resulting class} of operand–7;
- if the monadic operator is the string repetition operator, then it is the CHARS \((n)\)- or BOOLS \((n)\)-derived class (depending on whether the literal was a character string literal or bit string literal) where \(n = r \ast l\), where \(r\) is the value delivered by the \textit{integer literal expression} and \(l\) is the \textbf{string length} of the string literal.

An operand–6 is \textbf{constant} if and only if the operand–7 is \textbf{constant}. An operand–6 is \textbf{literal} if and only if the operand–7 is \textbf{literal} and the monadic operator is \texttt{--} or \texttt{NOT}.

\textbf{static conditions}: If monadic operator is \texttt{--}, the class of operand–7 must have an integer \textbf{root} mode or a floating point \textbf{root} mode. Furthermore, if operand–6 is not \textbf{constant}, the \textbf{root} mode of the class of operand–6 must be a \textbf{predefined} integer mode or a \textbf{predefined} floating point mode.

If monadic operator is \texttt{NOT}, the class of operand–7 must have a boolean, bit string or powerset \textbf{root} mode.

If monadic operator is the string repetition operator, operand–7 must be a character string literal or a bit string literal. The \textit{integer literal expression} must deliver a non-negative integer-value.

\textbf{dynamic conditions}: If operand–6 is not \textbf{constant}, an \textbf{OVERFLOW} exception occurs if a change sign (\texttt{--}) operation gives rise to a value which is not one of the values defined by the \textbf{root} mode of the class of the operand–6.

In the case of an operand–6 that is not \textbf{constant}, an \textbf{UNDERFLOW} exception occurs if the class of operand–6 has a floating point \textbf{root} mode and the exact mathematical change sign operation (\texttt{--}) give rise to a value that is greater than the \textbf{negative upper limit} and less than the \textbf{positive lower limit} of the \textbf{root} mode of operand–6, and is different from zero.

\textbf{examples}:

\begin{itemize}
  \item 5.10 \texttt{NOT k2} (1.1)
  \item 7.54 \texttt{(6)""} (1.1)
  \item 7.54 \texttt{(6)} (2.2)
\end{itemize}

5.3.10 \textbf{Operand–7}

\textbf{syntax}:

\begin{verbatim}
<operand–7> ::= <referenced location> | <primitive value> (1)
| <referenced location> ::= -> <location> (2)
\end{verbatim}

\textbf{semantics}: A referenced location delivers a reference to the specified location.

\textbf{static properties}: The class of an operand–7 is the class of the referenced location or primitive value, respectively. The class of the referenced location is the M-reference class where M is the mode of the location.

An operand–7 is \textbf{constant} if and only if the primitive value is \textbf{constant} or the referenced location is \textbf{constant}. A referenced location is \textbf{constant} if and only if the location is \textbf{static}. An operand–7 is \textbf{literal} if and only if the primitive value is \textbf{literal}.

\textbf{static conditions}: The location must be \textbf{referable}.

\textbf{examples}:

\begin{itemize}
  \item 8.25 \texttt{-> c} (2.1)
\end{itemize}
6 Actions

6.1 General

**syntax:**

```
<action statement> ::= (1)
[ <defining occurrence> : ] <action> [ <handler> ] [ <simple name string> ] ; (1.1)
| <module>
| <spec module>
| <context module>

<action> ::= (2)
<bracketed action>
| <assignment action>
| <call action>
| <exit action>
| <return action>
| <result action>
| <goto action>
| <assert action>
| <empty action>
| <start action>
| <stop action>
| <delay action>
| <continue action>
| <send action>
| <cause action>
```

**semantics:** Action statements constitute the algorithmic part of a CHILL program. Any action statement may be labelled. Those actions that have no exception defined may not have a handler appended.

**static properties:** A **defining occurrence** in an **action statement** defines a **label** name.

**static conditions:** The **simple name string** may only be given after an **action** which is a **bracketed action** or if a **handler** is specified, and only if a **defining occurrence** is specified. The **simple name string** must be the same name string as the **defining occurrence**.

6.2 Assignment action

**syntax:**

```
<assignment action> ::= (1)
<single assignment action>
| <multiple assignment action>

<single assignment action> ::= (2)
<location> <assignment symbol> <value>
| <location> <assigning operator> <expression>

<multiple assignment action> ::= (3)
<location> { , <location> }+ <assignment symbol> <value>

<assigning operator> ::= (4)
<closed dyadic operator> <assignment symbol>
```
semantics: An assignment action stores a value into one or more locations.

If an assignment symbol is used, the value yielded by the right hand side is stored into the location(s) specified at the left hand side.

If an assigning operator is used, the value contained in the location is combined with the right hand side value (in that order) according to the semantics of the specified closed dyadic operator, and the result is stored back into the same location.

The evaluation of the left hand side location(s), of the right hand side value, and of the assignment themselves are performed in any order. Any assignment may be performed as soon as the value and a location have been evaluated.

If the location (or any of the locations) is the tag field of a variant structure, the semantics for the variant fields that depend on it are implementation defined.

static conditions: The modes of all location occurrences must be equivalent and they must have neither the read-only property nor the non-value property. Each mode must be compatible with the class of the value. The checks are dynamic in the case where dynamic mode locations and/or a value with a dynamic class are involved.

The value must be regionally safe for every location (see 11.2.2).

If any location has a fixed string mode, then the string length of the mode and the actual length of the value must be the same; otherwise, if it has a varying string mode, then the string length of the mode must not be less than the actual length of the value. This check is dynamic if one or both modes is (are) dynamic or varying string modes. This condition is called the string assignment condition.

If one of the assignments is of the form "pvl–> := pvr–>;" where pvl and pvr have the mode "REF ML" and "REF MR" respectively, and ML and MR are moreta mode names, then ML and MR must be on the same path.

If one of the assignments is of the form "pvl–> := mr;" where pvl has the mode "REF ML", and ML and the mode name of mr are module mode names, then MR succ ML must hold.

If one of the assignments is of the form "ml := pvr–>;" where pvr has the mode "REF MR", and MR and the mode name of ml are module mode names, then MR succ ml must hold.

If the mode of any of the locations of the left hand side is a module mode then the mode names of all those modes must be pairwise synonymous.

dynamic conditions: The RANGEFAIL or TAGFAIL exception occurs if the mode of the location and/or that of the value are dynamic modes and the dynamic part of the above-mentioned compatibility checks fails.

The RANGEFAIL exception occurs if the mode of the location and/or that of the value are varying string modes and the dynamic part of the above mentioned compatibility checks fails.

The RANGEFAIL exception occurs if any location has a discrete range mode (floating point range mode) and the value delivered by the evaluation of value is neither one of the values defined by the discrete range mode (floating point range mode) nor the undefined value.

If the mode of any location L is of the kind REF MM, where MM is a moreta mode, the following must hold: the mode of the current value of the rhs must be a successor of the mode of L; otherwise the exception RANGEFAIL occurs.

The above-mentioned dynamic conditions together with the string assignment condition are called the assignment conditions of a value with respect to a mode.

In the case of an assigning operator, the same exceptions are caused as if the expression:

\[ <location> <closed dyadic operator> (<expression>) \]

were evaluated and the delivered value stored into the specified location (note that the location is evaluated once only).
If the mode of any location L is of the kind "REF MM", where MM is a moreta mode, the mode of the current value of the rhs must be a successor of the mode of L. Otherwise the exception RANGEFAIL occurs.

If any of the assignments is of the form "pvl := pvr;", "pvl := mr;" or "ml := pvr;", where pvl and pvr have the modes "REF ML" and "REF MR" respectively and ML, MR, ml, and mr are module modes, then the current modes of the lhs and the rhs must fulfill the rules for the assignment of module modes.

examples:

4.12 \( a := b+c \) \hspace{1cm} (1.1)

10.25 \( \text{stackindex} := 1 \) \hspace{1cm} (2.1)

19.19 \( x \rightarrow .\text{prev}, x \rightarrow .\text{next} := \text{NULL} \) \hspace{1cm} (3.1)

10.25 \( - := \) \hspace{1cm} (4.1)

6.3 If action

syntax:

\[
<\text{if action}> ::= \hspace{1cm} (1)
\]

\[
\text{IF} \hspace{1cm} (1.1)
\]

\[
<\text{boolean expression}> \hspace{1cm} (1.1)
\]

\[
<\text{then clause}> [ <\text{else clause}> ] \hspace{1cm} (2)
\]

\[
\text{THEN} \hspace{1cm} (2.1)
\]

\[
<\text{action statement list}> \hspace{1cm} (2.1)
\]

\[
<\text{else clause}> ::= \hspace{1cm} (3)
\]

\[
<\text{ELSE} \hspace{1cm} (3.1)
\]

\[
<\text{action statement list}> \hspace{1cm} (3.1)
\]

\[
| \hspace{1cm} (3.2)
\]

\[
<\text{ELSIF} <\text{boolean expression}> <\text{then clause}> [ <\text{else clause}> ] \hspace{1cm} (3.2)
\]

derived syntax: The notation:

\[\text{ELSIF} <\text{boolean expression}> <\text{then clause}> [ <\text{else clause}> ] \]

is derived syntax for:

\[\text{ELSE IF} <\text{boolean expression}> <\text{then clause}> [ <\text{else clause}> ] \hspace{1cm} (3.2)\]

semantics: An if action is a conditional two-way branch. If the boolean expression yields TRUE, the action statement list following THEN is entered; otherwise the action statement list following ELSE, if present, is entered.

dynamic conditions: The SPACEFAIL exception occurs if storage requirements cannot be satisfied.

examples:

7.22 \hspace{1cm} \text{IF } n \geq 50 \hspace{1cm} (1.1)
\hspace{1cm} \text{THEN } \text{rn}(r) := 'L';
\hspace{1cm} \hspace{1cm} n_\sim := 50;
\hspace{1cm} \hspace{1cm} r_+ := 1;
\hspace{1cm} \hspace{1cm} \text{FI}

10.50 \hspace{1cm} \text{IF } \text{last} = \text{NULL}
\hspace{1cm} \hspace{1cm} \text{THEN } \text{first}, \text{last} := p;
\hspace{1cm} \hspace{1cm} \hspace{1cm} \text{ELSE } \text{last} \rightarrow . \text{succ} := p;
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} p \rightarrow . \text{pred} := \text{last};
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \text{last} := p;
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \text{FI}

6.4 Case action

syntax:

\[
<\text{case action}> ::= \hspace{1cm} (1)
\]

\[
\text{CASE} \hspace{1cm} (1.1)
\]

\[
<\text{case selector list}> \hspace{1cm} (1.1)
\]

\[
\text{OF} \hspace{1cm} (1.1)
\]

\[
[ <\text{range list}> ; ] \{ <\text{case alternative}> \} + \hspace{1cm} (1.1)
\]

\[
| \hspace{1cm} (1.1)
\]

\[
\text{ELSE} \hspace{1cm} (1.1)
\]

\[
<\text{action statement list}> \hspace{1cm} (1.1)
\]

\[
\text{ESAC} \hspace{1cm} (1.1)
\]
semantics: A case action is a multiple branch. It consists of the specification of one or more discrete expressions (the case selector list) and a number of labelled action statement lists (case alternatives). Each action statement list is labelled with a case label specification which consists of a list of case label list specifications (one for each case selector). Each case label list defines a set of values. The use of a list of discrete expressions in the case selector list allows selection of an alternative based on multiple conditions.

The case action enters that action statement list for which values given in the case label specification match the values in the case selector list; if no value match, the action statement list following ELSE is entered.

The expressions in the case selector list are evaluated in any order. They need be evaluated only up to the point where a case alternative is uniquely determined.

static conditions: For the list of case label specification occurrences, the case selection conditions apply (see 12.3).

The number of discrete expression occurrences in the case selector list must be equal to the number of classes in the resulting list of classes of the list of case label list occurrences and, if present, to the number of discrete mode name occurrences in the range list.

The class of any discrete expression in the case selector list must be compatible with the corresponding (by position) class of the resulting list of classes of the case label list occurrences and, if present, compatible with the corresponding (by position) discrete mode name in the range list. The latter mode must also be compatible with the corresponding class of the resulting list of classes.

Any value delivered by a discrete literal expression or defined by a literal range or by a discrete mode name in a case label (see 12.3) must lie in the range of the corresponding discrete mode name of the range list, if present, and also in the range defined by the mode of the corresponding discrete expression in the case selector list, if it is a strong discrete expression. In the latter case, the values defined by the corresponding discrete mode name of the range list, if present, must also lie in that range.

The optional ELSE part according to the syntax may only be omitted if the list of case label list occurrences is complete (see 12.3).

dynamic conditions: The RANGEFAIL exception occurs if a range list is specified and the value delivered by a discrete expression in the case selector list does not lie within the bounds specified by the corresponding discrete mode name in the range list.

The SPACEFAIL exception occurs if storage requirements cannot be satisfied.

examples:

4.11   CASE order OF
       (1):   a := b+c;
             RETURN;
       (2):   d := 0;
             (ELSE): d := 1;
       ESAC

11.43   starting.p.kind, starting.p.color

11.58   (rook),(*):
       IF NOT ok_rook(b,m)
       THEN
           CAUSE illegal;
       FI;
6.5 Do action

6.5.1 General

syntax:

<do action> ::= (1)  
   DO [ <control part> ; ] <action statement list> OD (1.1)

<control part> ::= (2)  
   <for control> [ <while control> ] (2.1)  
   | <while control> (2.2)  
   | <with part> (2.3)

semantics: A do action has one out of three different forms: the do-for and the do-while versions, both for looping, and the do-with version as a convenient short hand notation for accessing structure fields in an efficient way. If no control part is specified, the action statement list is entered once, each time the do action is entered.

When the do-for and the do-while versions are combined, the while control is evaluated after the for control, and only if the do action is not terminated by the for control.

If the specified control part is a for control and/or while control, then for as long as control stays inside the reach of the do action, the action statement list is entered according to the control part, but the do reach is not re-entered for each execution of the action statement list.

dynamic conditions: The SPACEFAIL exception occurs if storage requirements cannot be satisfied.

eamples:

4.17   DO FOR i := 1 TO c;
       op(a,b,d,order–1);
       d := a;
       OD (1.1)

15.58  DO WITH each;
       IF this_counter = counter
           THEN
               status := idle;
               EXIT find_counter;
       FI;
       OD (1.1)

6.5.2 For control

syntax:

<for control> ::= (1)  
   FOR { <iteration> ; , <iteration> }* EVER (1.1)

<iteration> ::= (2)  
   <value enumeration> (2.1)  
   | <location enumeration> (2.2)

<value enumeration> ::= (3)  
   <step enumeration> (3.1)  
   | <range enumeration> (3.2)  
   | <powerset enumeration> (3.3)

<step enumeration> ::= (4)  
   <loop counter> <assignment symbol> <start value> [ <step value> ] [ DOWN ] <end value> (4.1)

<loop counter> ::= (5)  
   <defining occurrence> (5.1)

<start value> ::= (6)  
   <discrete expression> (6.1)

<step value> ::= (7)  
   BY <integer expression> (7.1)
<end value> ::= (8)
TO <discrete expression>
     (8.1)

<range enumeration> ::= (9)
<loop counter> [ DOWN ] IN <discrete mode name>
     (9.1)

<powerset enumeration> ::= (10)
<loop counter> [ DOWN ] IN <powerset expression>
     (10.1)

<location enumeration> ::= (11)
<loop counter> [ DOWN ] IN <composite object>
     (11.1)

<composite object> ::= (12)
<array location>
     (12.1)
| <array expression>
     (12.2)
| <string location>
     (12.3)
| <string expression>
     (12.4)

NOTE – If the composite object is a string location or an array location, the syntactic ambiguity is resolved by interpreting composite object as a location rather than an expression.

semantics: The for control may mention several loop counters. The loop counters are evaluated each time in an unspecified order, before entering the action statement list, and they need be evaluated only up to the point that it can be decided to terminate the do action. The do action is terminated if at least one of the loop counters indicates termination.

1) do for ever:
The action list is indefinitely repeated. The do action can only terminate by a transfer of control out of it.

2) value enumeration:
The action statement list is repeatedly entered for the set of specified values of the loop counters. The set of values is either specified by a discrete mode name (range enumeration), or by a powerset value (powerset enumeration), or by a start value, step value and end value (step enumeration).

The loop counter implicitly defines a name which denotes its value or location inside the action statement list.

range enumeration:
In the case of range enumeration without (with) DOWN specification, the initial value of the loop counter is the smallest (greatest) value in the set of values defined by the discrete mode name. For subsequent executions of the action statement list, the next value will be evaluated as:

SUCC(previous value) (PRED(previous value)).

Termination occurs if the action statement list has been executed for the greatest (smallest) value defined by the discrete mode name.

powerset enumeration:
In the case of powerset enumeration without (with) DOWN specification, the initial value of the loop counter is the smallest (highest) member value in the denoted powerset value. If the powerset value is empty, the action statement list will not be executed. For subsequent executions of the action statement list, the next value will be the next greater (smaller) member value in the powerset value. Termination occurs if the action statement list has been executed for the greatest (smallest) value. When the do action is executed, the powerset expression is evaluated only once.

step enumeration:
In the case of step enumeration without (with) DOWN specification, the set of values of the loop counter is determined by a start value, an end value, and possibly a step value. When the do action is executed, these expressions are evaluated only once in any order. The step value is always positive. The test for termination is made before each execution of the action statement list. Initially, a test is made to determine whether the start value of the loop counter is greater (smaller) than the end value. For subsequent executions, next value will be evaluated as:

previous value + step value (previous value – step value)
in the case of step value specification; otherwise as:

SUCC(previous value) (PRED(previous value)).
Termination occurs if the evaluation yields a value which is greater (smaller) than the end value or would have caused an OVERFLOW exception.

3) location enumeration:

In the case of a location enumeration without (with) DOWN specification, the action statement list is repeatedly entered for a set of locations which are the elements of the array location denoted by array location or the components of the string location denoted by string location. If an array expression or a string expression is specified that is not a location, a location containing the specified value will be implicitly created. The lifetime of the created location is the do action. The mode of the created location is dynamic if the value has a dynamic class. The semantics are as if before each execution of the action statement list the loc-identity declaration:

```plaintext
DCL <loop counter> <mode> LOC := <composite object> (<index>);
```

were encountered, where mode is the element mode of the array location or &name(1) such that &name is a virtual synmode name synonymous with the mode of the string location if it is a fixed string mode, otherwise with the component mode, and where index is initially set to the lower bound (upper bound) of the mode of location and index before each subsequent execution of the action statement list is set to SUCC (index) (PRED (index)). The action statement list will not be executed if the actual length of the string location equals 0. The do action is terminated if index just after an execution of the action statement list is equal to the upper bound (lower bound) of the mode of location. When the do action is executed, the composite object is evaluated only once.

**static properties:** A loop counter has a name string attached which is the name string of its defining occurrence.

**value enumeration:** The name defined by the loop counter is a value enumeration name.

**step enumeration:** The class of the name defined by a loop counter is the resulting class of the classes of the start value, the step value, if present, and the end value.

**range enumeration:** The class of the name defined by the loop counter is the M-value class, where M is the discrete mode name.

**powerset enumeration:** The class of the name defined by the loop counter is the M-value class, where M is the member mode of the mode of the (strong) powerset expression.

**location enumeration:** The name defined by the loop counter is a location enumeration name. Its mode is the element mode of the mode of the array location or array expression or the string mode &name(1), where &name is a virtual synmode name synonymous with the mode of string location or the root mode of the string expression.

A location enumeration name is referable if the element layout of the mode of the array location is NOPACK.

**static conditions:** The classes of start value, end value and step value, if present, must be pairwise compatible.

The root mode of the class of a loop counter in a value enumeration must not be a numbered set mode.

If the root mode of the class of a loop counter is an integer mode, there must exist a predefined integer mode that contains all the values delivered by start value, end value and step value, if present.

**dynamic conditions:** A RANGEFAIL exception occurs if the value delivered by step value is not greater than 0. This exception occurs outside the block of the do action.

**examples:**

```
4.17    FOR i := 1 TO c  
15.37   FOR EVER        
4.17    i := 1 TO c     
9.12    j := MIN (sieve) BY MIN (sieve) TO max
14.28   i IN INT (1:100) 
```
6.5.3 While control

syntax:

\[
<\text{while control}> ::= \\
\quad \text{WHILE} \ <\text{boolean expression}> \quad (1)
\]

semantics: The boolean expression is evaluated just before entering the action statement list (after the evaluation of the for control, if present). If it yields \text{TRUE}, the action statement list is entered; otherwise the do action is terminated.

examples:

7.35 \text{WHILE}\ n >= 1 \quad (1.1)

6.5.4 With part

syntax:

\[
<\text{with part}> ::= \\
\quad \text{WITH} \ <\text{with control}> \ \{, <\text{with control}> \}\* \\
\quad <\text{with control}> ::= \\
\quad \quad <\text{structure location}> \quad (2) \\
\quad \quad | <\text{structure primitive value}> \quad (2.1)
\]

NOTE – If the with control is a structure location, the syntactic ambiguity is resolved by interpreting with control as a location rather than a primitive value.

semantics: The (visible) field names of the mode of the structure locations or structure value specified in each with control are made available as direct accesses to the fields.

The visibility rules are as if a field name defining occurrence were introduced for each field name attached to the mode of the location or primitive value and with the same name string as the field name.

If a structure location is specified, access names with the same name string as the field names of the mode of the structure location are implicitly declared, denoting the sub-locations of the structure location.

If a structure primitive value is specified, value names with the same name string as the field names of the mode of the (strong) structure primitive value are implicitly defined, denoting the sub-values of the structure value.

When the do action is entered, the specified structure locations and/or structure values are evaluated once only on entering the do action, in any order.

static properties: The (virtual) defining occurrence introduced for a field name has the same name string as the field name defining occurrence of that field name.

If a structure primitive value is specified, a (virtual) defining occurrence in a with part defines a value do-with name. Its class is the M-value class, where M is the mode of that field name of the structure mode of the structure primitive value which is made available as value do-with name.

If a structure location is specified, a (virtual) defining occurrence in a with part defines a location do-with name. Its mode is the mode of that field name of the mode of the structure location which is made available as location do-with name. A location do-with name is referable if the field layout of the associated field name is NOPACK.

examples:

15.58 \text{WITH}\ each \quad (1.1)

6.6 Exit action

syntax:

\[
<\text{exit action}> ::= \\
\quad \text{EXIT} \ <\text{label name}> \quad (1)
\]

semantics: An exit action is used to leave a bracketed action statement or a module. Execution is resumed immediately after the closest surrounding bracketed action statement or module labelled with the label name.
**static conditions:** The *exit action* must lie within the bracketed action statement or module of which the *defining occurrence* in front has the same name string as *label name*.

If the *exit action* is placed within a procedure or process definition, the exited bracketed action statement or module must also lie within the same procedure or process definition (i.e. the exit action cannot be used to leave procedures or processes).

No *handler* may be appended to an *exit action*.

**examples:**

15.62 **EXIT** find_counter

6.7 **Call action**

**syntax:**

\[
<\text{call action}> ::= (1) \\
\quad <\text{procedure call}> (1.1) \ |
\quad <\text{built-in routine call}> (1.2) \ |
\quad <\text{moreta component procedure call}> (1.3) \\
\]

\[
<\text{procedure call}> ::= (2) \\
\quad \{ <\text{procedure name}> | <\text{procedure primitive value} > \} \\
\quad \{ <\text{actual parameter list}> \} (2.1)
\]

\[
<\text{actual parameter list}> ::= (3) \\
\quad <\text{actual parameter}> \{ , <\text{actual parameter}> \}^*$ (3.1)
\]

\[
<\text{actual parameter}> ::= (4) \\
\quad <\text{value}> (4.1) \ |
\quad <\text{location}> (4.2)
\]

\[
<\text{built-in routine call}> ::= (5) \\
\quad <\text{built-in routine name}> ( \{ <\text{built-in routine parameter list}> \} ) (5.1)
\]

\[
<\text{built-in routine parameter list}> ::= (6) \\
\quad <\text{built-in routine parameter}> \{ , <\text{built-in routine parameter}> \}^* (6.1)
\]

\[
<\text{built-in routine parameter}> ::= (7) \\
\quad <\text{value}> (7.1) \ |
\quad <\text{location}> (7.2) \ |
\quad <\text{non-reserved name}> [ ( <\text{built-in routine parameter list}> ) ] (7.3)
\]

\[
<\text{moreta component procedure call}> ::= (8) \\
\quad <\text{moreta location}> \cdot <\text{moreta component procedure call}> [ <\text{priority}> ] (8.1) \ |
\quad <\text{bound reference moreta location primitive value}> \rightarrow \cdot \ |
\quad <\text{moreta component procedure call}> [ <\text{priority}> ] (8.2) \ |
\quad <\text{moreta component procedure call}> [ <\text{priority}> ] (8.3)
\]

**NOTE** – If the *actual parameter* or *built-in routine parameter* is a *location*, the syntactic ambiguity is resolved by interpreting it as a *location* rather than a *value*.

**derived syntax:** A procedure call \( P(... \) of a *moreta component procedure* \( P \) is derived syntax for \( \text{SELF.P}(...) \).

**semantics:** A call action causes the call of either a procedure, a built-in routine, or a moreta component procedure. A procedure call causes a call of the *general* procedure indicated by the value delivered by the *procedure primitive value* or the procedure indicated by the *procedure name*. A *moreta component procedure call* \( \text{L.name}(...) \) causes the call of that moreta component procedure which is identified by name in the mode of \( \text{L} \). \( \text{L} \) is passed as an initial location parameter to the procedure. The actual values and locations specified in the actual parameter list are passed to the procedure.

A *built-in routine call* is either a *CHILL built-in routine call* or an implementation *built-in routine call* (see 6.20 and 13.1, respectively).

A value, a location, or any program defined name that is not a *reserved* simple name string may be passed as *built-in routine parameter*. The built-in routine call may return a value or a location.
A built-in routine may be generic, i.e. its class (if it is a value built-in routine call) or its mode (if it is a location built-in routine call) may depend not only on the built-in routine name but also on the static properties of the actual parameters passed and the static context of the call.

A moreta component procedure call has always the structure "location . procedure call". This is characterized by the expression "the procedure call is applied to the location".

For a moreta component procedure call the following steps are performed:

a) the called procedure is applied to a module mode location:
   1) evaluation of the actual parameters
   2) check of the precondition
   3) check of the complete invariant
   4) execution of the body of the procedure
   5) check of the complete invariant
   6) check of complete postcondition
   7) return to the calling point

b) the called procedure is applied to a region mode location RL:
   1) evaluation of the actual parameters
   2) wait until RL is free and lock RL
   3) check of the precondition
   4) check of the complete invariant
   5) execution of the body of the procedure
   6) check of the complete invariant
   7) check of complete postcondition
   8) release RL
   9) return to the calling point

c) the called procedure is applied to a task mode location TL:
the caller performs the following steps:
   1) evaluation of the actual parameters
   2) send procedure identification, actual parameters and priority to TL
   3) continue with next action
TL performs the following steps:
   1) receive procedure identification and actual parameters according to priority
   2) check of the precondition
   3) check of the complete invariant
   4) execution of the body of the procedure
   5) check of the complete invariant
   6) check of complete postcondition

static properties: A procedure call has the following properties attached: a list of parameter specs, possibly a result spec, a possibly empty set of exception names, a generality, a recursivity, and possibly it is intra-regional (the latter is only possible with a procedure name, see 11.2.2). These properties are inherited from the procedure name, moreta component procedure name or any mode compatible with the class of the procedure primitive value (in the latter case, the generality is always general).

A procedure call with a result spec is a location procedure call if and only if LOC is specified in the result spec; otherwise it is a value procedure call.

A built-in routine name is a CHILL or an implementation defined name that is considered to be defined in the reach of the imaginary outermost process definition or in any context (see 10.8).
A built-in routine call is a location built-in routine call if it delivers a location; it is a value built-in routine call if it delivers a value.

**static conditions:** A priority can only be used in a call of a procedure applied to a task location.

The number of actual parameter occurrences in the procedure call must be the same as the number of its parameter specs. The compatibility requirements for the actual parameter and corresponding (by position) parameter spec of the procedure call are:

- If the parameter spec has the IN attribute (default), the actual parameter must be a value whose class is compatible with the mode in the corresponding parameter spec. The latter mode must not have the non-value property. The actual parameter is a value which must be regionally safe for the procedure call.

- If the parameter spec has the INOUT or OUT attribute, the actual parameter must be a location, whose mode must be compatible with the M-value class, where M is the mode in the corresponding parameter spec. The mode of the (actual) location must be static and must not have the read-only property nor the non-value property. The actual parameter is a location. It can be viewed as a value which must be regionally safe for the procedure call.

- If the parameter spec has the INOUT attribute, the mode in the parameter spec must be compatible with the M-value class where M is the mode of the location.

- If the parameter spec has the LOC attribute specified without DYNAMIC, the actual parameter must be a location which is both referable and such that the mode in the parameter spec is read-compatible with the mode of the (actual) location, or the actual parameter must be a value which is not a location but whose class is compatible with the mode in the parameter spec. If the mode of the formal parameter is a moreta mode, the mode name of the formal parameter and the mode name of the actual parameter must be synonymous. If the mode of the formal parameter is of the form "REF MM", where MM is a moreta mode, the mode of the formal parameter and the mode of the actual parameter must be similar.

- If the parameter spec has the LOC attribute with DYNAMIC specified, the actual parameter must be a location which is both referable and such that the mode in the parameter spec is dynamic read-compatible with the mode of the (actual) location, or the actual parameter must be a value which is not a location but whose class is compatible with a parameterized version of this mode.

- If the parameter spec has the LOC attribute then:
  - if the actual parameter is a location it must have the same regionality as the procedure call;
  - if the actual parameter is a value then it must be regionally safe for the procedure call.

**dynamic conditions:** A call action can cause any of the exceptions from the attached set of exception names. A procedure call causes the EMPTY exception if the procedure primitive value delivers NULL. A call action causes the SPACEFAIL exception if storage requirements cannot be satisfied. If the recursivity of the procedure is non-recursive, then the procedure must not call itself either directly or indirectly.

Parameter passing can cause the following exceptions:

- If the parameter spec has the IN or INOUT attribute, the assignment conditions of the (actual) value with respect to the mode of the parameter spec apply at the point of the call (see 6.2) and the possible exceptions are caused before the procedure is called.

- If the parameter spec has the INOUT or OUT attribute, the assignment conditions of the local value of the formal parameter with respect to the mode of the (actual) location apply at the point of return (see 6.2) and possible exceptions are caused after the procedure has returned.

- If the parameter spec has the LOC attribute and the actual parameter is a value which is not a location, the assignment conditions of the (actual) value with respect to the mode of the parameter spec apply at the point of the call and the possible exceptions are caused before the procedure is called (see 6.2).

Assertion checking can cause the following exceptions:

- If the precondition evaluates to FALSE the exception PREFAIL is caused. The search for an appropriate handler begins at the end of the procedure body and continues according to 8.3.

- If the postcondition evaluates to FALSE the exception POSTFAIL is caused. The search for an appropriate handler begins at the end of the procedure body and continues according to 8.3.

- If the invariant evaluates to FALSE the exception INVFAIL is caused. The search for an appropriate handler begins at the end of the body of the corresponding moreta mode and continues according to 8.3.
The procedure primitive value must not deliver a procedure defined within a process definition whose activation is not the same as the activation of the process executing the procedure call (other than the imaginary outermost process) and the lifetime of the denoted procedure must not have ended.

If a call is applied to a task location TL then TL must not be ended.

examples:

4.18 \( op(a,b,d,\text{order}–1) \) \( (1.1) \)

6.8 Result and return action

syntax:

\[
\begin{align*}
\langle \text{return action} \rangle & ::= \langle \text{result} \rangle \quad (1) \\
& \quad \text{RETURN} \ [ \langle \text{result} \rangle ] \quad (1.1) \\
\langle \text{result action} \rangle & ::= \langle \text{result} \rangle \quad (2) \\
& \quad \text{RESULT} \ <\langle \text{result} \rangle> \quad (2.1) \\
\langle \text{result} \rangle & ::= \langle \text{value} \rangle \quad (3) \\
& \quad \mid \langle \text{location} \rangle \quad (3.1)
\end{align*}
\]

derived syntax: The return action with result is derived from \texttt{DO RESULT <\text{result}> ; RETURN; OD}.

semantics: A result action serves to establish the result to be delivered by a procedure call. This result may be a location or a value. A return action causes the return from the invocation of the procedure within whose definition it is placed. If the procedure returns a result, this result is determined by the latest executed result action. If no result action has been executed, the procedure call delivers an undefined location or undefined value, respectively.

static properties: A result action and a return action have a procedure name attached, which is the name of the closest surrounding procedure definition.

static conditions: A return action and a result action must be textually surrounded by a procedure definition. A result action may only be specified if its procedure name has a result spec.

A handler must not be appended to a return action (without result).

If LOC (LOC DYNAMIC) is specified in the result spec of the procedure name of the result action, the result must be a location, such that the mode in the result spec is read-compatible (dynamic read-compatible) with the mode of the location. The location must be referable if NONREF is not specified in the result spec. The result is a location which must have the same regionality as the procedure name attached to the result action.

If LOC is not specified in the result spec of the procedure name of the result action, the result must be a value, whose class is compatible with the mode in the result spec. The result is a value which must be regionally safe for the procedure name attached to the result action.

dynamic conditions: If LOC is not specified in the result spec of the procedure name, the assignment conditions of the value in the result action with respect to the mode in the result spec of its procedure name apply.

examples:

4.21 \( \text{RETURN} \) \( (1.1) \)

1.6 \( \text{RESULT } i+j \) \( (2.1) \)

5.19 \( c \) \( (3.1) \)

6.9 Goto action

syntax:

\[
\langle \text{goto action} \rangle ::= \langle \text{label name} \rangle \quad (1)
\]

semantics: A goto action causes a transfer of control. Execution is resumed with the action statement labelled with the label name.
**static conditions**: If a goto action is placed within a procedure or process definition, the label indicated by the label name must also be defined within the definition (i.e. it is not possible to jump outside a procedure or process invocation).

A handler must not be appended to a goto action.

### 6.10 Assert action

**syntax:**

```
<assert action> ::= (1)
   ASSERT <boolean expression> (1.1)
```

**semantics**: An assert action provides a means of testing a condition.

**dynamic conditions**: The ASSERTFAIL exception occurs if the boolean expression delivers FALSE.

**examples:**

4.7  \text{ASSERT } b > 0 \text{ AND } c > 0 \text{ AND } order > 0 (1.1)

### 6.11 Empty action

**syntax:**

```
<empty action> ::= (1)
   <empty> (1.1)
```

**semantics**: An empty action causes no action.

**static conditions**: A handler must not be appended to an empty action.

### 6.12 Cause action

**syntax:**

```
<cause action> ::= (1)
   CAUSE <exception name> (1.1)
```

**semantics**: A cause action causes the exception whose name is indicated by exception name to occur.

**static conditions**: A handler must not be appended to a cause action.

**examples:**

4.9  \text{CAUSE wrong\_input} (1.1)

### 6.13 Start action

**syntax:**

```
<start action> ::= (1)
   <start expression> (1.1)
```

**semantics**: A start action evaluates the start expression (see 5.2.15) without using the resulting instance value.

**examples:**

14.45 \text{START call\_distributor ( )} (1.1)

### 6.14 Stop action

**syntax:**

```
<stop action> ::= (1)
   STOP (1.1)
```

**semantics**: A stop action terminates the process executing it (see 11.1).

**static conditions**: A handler must not be appended to a stop action.
6.15 Continue action

**syntax:**

\[
\text{<continue action> ::=}
\]
\[
\text{CONTINUE <event location>}
\]

**semantics:** A continue action evaluates the *event location*.

If the event location has a non-empty set of delayed processes attached, one of these, with the highest priority, will be re-activated. If there are several such processes, one will be selected in an implementation defined way. If there are no such processes, the continue action has no further effect.

If a process becomes re-activated, it is removed from all sets of delayed processes of which it was a member.

**Examples:**

13.25 CONTINUE resource_freed (1.1)

6.16 Delay action

**syntax:**

\[
\text{<delay action> ::=}
\]
\[
\text{DELAY <event location> [ <priority> ]}
\]

**semantics:** A delay action evaluates the *event location*.

Then a \textit{DELAYFAIL} exception occurs (see below) or the executing process becomes delayed.

If the executing process becomes delayed, it becomes a member with a priority of the set of delayed processes attached to the specified event location. The priority is the one specified, if any, otherwise 0 (lowest).

**Dynamic properties:** A process executing a delay action becomes \textit{timeoutable} when it reaches the point of execution where it may become delayed. It ceases to be \textit{timeoutable} when it leaves that point.

**Static conditions:** The \textit{integer literal expression} must not deliver a negative value.

**Dynamic conditions:** The \textit{DELAYFAIL} exception occurs if the *event location* has a mode with an *event length* attached which is equal to the number of processes already delayed on the event location.

The lifetime of the *event location* must not end while the executing process is delayed on it.

**Examples:**

13.18 DELAY resource_freed (1.1)

6.17 Delay case action

**syntax:**

\[
\text{<delay case action> ::=}
\]
\[
\text{DELAY CASE [ SET <instance location> [ <priority> ] ; <priority> ; ]}
\]
\[
\{ <delay alternative> ; <priority> ; \}^+
\]
\[
\text{ESAC}
\]

**semantics:** A delay case action evaluates, in any order, the *instance location*, if present, and all *event locations* specified in a *delay alternative*.

Then a \textit{DELAYFAIL} exception occurs (see below) or the executing process becomes delayed.
If the executing process becomes delayed, it becomes a member with a priority of the set of delayed processes attached to each of the specified event locations. The priority is the one specified, if any, otherwise 0 (lowest).

If the delayed process becomes re-activated by another process executing a continue action on an event location, the corresponding action statement list is entered. If several delay alternatives specify the same event location, the choice between them is not specified. Prior to entering, if an instance location is specified, the instance value identifying the process that has executed the continue action is stored in it.

dynamic properties: A process executing a delay case action becomes timeoutable when it reaches the point of execution where it may become delayed. It ceases to be timeoutable when it leaves that point.

static conditions: The mode of the instance location must not have the read-only property. The integer literal expression in priority must not deliver a negative value.

dynamic conditions: The DELAYFAIL exception occurs if any event location has a mode with an event length attached which is equal to the number of processes already delayed on that event location.

The lifetime of none of the event locations must end while the executing process is delayed on them.

The SPACEFAIL exception occurs if storage requirements cannot be satisfied.

examples:

14.26  DELAY CASE
    (operator_is_ready): /* some actions */
    (switch_is_closed):
        DO FOR i IN INT (1:100):
            CONTINUE operator_is_ready;
        /* empty the queue */
    OD;
    ESAC

6.18  Send action
6.18.1  General

syntax:

<send action> ::= (1)
    <send signal action> (1.1)  
    | <send buffer action> (1.2)

semantics: A send action initiates the transfer of synchronization information from a sending thread. The detailed semantics depend on whether the synchronization object is a signal or a buffer.

6.18.2  Send signal action

syntax:

<send signal action> ::= (1)
    SEND <signal name> [ ( <value> { , <value> }* ) ]
    TO <instance primitive value> [ <priority> ] (1.1)

semantics: A send signal action evaluates, in any order, the list of values, if present, and the instance primitive value.

The signal specified by signal name is composed for transmission from the specified values and a priority. The priority is the one specified, if any, otherwise 0 (lowest).

If the signal name has a process name attached, only processes with that name may receive the signal; if an instance primitive value is specified, only the process identified by the instance primitive value may receive the signal.

If the signal has a non-empty set of delayed processes attached, in which one or more may receive the signal, one of these will be re-activated. If there are several such processes, one will be selected in an implementation defined way. If there are no such processes, the signal becomes pending.

If a process becomes re-activated, it is removed from all sets of delayed processes of which it was a member.
static conditions: The number of value occurrences must be equal to the number of modes of the signal name. The class of each value must be compatible with the corresponding mode of the signal name. No value occurrence may be intra-regional (see 11.2.2). The integer literal expression in priority must not deliver a negative value.

dynamic conditions: The assignment conditions of each value with respect to its corresponding mode of the signal name apply.

The EMPTY exception occurs if the instance primitive value delivers NULL.

The lifetime of the process indicated by the value delivered by the instance primitive value must not have ended at the point of the execution of the send signal action.

The SENDFAIL exception occurs if the signal name has a process name attached which is not the name of the process indicated by the value delivered by the instance primitive value.

examples:

15.78 SEND ready TO received_user

15.86 SEND readout(count) TO user

6.18.3 Send buffer action

syntax:

```
<send buffer action> ::= 
  SEND <buffer location> (<value>) [ <priority>] (1.1)
```

semantics: A send buffer action evaluates the buffer location and the value in any order.

If the buffer location has a non-empty set of delayed processes attached, one of these will be re-activated. If there are several such processes, one will be selected in an implementation defined way. If there are no such processes and the capacity of the buffer location is exceeded, the executing process becomes delayed with a priority. Otherwise the value is stored with a priority. The priority is the one specified, if any, otherwise 0 (lowest). The capacity of the buffer is exceeded if the buffer location has a mode with a buffer length attached which is equal to the number of values already stored in the buffer location.

If the executing process becomes delayed, it becomes a member of the set of delayed sending processes attached to the buffer location. If a process becomes re-activated, it is removed from all sets of delayed processes of which it was a member.

dynamic properties: A process executing a send buffer action becomes timeoutable when it reaches the point of execution where it may become delayed. It ceases to be timeoutable when it leaves that point.

static conditions: The class of the value must be compatible with the buffer element mode of the mode of the buffer location. The value must not be intra-regional (see 11.2.2). The integer literal expression in priority must not deliver a negative value.

dynamic conditions: The assignment conditions of the value with respect to the buffer element mode of the mode of the buffer location apply; the possible exceptions occur before the process may become delayed.

The lifetime of the buffer location must not end while the executing process is delayed on it.

examples:

16.123 SEND user→ ([(ready, →counter_buffer)]) ;

6.19 Receive case action

6.19.1 General

syntax:

```
<receive case action> ::= 
  <receive signal case action> (1.1) 
  | <receive buffer case action> (1.2)
```

semantics: A receive case action receives synchronization information transmitted by a send action. The detailed semantics depend on the synchronization object used, which is either a signal or a buffer. Entering a receive case action does not necessarily result in a delaying of the executing thread (see clause 11 for further details).
6.19.2 Receive signal case action

syntax:

\[
<\text{receive signal case action}> ::= (1)
\]
\[
\text{RECEIVE CASE} [\text{SET} <\text{instance location}> ; ]
\]
\[
\{ <\text{signal receive alternative}> \}^+ [ \text{ELSE} <\text{action statement list}> ] \text{ESAC} (1.1)
\]
\[
| \text{RECEIVE} [\text{SET} <\text{instance location}> ]
\]
\[
( <\text{signal name}> [\text{IN} <\text{location list}> ] ) (1.2)
\]
\[
<\text{location list}> ::= (2)
\]
\[
<\text{location}> \{ , <\text{location}> \}^* (2.1)
\]
\[
<\text{signal receive alternative}> ::= (3)
\]
\[
( <\text{signal name}> [\text{IN} <\text{defining occurrence list}> ] ) : <\text{action statement list}> (3.1)
\]

derived syntax: The notation (1.2) is derived syntax for

\[
\text{RECEIVE CASE} [\text{SET} <\text{instance location}> ; ] (4)
\]
\[
( <\text{signal name}> [\text{IN} <\&name>_1, \ldots, <\&name>_n ] ) : <\text{action statement list}> (4.1)
\]

where \(<\&name>_1, \ldots, <\&name>_n\) are virtually introduced value receive names, and

\(<\text{location}>_1, \ldots, <\text{location}>_n\) are the locations in the location list.

semantics: A receive signal case action evaluates the instance location, if present.

Then the executing process: (immediately) receives a signal or, if ELSE is specified, enters the corresponding action statement list, otherwise becomes delayed. The executing process immediately receives a signal if one of a signal name specified in a signal receive alternative is pending and may be received by the process. If more than one signal may be received, one with the highest priority will be selected in an implementation defined way.

If the executing process becomes delayed, it becomes a member of the set of delayed processes attached to each of the specified signals. If the delayed process becomes re-activated by another process executing a send signal action, it receives a signal.

If the executing process receives a signal, the corresponding action statement list is entered. Prior to entering, if an instance location is specified, the instance value identifying the process that has sent the received signal is stored in it. If the signal name of the received signal has a list of modes attached, a list of value receive names is specified; the signal carries a list of values, and the value receive names denote their corresponding value in the entered action statement list.

static properties: A defining occurrence in the defining occurrence list of a signal receive alternative defines a value receive name. Its class is the M-value class, where M is the corresponding mode in the list of modes attached to the signal name in front of it.

dynamic properties: A process executing a receive signal case action becomes timeoutable when it reaches the point of execution where it may become delayed. It ceases to be timeoutable when it leaves that point.

static conditions: The mode of the instance location must not have the read-only property.

All signal name occurrences must be different.

The optional IN and the defining occurrence list in the signal receive alternative must be specified if and only if the signal name has a non-empty set of modes. The number of names in the defining occurrence list must be equal to the number of modes of the signal name.

The assignment conditions of the values delivered by \(<\&name>_1, \ldots, <\&name>_n\) with respect to the modes of \(<\text{location}>_1, \ldots, <\text{location}>_n\) apply.

dynamic conditions: The SPACEFAIL exception occurs if storage requirements cannot be satisfied.
examples:

15.83 RECEIVE CASE
   (advance): count + := 1;
   (terminate):
      SEND readout(count) TO user;
      EXIT work_loop;
   ESAC

6.19.3 Receive buffer case action

syntax:

<receive buffer case action> ::= (1)
   RECEIVE CASE [ SET <instance location> ; ]
   { <buffer receive alternative> }*
   [ ELSE <action statement list> ]
   ESAC
   | RECEIVE [ SET <instance location> ]
     ( <buffer location> IN <location> )

<buffer receive alternative> ::= (2)
   ( <buffer location> IN <defining occurrence> ) : <action statement list>

derived syntax: The notation (1.2) is derived syntax for
      RECEIVE CASE [ SET <instance location>; ]
      ( <buffer location> IN &<name> ) : <location> := &<name>;
where &<name> is a virtually introduced value receive name.

semantics: A receive buffer case action evaluates, in any order, the instance location, if present, and all buffer locations
specified in a buffer receive alternative.

Then the executing process: (immediately) receives a value or, if ELSE is specified, enters the corresponding action statement list, otherwise becomes delayed. The executing process immediately receives a value if one is stored in, or a sending process delayed on, one of the specified buffer locations. If more than one value may be received, one with the highest priority will be selected in an implementation defined way.

If the executing process becomes delayed, it becomes a member of the set of delayed processes attached to each of the specified buffer locations. If the delayed process becomes re-activated by another process executing a send buffer action, it receives a value.

If the executing process receives a value, the corresponding action statement list is entered. If several buffer receive alternatives specify the same buffer location, the choice between them is not specified. Prior to entering, if an instance location is specified, the instance value identifying the process that has sent the received value is stored in it. The specified value receive name denotes the received value in the entered action statement list.

Another process becomes re-activated if the executing process receives a value from a buffer location, the attached set of delayed sending processes of which is not empty. The re-activated process is one with the highest priority attached, if the received value was stored in the buffer location, otherwise the one sending the received value. In the former case, the value to be sent by the re-activated process is stored in the buffer location (the capacity of which remains exceeded), and if more than one process may be re-activated, one will be selected in an implementation defined way. The re-activated process is removed from the set of delayed sending processes attached to the buffer location.

static properties: A defining occurrence in a buffer receive alternative defines a value receive name. Its class is the M-value class, where M is the buffer element mode of the mode of the buffer location labelling the buffer receive alternative.

dynamic properties: A process executing a receive buffer case action becomes timeoutable when it reaches the point of execution where it may become delayed. It ceases to be timeoutable when it leaves that point.

static conditions: The mode of the instance location must not have the read-only property.

The assignment conditions of the value denoted by &name with respect to the mode of the location apply.

dynamic conditions: The SPACEFAIL exception occurs if storage requirements cannot be satisfied.
The lifetime of none of the buffer locations must end while the executing process is delayed on them.

### 6.20 CHILL built-in routine calls

**syntax:**

\[
<\text{CHILL built-in routine call}> ::= \quad (1) \\
\quad <\text{CHILL simple built-in routine call}> \quad (1.1) \\
\quad | <\text{CHILL location built-in routine call}> \quad (1.2) \\
\quad | <\text{CHILL value built-in routine call}> \quad (1.3)
\]

**predefined names:** The CHILL built-in routine names are predefined as built-in routine names (see 6.7).

**semantics:** A CHILL built-in routine call is either a CHILL simple built-in routine call, which delivers no results (see 6.20.1), a CHILL location built-in routine call, which delivers a location (see 6.20.2), or a CHILL value built-in routine call, which delivers a value (see 6.20.3).

**static properties:** A CHILL built-in routine call is a location built-in routine call if it is a CHILL location built-in routine call; it is a value built-in routine call if it is a CHILL value built-in routine call.

#### 6.20.1 CHILL simple built-in routine calls

**syntax:**

\[
<\text{CHILL simple built-in routine call}> ::= \quad (1) \\
\quad \text{terminate built-in routine call} \quad (1.1) \\
\quad | \quad \text{io simple built-in routine call} \quad (1.2) \\
\quad | \quad \text{timing simple built-in routine call} \quad (1.3)
\]

**semantics:** A CHILL simple built-in routine call is a built-in routine call which delivers neither a value nor a location. The simple built-in routines for input output are defined in clause 7. The simple built-in routines for timing are defined in clause 9.

#### 6.20.2 CHILL location built-in routine calls

**syntax:**

\[
<\text{CHILL location built-in routine call}> ::= \quad (1) \\
\quad \text{io location built-in routine call} \quad (1.1)
\]

**semantics:** A CHILL location built-in routine call is a built-in routine call that delivers a location. The location built-in routines for input output are defined in clause 7.

#### 6.20.3 CHILL value built-in routine calls

**syntax:**

\[
<\text{CHILL value built-in routine call}> ::= \quad (1) \\
\quad \text{NUM} \ (<\text{discrete expression}> ) \quad (1.1) \\
\quad | \quad \text{PRED} \ (<\text{discrete expression}> ) \quad (1.2) \\
\quad | \quad \text{SUCC} \ (<\text{discrete expression}> ) \quad (1.3) \\
\quad | \quad \text{ABS} \ (<\text{numeric expression}> ) \quad (1.4) \\
\quad | \quad \text{CARD} \ (<\text{powerset expression}> ) \quad (1.5) \\
\quad | \quad \text{MAX} \ (<\text{powerset expression}> ) \quad (1.6) \\
\quad | \quad \text{MIN} \ (<\text{powerset expression}> ) \quad (1.7) \\
\quad | \quad \text{SIZE} \ ( \{ \text{<location> } | \text{<mode argument> } \} ) \quad (1.8) \\
\quad | \quad \text{UPPER} \ (<\text{upper lower argument}> ) \quad (1.9) \\
\quad | \quad \text{LOWER} \ (<\text{upper lower argument}> ) \quad (1.10) \\
\quad | \quad \text{LENGTH} \ (<\text{length argument}> ) \quad (1.11) \\
\quad | \quad \text{allocate built-in routine call} \quad (1.12) \\
\quad | \quad \text{io value built-in routine call} \quad (1.13) \\
\quad | \quad \text{time value built-in routine call} \quad (1.14) \\
\quad | \quad \text{SIN} \ (<\text{floating point expression}> ) \quad (1.15) \\
\quad | \quad \text{COS} \ (<\text{floating point expression}> ) \quad (1.16) \\
\quad | \quad \text{TAN} \ (<\text{floating point expression}> ) \quad (1.17)
\]
ISO-IEC 9496 : 2002(E)

| ARCSIN (<floating point expression>) | (1.18) |
| ARCCOS (<floating point expression>) | (1.19) |
| ARCTAN (<floating point expression>) | (1.20) |
| EXP (<floating point expression>) | (1.21) |
| LN (<floating point expression>) | (1.22) |
| LOG (<floating point expression>) | (1.23) |
| SQRT (<floating point expression>) | (1.24) |

<numeric expression> ::= (2)
  <integer expression> (2.1)
  <floating point expression> (2.2)

<mode argument> ::= (3)
  <mode name> (3.1)
  <array mode name> (<expression>) (3.2)
  <string mode name> (<integer expression>) (3.3)
  <variant structure mode name> (<expression list>) (3.4)

<upper lower argument> ::= (4)
  <array location> (4.1)
  <array expression> (4.2)
  <array mode name> (4.3)
  <string location> (4.4)
  <string expression> (4.5)
  <string mode name> (4.6)
  <discrete location> (4.7)
  <discrete expression> (4.8)
  <discrete mode name> (4.9)
  <floating point location> (4.10)
  <floating point expression> (4.11)
  <floating point mode name> (4.12)
  <access location> (4.13)
  <access mode name> (4.14)
  <text location> (4.15)
  <text mode name> (4.16)

<length argument> ::= (5)
  <string location> (5.1)
  <string expression> (5.2)
  <string mode name> (5.3)
  <event location> (5.4)
  <event mode name> (5.5)
  <buffer location> (5.6)
  <buffer mode name> (5.7)
  <text location> (5.8)
  <text mode name> (5.9)

NOTE – If the upper lower argument is an array location, a string location, a discrete location or a floating point location, the syntactic ambiguity is resolved by interpreting upper lower argument as a location rather than an expression or primitive value. If the length argument is a string location, the syntactic ambiguity is resolved by interpreting length argument as a location rather than an expression.

semantics: A CHILL value built-in routine call is a built-in routine call that delivers a value.

NUM delivers an integer value with the same internal representation as the value delivered by its argument.

PRED and SUCCE deliver respectively the next lower and higher discrete value of their argument.

ABS is defined on numeric values, i.e. integer values and floating point values, delivering the corresponding absolute value.

CARD, MAX and MIN are defined on powerset values. CARD delivers the number of element values in its argument.

MAX and MIN deliver respectively the greatest and smallest element value in their argument.
SIZE is defined on referable locations and (possibly dynamic) modes. In the first case, it delivers the number of addressable memory units occupied by that location; in the second case, the number of addressable memory units that a referable location of that mode will occupy. The mode is static if the mode argument is a mode name, otherwise it is a dynamically parameterized version of it, with parameters as specified in the mode argument. In the first case, the location will not be evaluated at run time.

UPPER and LOWER are defined on (possibly dynamic):

- array, string, discrete, floating point, access and text locations, delivering the upper bound and lower bound of the mode of the location;
- array and string expressions, delivering the upper bound and lower bound of the mode of the value’s class;
- strong discrete and floating point expressions, delivering the upper bound and lower bound of the mode of the value’s class;
- array, string, discrete, floating point, access and text mode names, delivering the upper bound and lower bound of the mode.

LENGTH is defined on (possibly dynamic):

- string and text locations and string expressions delivering the actual value of them;
- event locations delivering the event length of the mode of the locations;
- buffer locations delivering the buffer length of the mode of the locations;
- string mode names delivering the string length of the mode;
- text mode names delivering the text length of the mode;
- buffer mode names delivering the buffer length of the mode;
- event mode names delivering the event length of the mode.

SIN delivers the sine of its argument (interpreted in radians).

COS delivers the cosine of its argument (interpreted in radians).

TAN delivers the tangent of its argument (interpreted in radians).

ARCSIN delivers the sin⁻¹ function of its argument in the range –π/2 : π/2.

ARCCOS delivers the cos⁻¹ function of its argument in the range 0 : π.

ARCTAN delivers the tan⁻¹ function of its argument in the range –π/2 : π/2.

EXP delivers the eˣ function, where x is its argument.

LN delivers the natural logarithm of its argument.

LOG delivers the base 10 logarithm of its argument.

SQRt delivers the square root of its argument.

The same rules for the evaluation of the result of built-in routine call with constant arguments as that of constant expression apply (see 5.3.1).

static properties: The class of a NUM built-in routine call is the &INT-derived class. The built-in routine call is constant (literal) if and only if the argument is constant (literal).

The class of a PRED or SUCC built-in routine call is the resulting class of the argument. The built-in routine call is constant (literal) if and only if the argument is constant (literal).

The class of an ABS built-in routine call is the resulting class of the argument. The built-in routine call is constant (literal) if and only if the argument is constant (literal).

The class of a CARD built-in routine call is the &INT-derived class. The built-in routine call is constant if and only if the argument is constant.

The class of a MAX or MIN built-in routine call is the M-value class, where M is the member mode of the mode of the powerset expression. The built-in routine call is constant if and only if the argument is constant.

The class of a SIZE built-in routine call is the &INT-derived class. The built-in routine call is constant if the mode of the argument is static.
ISO-IEC 9496 : 2002(E)

The class of an UPPER and LOWER built-in routine call is:

- the M-value class if upper lower argument is an array location, array expression or array mode name, where M is the index mode of array location, array expression or array mode name, respectively;
- the &INT-derived class if upper lower argument is a string location, string expression or string mode name;
- the M-value class if upper lower argument is a discrete location, discrete expression or discrete mode name, where M is the mode of discrete location, or discrete expression, or discrete mode name, respectively;
- the M-value class if upper lower argument is a floating point location, floating point expression, or floating point mode name, where M is the mode of the floating point location, floating point expression, or floating point mode name, respectively;
- the M-value class if upper lower argument is an access location or access mode name, where M is the index mode of the mode of the access location or access mode name, respectively;
- the M-value class if upper lower argument is a text location or text mode name, where M is the index mode of the mode of the text location or text mode name, respectively.

An UPPER or LOWER built-in routine call is literal if the upper lower argument is an array mode name, a string mode name, a discrete mode name, a floating point mode name, an access mode name, or a text mode name, if the mode of the array location or string location is static, if the array expression or string expression has a static class, or if the upper lower argument is a discrete location, a discrete expression, a floating point location, a floating point expression, an access location, or a text location.

The class of a LENGTH built-in routine call is the &INT-derived class. The built-in routine call is literal if the length argument is a string location with a static mode, a string expression with a static class, an event location, or a buffer location, or if it is a string mode name, an event mode name, a buffer mode name, or a text mode name.

The class of a TAN, EXP, LN, LOG or SQRT built-in routine call is the resulting class of its argument.

The class of SIN, COS, ARCSIN, ARCCOS, ARCTAN is the 1. N-derived class, 2. N-value class if the class of the argument is 1. an N-derived class, 2. an N-value class, where N is a mode constructed as follows:

- for SIN: &RANGE (–1.0 : 1.0, S)
- for COS: &RANGE (–1.0 : 1.0, S)
- for ARCSIN: &RANGE (–π/2 : π/2, S)
- for ARCCOS: &RANGE (0 : π, S)
- for ARCTAN: &RANGE (–π/2 : π/2, S)

where S is the precision of N, and the novelty is that of N.

A SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN, EXP, LN, LOG or SQRT built-in routine call is constant (literal) if and only if the argument is constant (literal).

static conditions: If the argument of a PRED or SUCC built-in routine call is constant, it must not deliver, respectively, the smallest or greatest discrete value defined by the root mode of the class of the argument. The root mode of the discrete expression argument of PRED and SUCC must not be a numbered set mode.

If the argument of a MAX or MIN built-in routine call is constant, it must not deliver the empty powerset value.

The location argument of SIZE must be referable.

The discrete expression and floating point expression as arguments of UPPER and LOWER must be strong.

If the upper lower argument is an access mode name or an access location, the corresponding access mode must have an index mode.

If the upper lower argument is a text mode name or a text location, the corresponding text mode must have an index mode.
The following compatibility requirements hold for a mode argument which is not a single mode name:

- The class of the expression must be compatible with the index mode of the array mode name.
- The variant structure mode name must be parameterizable and there must be as many expressions in the expression list as there are classes in its list of classes and the class of each expression must be compatible with the corresponding class in the list of classes.

**Dynamic conditions:**

PRED and SUCC that are not constant cause the OVERFLOW exception if they are applied to the smallest or greatest discrete value defined by the root mode of the class of the argument.

NUM and CARD that are not constant cause the OVERFLOW exception if the resulting value is outside the set of values defined by &INT.

MAX and MIN cause the EMPTY exception if they are applied to empty powerset values.

ABS that is not constant causes the OVERFLOW exception if the resulting value is outside the bounds defined by the root mode of the class of the argument.

The RANGEFAIL exception occurs if in the mode argument:

- the expression delivers a value which does not belong to the set of values defined by the index mode of the array mode name;
- the integer expression delivers a negative value or a value which is greater than the string length of the string mode name;
- any expression in the expression list for which the corresponding class in the list of classes of the variant structure mode name is an M-value class (i.e. is strong) delivers a value which is outside the set of values defined by M.

ARCSIN and ARCCOS that are not constant cause the OVERFLOW exception if the argument does not lie in the range −1.0 : 1.0.

LN and LOG that are not constant cause the OVERFLOW exception if the argument is not greater than zero.

SQRT that is not constant causes the OVERFLOW exception if the argument is not greater than or equal to zero.

SIN, COS, TAN, ARCSIN, ARCTAN, LN and LOG that are not constant cause the OVERFLOW exception if the resulting value is greater than the upper bound or less than the lower bound of the root mode of the class of the argument. In the case of an exact mathematical resulting value that is greater than the negative upper limit and less than the positive lower limit of the root mode of the argument, and is different from zero, an UNDERFLOW exception occurs.

ARCCOS, EXP and SQRT that are not constant cause the OVERFLOW exception if the resulting value is greater than the upper bound or less than the lower bound of the root mode of the class of the argument. In the case of an exact mathematical resulting value that is greater than zero and less than the positive lower limit of the root mode of the argument, an UNDERFLOW exception occurs.

**Examples:**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN (sieve)</td>
<td>9.12</td>
<td>(1.7)</td>
</tr>
<tr>
<td>PRED (col_1)</td>
<td>11.47</td>
<td>(1.2)</td>
</tr>
<tr>
<td>SUCC (col_1)</td>
<td>11.47</td>
<td>(1.3)</td>
</tr>
</tbody>
</table>

**6.20.4 Dynamic storage handling built-in routines**

**Syntax:**

```
<allocate built-in routine call> ::= (1)
    GETSTACK ( <mode argument> [ , <value> ] |
        ( [ <constructor actual parameter list> ] ) ) (1.1)
    | ALLOCATE ( <mode argument> [ , <value> ] |
        ( [ <constructor actual parameter list> ] ) ) (1.2)

<terminate built-in routine call> ::= (2)
    TERMINATE ( <reference primitive value> ) (2.1)
```

**Semantics:**

GETSTACK and ALLOCATE create a location of the specified mode and deliver a reference value for the created location. GETSTACK creates this location on the stack (see 10.9). A location whose mode is that of the mode
argument is created and a value referring to it is delivered. The created location is initialized with the value of value, if
present; otherwise with the undefined value (see 4.1.2) if the mode argument is not a moreta mode.

If the mode argument is a moreta mode, first all initializations in the components are performed in textual order. If a
(possibly empty) parameter list is specified, the corresponding constructor of the mode argument is applied to the newly
created location. If the mode argument is a task mode, the task belonging to the newly created location is started.

TERMINATE ends the lifetime of the location referred to by the value delivered by reference primitive value. An
implementation might as a consequence release the storage occupied by this location, and if the reference primitive value
is a location which is not read-only, assign the undefined value to the location.

If the reference primitive value refers to a region or a task location L, the following steps are performed sequentially:

a) L is closed. If a location is closed, no more external calls of the public component procedures in L are accepted.
b) The thread executing the TERMINATE waits until L is empty.
c) If the mode of L contains a destructor, that destructor is applied to L.

static properties: The class of a GETSTACK or ALLOCATE built-in routine call is the M-reference class, where M is the
mode of mode argument. M is either the mode name or a parameterized mode constructed as:

&<array mode name> ( <expression> ) or
&<string mode name> ( <integer expression> ) or
&<variant structure mode name> ( <expression list> ),

respectively.

A GETSTACK or ALLOCATE built-in routine call is intra-regional if it is surrounded by a region, otherwise it is extra-
regional.

static conditions: The class of the value, if present, in the GETSTACK and ALLOCATE built-in routine call must be
compatible with the mode of mode argument; this check is dynamic in case the mode of mode argument is a dynamic
mode.

If the mode of mode argument has the read-only property, the second argument must be present.

The value, if present, in the GETSTACK and ALLOCATE built-in routine call, must be regionally safe for the created
location.

dynamic properties: A reference value is an allocated reference value if and only if it is returned by an ALLOCATE
built-in routine call.

dynamic conditions: GETSTACK causes the SPACEFAIL exception if storage requirements cannot be satisfied.

ALLOCATE causes the ALLOCATEFAIL exception if storage requirements cannot be satisfied.

For GETSTACK and ALLOCATE the assignment conditions of the value delivered by value with respect to the mode of
mode argument apply.

TERMINATE causes the EMPTY exception if the reference primitive value delivers the value NULL.

The reference primitive value must deliver an allocated reference value. The lifetime of the referenced location must not
have ended.

7 Input and Output

7.1 I/O reference model

A model is used for the description of the input/output facilities in an implementation independent way; it distinguishes
three states for a given association location: a free state, a file handling state and a data transfer state.
The diagram shows the three states and the possible transitions between the states.

```
free state

ASSOCIATE

file handling state

DISASSOCIATE

connect

data transfer state

DISCONNECT

The diagram shows the three states and the possible transitions between the states.

free state

ASSOCIATE

file handling state

DISASSOCIATE

connect

data transfer state

DISCONNECT

The association location contains no value. No relation to an outside worl object.

The association location contains an association. Operations like create and delete a file, or change its properties.

An access location is connected to the association location. Transfer data to/from a file: a read and write operations.
```

The model assumes that objects, in implementations often referred to as datasets, files or devices, exist in the outside world, i.e. the external environment of a CHILL program. Such an outside world object is called a file in the model. A file can be a physical device, a communication line or just a file in a file management system; in general, a file is an object that can produce and/or consume data.

Manipulating a file in CHILL requires an association; an association is created by the associate operation and it identifies a file. An association has attributes; these attributes describe the properties of a file that is or could be attached to the association.

In the free state, there is no interaction or relation between the CHILL program and outside world objects. The associate operation changes the state of the model from the free state into the file handling state. This operation takes as one argument an association location and an implementation defined denotation for an outside world object for which an association must be created; additional arguments may be used to indicate the kind of association for the object and the initial values for the attributes of the association. A particular association also implies an (implementation dependent) set of operations that may be applied on the file that is attached to that association.

In the file handling state, it is possible to manipulate a file and its properties via an association, provided that the association enables the particular operation; for operations that change the properties of a file, an exclusive association for the file will be necessary in general.

The model assumes associations in general are exclusive, i.e. only one association exists at the same time for a given outside world object. However, implementations may allow the creation of more associations for the same object, provided that the object can be shared among different users (programs) and/or among different associations within the same program. All operations in the file handling state take an association as an argument.

The dissociate operation is used to end an association for an outside world object; this operation causes transition from the file handling state back to the free state.

Transferring data to or from a file is possible only in the data transfer state; transfer operations require an access location to be connected to an association for that file. The connect operation connects an access location to an association and changes the state of the model into the data transfer state. The operation takes an association location and an access
location as arguments; the association location contains an association for the file to, or from, which data can be transferred via the access location. Additional arguments of the connect operation denote for which type of transfer operations the access location must be connected, and to which record the file must be positioned. At most one access location can be connected to an association location at any one time.

The disconnect operation takes an access location as argument and disconnects it from the association it is connected to; it changes the state of the model back to the file handling state.

In the data transfer state, an access location must be used as an argument of a transfer operation; there are two transfer operations provided, namely, a read operation to transfer data from a file to the program and a write operation to transfer data from the program to a file. The transfer operations use the record mode of the access location to transform CHILL values into records of the file, and vice versa.

A file is viewed in the model as an array of values; each element of this array relates to a record of the file. The element mode of this array is determined by the connect operation to be the record mode of the access location being connected. An index value is assigned to each record of the file; this value uniquely identifies each record of the file. In the description of the connect and transfer operations, three special index values will be used, namely, a base index, a current index and a transfer index. The base index is set by the connect operation and remains unchanged until a subsequent connect operation; it is used to calculate the transfer index in transfer operations and the current index in a connect operation. The transfer index denotes the position in the file where a transfer will take place; the current index denotes the record to which the file currently is positioned.

7.2 Association values

7.2.1 General

An association value reflects the properties of a file that is or could be attached to it. A particular association value also implies an (implementation dependent) set of operations on the file that is possibly attached to it.

Association values have no denotation but are contained in locations of association mode; there exists no expression denoting a value of association mode. Association values can only be manipulated by built-in routines that take an association location as parameter.

7.2.2 Attributes of association values

An association value has attributes; the attributes describe the properties of the association and the file that may or could be attached to it.

The following attributes are language defined:

- **existing**: indicating that a (possibly empty) file is attached to the association;
- **readable**: indicating that read operations are possible for the file when it is attached to the association;
- **writeable**: indicating that write operations are possible for the file when it is attached to the association;
- **indexable**: indicating that the file, when it is attached to the association, allows for random access to its records;
- **sequencible**: indicating that the file, when it is attached to the association, allows for sequential access to its records;
- **variable**: indicating that the size of the records of the file, when it is attached to the association, may vary within the file.

These attributes have a boolean value; the attributes are initialized when the association is created and may be updated as a consequence of particular operations on the association. This list comprises the language defined attributes only; implementations may add attributes according to their own needs.

7.3 Access values

7.3.1 General

Access values are contained in locations of access mode. An access location is necessary to transfer data from or to a file in the outside world.
Access values have no denotation but are contained in locations of access mode; there exists no expression denoting a value of access mode. Access values can only be manipulated by built-in routines that take an access location as parameter.

7.3.2 Attributes of access values

Access values have attributes that describe their dynamic properties, the semantics of transfer operations, and the conditions under which exceptions can occur.

CHILL defines the following attributes:

- **usage**: indicating for which transfer operation(s) the access location is connected to an association; the attribute is set by the connect operation.

- **outoffile**: indicating whether or not the transfer index calculated by the last read operation was in the file; the attribute is initialized to *FALSE* by the connect operation and is set by every read operation.

7.4 Built-in routines for input output

7.4.1 General

Language defined built-in routines are defined for operations on association locations and access locations, and for inspecting and changing the attributes of their values.

The built-in routines will be described in the following sections.

**syntax:**

\[
\text{<io value built-in routine call> ::= (1)}
\]

\[
\text{<association attr built-in routine call> (1.1)}
\]

\[
\text{| <isassociated built-in routine call> (1.2)}
\]

\[
\text{| <access attr built-in routine call> (1.3)}
\]

\[
\text{| <readrecord built-in routine call> (1.4)}
\]

\[
\text{| <gettext built-in routine call> (1.5)}
\]

\[
\text{<io simple built-in routine call> ::= (2)}
\]

\[
\text{<dissociate built-in routine call> (2.1)}
\]

\[
\text{| <modification built-in routine call> (2.2)}
\]

\[
\text{| <connect built-in routine call> (2.3)}
\]

\[
\text{| <disconnect built-in routine call> (2.4)}
\]

\[
\text{| <writerecord built-in routine call> (2.5)}
\]

\[
\text{| <text built-in routine call> (2.6)}
\]

\[
\text{| <settext built-in routine call> (2.7)}
\]

\[
\text{<io location built-in routine call> ::= (3)}
\]

\[
\text{<associate built-in routine call> (3.1)}
\]

**static conditions**: A built-in routine parameter in an io built-in routine that is an *association* location, an *access* location or a *text* location must be *referable*.

7.4.2 Associating an outside world object

**syntax:**

\[
\text{<associate built-in routine call> ::= (1)}
\]

\[
\text{ASSOCIATE ( <association location> [ , <associate parameter list> ] ) (1.1)}
\]

\[
\text{<isassociated built-in routine call> ::= (2)}
\]

\[
\text{ISASSOCIATED ( <association location> ) (2.1)}
\]

\[
\text{<associate parameter list> ::= (3)}
\]

\[
\text{| <associate parameter> { , <associate parameter> }* (3.1)}
\]

\[
\text{<associate parameter> ::= (4)}
\]

\[
\text{| <location> (4.1)}
\]

\[
\text{| <value> (4.2)}
\]
**semantics:** ASSOCIATE creates an association to an outside world object. It initializes the association location with the created association. It initializes the attributes of the created association. The association location is also returned as a result of the call. The particular association that is created is determined by the locations and/or values occurring in the associate parameter list; the modes (classes) and the semantics of these locations (values) are implementation defined.

ISASSOCIATED returns TRUE if association location contains an association and FALSE otherwise.

**static properties:** The class of an ISASSOCIATED built-in routine call is the BOOL-derived class. The mode of an ASSOCIATE built-in routine call is the mode of the association location.

The regionality of an ASSOCIATE built-in routine call is that of the association location.

**static conditions:** The mode and the class of each associate parameter is implementation defined.

**dynamic conditions:** ASSOCIATE causes the ASSOCIATEFAIL exception if the association location already contains an association or if the association cannot be created due to implementation defined reasons.

**examples:**

20.21 ASSOCIATE (file_association,"DSK:RECORDS.DAT"); (1.1)

7.4.3 Dissociating an outside world object

**syntax:**

```
<dissociate built-in routine call> ::= (1)
DISSOCIATE ( <association location> ) (1.1)
```

**semantics:** DISSOCIATE terminates an association to an outside world object. An access location that is still connected to the association contained in an association location is disconnected before the association is terminated.

**dynamic conditions:** DISSOCIATE causes the NOTASSOCIATED exception if association location does not contain an association.

**examples:**

22.38 DISSOCIATE (association); (1.1)

7.4.4 Accessing association attributes

**syntax:**

```
<association attr built-in routine call> ::= (1)
EXISTING ( <association location> ) (1.1)
| READABLE ( <association location> ) (1.2)
| WRITEABLE ( <association location> ) (1.3)
| INDEXABLE ( <association location> ) (1.4)
| SEQUENCIBLE ( <association location> ) (1.5)
| VARIABLE ( <association location> ) (1.6)
```

**semantics:** EXISTING, READABLE, WRITEABLE, INDEXABLE, SEQUENCIBLE and VARIABLE return respectively the value of the existing-, readable-, writeable-, indexable-, sequencible- and variable-attribute of the association contained in association location.

**static properties:** The class of an association attr built-in routine call is the BOOL-derived class.

**dynamic conditions:** The association attr built-in routine call causes the NOTASSOCIATED exception if association location does not contain an association.

7.4.5 Modifying association attributes

**syntax:**

```
<modification built-in routine call> ::= (1)
CREATE ( <association location> ) (1.1)
| DELETE ( <association location> ) (1.2)
| MODIFY ( <association location> [ , <modify parameter list> ] ) (1.3)
```
modify parameter list ::= (2)
\ 
\<modify parameter> \{ , <modify parameter> \}* (2.1)
\<modify parameter> ::= (3)
\<value> \| <location> (3.1) (3.2)

semantics: CREATE creates an empty file and attaches it to the association denoted by the association location. The existing-attribute of the indicated association is set to TRUE if the operation succeeds.

DELETE detaches a file from the association denoted by association location and deletes the file. The existing-attribute of the indicated association is set to FALSE if the operation succeeds.

MODIFY provides the means of changing properties of an outside world object for which an association exists and that is denoted by association location; the locations and/or values that occur in modify parameter list describe how the properties must be modified. The modes (classes) and the semantics of these locations (values) are implementation defined.

dynamic conditions: CREATE, DELETE and MODIFY cause the NOTASSOCIATED exception if the association location does not contain an association.

CREATE causes the CREATEFAIL exception if one of the following conditions occurs:

- the existing-attribute of the association is TRUE;
- the creation of the file fails (implementation defined).

DELETE causes the DELETEFAIL exception if one of the following conditions occurs:

- the existing-attribute of the association is FALSE;
- the deletion of the file fails (implementation defined).

MODIFY causes the MODIFYFAIL exception if the properties, defined by modify parameter list cannot or may not be modified; the conditions under which this exception can occur are implementation defined.

examples:
21.39 CREATE (outassoc); (1.1)
21.69 DELETE (curassoc); (1.2)

7.4.6 Connecting an access location

syntax:

<connect built-in routine call> ::= (1)
\ 
CONNECT ( <transfer location> , <association location> ,
\<usage expression> [ , <where expression> [ , <index expression> ] ] ) (1.1)
\<transfer location> ::= (2)
\<access location> \| <text location> (2.1) (2.2)
\<usage expression> ::= (3)
\<expression> (3.1)
\<where expression> ::= (4)
\<expression> (4.1)
\<index expression> ::= (5)
\<expression> (5.1)

predefined names: To control the connect operation, performed by the built-in routine CONNECT, two synnode names are predefined in the language, namely, USAGE and WHERE; their defining modes are SET (READONLY, WRITEONLY, READWRITE) and SET (FIRST, SAME, LAST), respectively.

Values of the mode USAGE indicate for which type of transfer operations the access location must be connected to an association, while values of the mode WHERE indicate how the file that is attached to an association must be positioned by the connect operation.
semantics: CONNECT connects the access location denoted by transfer location to the association that is contained in association location; there must be a file attached to the denoted association; i.e. the association's existing attribute must be TRUE.

The access location denoted by transfer location is the location itself if it is an access location; otherwise the access sub-location of the text location.

The value that is delivered by usage expression indicates for which type of transfer operations the access location must be connected to the file. If the expression delivers READONLY, the connection is prepared for read operations only; if it delivers WRITEONLY, the connection is set up for write operations only; if it delivers READWRITE, the connection is prepared for both read and write operations.

The indexable attribute of the denoted association must be TRUE if the access location has an index mode, while the sequencible attribute must be TRUE if the location has no index mode.

CONNECT (re)positions the file that is attached to the denoted association; i.e. it establishes a (new) base index and current index in the file. The (new) base index depends upon the value that is delivered by where expression:

- if where expression delivers FIRST or is not specified, the base index is set to 0; i.e. the file is positioned before the first record;
- if where expression delivers SAME, the base index is set to the current index in the file; i.e. the file position is not changed;
- if where expression delivers LAST, the base index is set to N, where N denotes the number of records in the file; i.e. the file is positioned after the last record.

After a base index is set, a current index will be established by CONNECT. This current index depends upon the optional specification of an index expression:

- if no index expression is specified, the current index is set to the (new) base index;
- if an index expression is specified, the current index is set to
  \[
  \text{base index} + \text{NUM}(v) - \text{NUM}(l)
  \]
  where l denotes the lower bound of the access location's index mode and v denotes the value that is delivered by index expression.

If the access location is being connected for sequential write operations (i.e. the access location has no index mode and the usage expression delivers WRITEONLY), then those records in the file that have an index greater than the (new) current index will be removed from the file; i.e. the file may be truncated or emptied by CONNECT.

An access location that has no index mode cannot be connected to an association for read and write operations at the same time.

Any access location to which the denoted association may be connected will be disconnected implicitly before the association is connected to the location that is denoted by transfer location.

CONNECT initializes the outoffile attribute of the access location to FALSE and sets the usage attribute according to the value that is delivered by usage expression.

static properties: The mode attached to a transfer location is the mode of the access location or the access mode of the text location, respectively.

static conditions: The mode of transfer location must have an index mode if an index expression is specified; the class of the value delivered by index expression must be compatible with that index mode. The transfer location must have the same regionality as the association location.

The class of the value delivered by usage expression must be compatible with the USAGE-derived class.

The class of the value delivered by where expression must be compatible with the WHERE-derived class.

dynamic conditions: CONNECT causes the NOTASSOCIATED exception if association location does not contain an association.
CONNECT causes the CONNECTFAIL exception if one of the following conditions occurs:

- the association's existing-attribute is FALSE;
- the association's readable-attribute is FALSE and usage expression delivers READONLY or READWRITE;
- the association's writable-attribute is FALSE and usage expression delivers WRITEONLY or READWRITE;
- the association's indexable-attribute is FALSE and access location has an index mode;
- the association's sequencible-attribute is FALSE and access location has no index mode;
- where expression delivers SAME, while the association contained in association location is not connected to an access location;
- the association's variable-attribute is FALSE and the access location has a dynamic record mode, while usage expression delivers WRITEONLY or READWRITE;
- the association's variable-attribute is TRUE and the access location has a static record mode, while usage expression delivers READONLY or READWRITE;
- the access location has no index mode, while usage expression delivers READWRITE;
- the association contained in association location cannot be connected to the access location, due to implementation defined conditions.

CONNECT causes the RANGEFAIL exception if the index mode of access location is a discrete range mode and the index expression delivers a value which lies outside the bounds of that discrete range mode.

The EMPTY exception occurs if the access reference of the text location delivers the value NULL.

examples:

20.22 CONNECT (record_file, file_association, READWRITE); (1.1)
20.22 READWRITE (3.1)

7.4.7 Disconnecting an access location

syntax:

<disconnect built-in routine call> ::=  
DISCONNECT ( <transfer location> ) (1.1)

semantics: DISCONNECT disconnects the access location denoted by transfer location from the association it is connected to.

dynamic conditions: DISCONNECT causes the NOTCONNECTED exception if the access location denoted by transfer location is not connected to an association.

7.4.8 Accessing attributes of access locations

syntax:

<access attr built-in routine call> ::=  
GETASSOCIATION ( <transfer location> ) (1.1)  
| GETUSAGE ( <transfer location> ) (1.2)  
| OUTOFFILE ( <transfer location> ) (1.3)

semantics: GETASSOCIATION returns a reference value to the association location that the access location denoted by transfer location is connected to; it returns NULL if the access location is not connected to an association.

GETUSAGE returns the value of the usage-attribute; i.e. READONLY (WRITEONLY) if the access location is connected only for read (write) operations, or READWRITE if the access location is connected for both read and write operations.

OUTOFFILE returns the value of the outoffile-attribute of access location; i.e. TRUE if the last read operation calculated a transfer index that was not in the file, FALSE otherwise.

static properties: The class of a GETASSOCIATION built-in routine call is the ASSOCIATION-reference class. The regionality of an GETASSOCIATION built-in routine call is that of the transfer location.
The class of an OUTOFFILE built-in routine call is the BOOL-derived class.

The class of a GETUSAGE built-in routine call is the USAGE-derived class.

**dynamic conditions:** GETUSAGE and OUTOFFILE cause the NOTCONNECTED exception if the access location is not connected to an association.

**examples:**

21.47 OUTOFFILE (infiles (FALSE))

(1.3)

### 7.4.9 Data transfer operations

**syntax:**

```plaintext
<readrecord built-in routine call> ::= (1)
   READRECORD ( <access location> [ , <index expression> ]
   [ , <store location> ] ) (1.1)

<writererecord built-in routine call> ::= (2)
   WRITERECORD ( <access location> [ , <index expression> ] ,
   <write expression> ) (2.1)

<store location> ::= (3)
   <static mode location>

<write expression> ::= (4)
   <expression> (4.1)
```

**NOTE** – If the access location has an index mode, the syntactic ambiguity is resolved by interpreting the second argument as an index expression rather than a store location.

**semantics:** For the transfer of data to or from a file, the built-in routines WRITERECORD and READRECORD are defined. The access location must have a record mode, and it must be connected to an association in order to transfer data to or from the file that is attached to that association. The transfer direction must not be in contradiction with the value of the access location's usage-attribute.

Before a transfer takes place, the transfer index, i.e. the position in the file of the record to be transferred, is calculated. If the access location has no index mode, the transfer index is the current index incremented by 1; if the access location has an index mode, the transfer index is calculated as follows:

\[
\text{transfer index} := \text{base index} + \text{NUM}(v) - \text{NUM}(l) + 1
\]

where \(l\) is the lower bound of the mode of the access location's index mode and \(v\) denotes the value that is delivered by index expression. If the transfer of the record with the calculated transfer index has been performed successfully, the current index becomes the transfer index.

The read operation:

READRECORD transfers data from a file in the outside world to the CHILL program.

If the calculated transfer index is not in the file, the outoffile-attribute is set to TRUE; otherwise the file is positioned, the record is read, and the outoffile-attribute is set to FALSE.

The record that is read must not deliver an undefined value; the effect of the read operation is implementation defined if the record being read from the file is not a legal value according to the record mode of the access location.

If a store location is specified, then the value of the record that was read is assigned to this location. If no store location is specified, the value will be assigned to an implicitly created location; the lifetime of this location ends when the access location is disconnected or reconnected. Whether the referenced location is created only once by the connect operation, or every time a read operation is performed, is not defined.

READRECORD returns in both cases a reference value that refers to the (possibly dynamic mode) location to which the value was assigned.

If the outoffile-attribute is set to TRUE as a result of the built-in routine call, then the NULL value is returned as a result of the call.
The write operation:

*WRITERECORD* transfers data from the CHILL program to a file in the outside world. The file is positioned to the record with the calculated index and the record is written.

After the record has been written successfully, the number of records is set to the *transfer* index, if the latter is greater than the actual number of records.

The record written by *WRITERECORD* is the value delivered by *write expression*.

**static properties:** The class of the value that was read by *READRECORD* is the M-value class, where M is the *record* mode of the *access location*, if it has a *static record* mode, or a dynamically parameterized version of it, if the location has a dynamic *record* mode; the parameters of such a dynamically parameterized record mode are:

- the dynamic *string length* of the string value that was read in case of a string mode;
- the dynamic *upper bound* of the array value that was read in case of an array mode;
- the list of (tag) values associated with the mode of the structure value that was read in case of a *variant* structure.

The class of the *READRECORD* built-in routine call is the M-reference class if *store location* is not specified, otherwise it is the S-reference class, where S is the mode of the *store location*.

The *regionality* of a *READRECORD* built-in routine call is that of the *store location* if it is specified, otherwise it is that of the *access location*.

**static conditions:** The *access location* must have a *record* mode.

An *index expression* may not be specified if *access location* has no *index* mode and must be specified if *access location* has an *index* mode; the class of the value delivered by *index expression* must be *compatible* with that *index* mode.

The *store location* must be *referable*.

The mode of *store location* must not have the *read-only property*.

If *store location* is specified, then the mode of *store location* must be *equivalent* with the *record* mode of the *access location*, if it has a *static record* mode or a *varying string record* mode, otherwise a dynamically parameterized version of it; the parameters of such a dynamically parameterized mode are those of the value that has been read.

The class of the value delivered by *write expression* must be *compatible* with the *record* mode of the *access location*, if it has a *static record* mode or a *varying string record* mode; otherwise there should exist a dynamically parameterized version of *record* mode that is *compatible* with the class of *write expression*. The assignment conditions of the value of *write expression* with respect to the above mentioned mode apply.

**dynamic conditions:** The *RANGEFAIL* or *TAGFAIL* exceptions occur if the dynamic part of the above mentioned compatibility check fails.

The *READRECORD* and *WRITERECORD* built-in routine call cause the *NOTCONNECTED* exception if the *access location* is not connected to an association.

The *READRECORD* or *WRITERECORD* built-in routine call cause the *RANGEFAIL* exception if the *index* mode of *access location* is a discrete range mode and the *index expression* delivers a value that lies outside the bounds of that discrete range mode.

The *READRECORD* built-in routine call causes the *READFAIL* exception if one of the following conditions occurs:

- the value of the *usage*-attribute is *WRITEONLY*;
- the value of the *outoffile*-attribute is *TRUE* and the *access location* is connected for sequential read operations;
- the reading of the record with the calculated index fails, due to outside world conditions.

The *WRITERECORD* built-in routine call causes the *WRITEFAIL* exception if one of the following conditions occurs:

- the value of the *usage*-attribute is *READONLY*;
- the writing of the record with the calculated index fails, due to outside world conditions.
If the `RANGEFAIL` exception or the `NOTCONNECTED` exception occur then it occurs before the value of any attribute is changed and before the file is positioned.

**examples:**

20.24  `READRECORD (record_file, curindex, record_buffer);`  (1.1)
22.25  `READRECORD (fileaccess);`  (1.1)
20.32  `WRITERECORD (record_file, curindex, record_buffer);`  (2.1)
21.61  `WRITERECORD (outfile, buffers(flag));`  (2.1)
20.24  `record_buffer`  (3.1)
21.61  `buffers(flag)`  (4.1)

### 7.5 Text input output

#### 7.5.1 General

Text output operations allow the representation of CHILL values in a human-readable form; text input operations perform the opposite transformation.

Text transfer operations are defined on top of the basic CHILL input/output model and operate on files that may be accessed either sequentially or randomly and whose records may have a fixed or variable length.

The model assumes that every record has a (possibly empty) positioning information attached, in implementations often referred to as carriage control or control characters.

Manipulating a text file in CHILL requires an association; transferring data to or from a text file requires a text location to be connected to an association for that file.

Text transfer operations can be applied to CHILL values that may become records of some text file, as well as to CHILL locations that are not necessarily related to any i/o activity of the program.

The possibility to recover from a piece of text the same CHILL values that originated it cannot be guaranteed in general, but rather it depends on the specific representation that has been used.

Text values are contained in locations of text mode. A text location is necessary to transfer data in human-readable form.

Text values have no denotation but are contained in locations of text mode; there exists no expression denoting a value of text mode. Text values can only be manipulated by built-in routines that take a text location as parameter.

#### 7.5.2 Attributes of text values

Text values have attributes that describe their dynamic properties. The following attributes are defined:

- **actual index**: indicating the next character position of the text record to be read or written. It has a mode which is `RANGE (0:L–1)`, where `L` is the text length of the value's mode. It is initialized to 0 when a text location is created.

- **text record reference**: indicating a reference value to the text record sub-location of the text location. It has a mode which is `REF M`, where `M` is the text record mode of the value's mode.

- **access reference**: indicating a reference value to the access sub-location of the text location. It has a mode which is `REF M`, where `M` is the access mode of the value's mode.

#### 7.5.3 Text transfer operations

**syntax:**

```
<text built-in routine call> ::=  (1)
  READTEXT ( <text io argument list> )  (1.1)
  | WRITETEXT ( <text io argument list> )  (1.2)

<text io argument list> ::=  (2)
  <text argument> [ , <index expression> ] ,
  <format argument> [ , <io list> ]  (2.1)
```
<text argument> ::= 
  <text location> 
  | <character string location> 
  | <character string expression> 

<format argument> ::= 
  <character string expression> 

<io list> ::= 
  <io list element> { , <io list element> }* 

<io list element> ::= 
  <value argument> 
  | <location argument> 

<location argument> ::= 
  <discrete location> 
  | <floating point location> 
  | <string location> 

/value argument> ::= 
  <discrete expression> 
  | <floating point expression> 
  | <string expression> 

NOTE – If the io list element is a location, the syntactic ambiguity is resolved by interpreting the io list element as a location argument rather than a value argument.

semantics: READTEXT applies the conversion, editing and i/o control functions contained in the format argument to the text record denoted by the text argument; this (possibly) produces a list of values that are assigned to the elements of the io list in the sequence in which they are specified. WRITETEXT performs the opposite operation. No implicit i/o operations are performed.

If the text argument is a character string location or a character string expression, then the conversion and editing functions are applied without any relation with the external world. In this case the actual index denotes a location that is implicitly created at the beginning of the built-in routine call and initialized to 0. The text record is the character string denoted by character string location or character string expression and the text length its string length.

The elements of the io list may be either:

- value arguments and location arguments, or
- variable clause widths as described below.

Relationships between a format argument and an io list

The value delivered by a format argument must have the form of a format control string (see 7.5.4).

During the execution of a text i/o built-in routine call the format control string (see 7.5.4) denoted by the format argument and the io list are scanned from left to right. Each occurrence of a format text and format specification is interpreted and the appropriate action is taken as follows:

a) format text

In READTEXT the text record should contain at the actual index position a string slice which is equal to the string delivered by format text. In WRITETEXT, the string delivered by format text is transferred to the text record. The semantics are the same as if a format specification which is %C and an io list element that delivers the same string value as that delivered by format text were encountered.

b) format specification

If the format specification contains a repetition factor, then it is equivalent to a sequence of as many format element occurrences as the number denoted by repetition factor.

If the format specification is a format clause, then it contains a control code. If the control code is a conversion clause, then an io list element is taken from the io list and the conversion function selected by the conversion code, conversion qualifiers and clause width is applied to it (see 7.5.5). If the control code is an editing clause or an io clause, then the editing or io function selected by the editing code or io code and clause width is applied to the text argument without reference to the io list (see 7.5.6 and 7.5.7).
If the clause width is variable, then a value is taken from the list, which denotes the width parameter of the conversion or editing function.

If the format specification is a parenthesized clause, then the format control string that is contained in it is scanned.

The interpretation of the format control string terminates when the end of the string delivered by format control string has been reached.

The io list elements of the io list are scanned in the order that they are specified.

**Static conditions:** If the text argument is a string location, its mode must be a varying string mode.

An index expression may not be specified if the text argument is not a text location or if it is and its access mode has no index mode and must be specified if the access mode has an index mode; the class of the value delivered by index expression must be compatible with that index mode.

A text argument in a WRITETEXT built-in routine call must be a location.

A string location in a text argument must be referable.

**Dynamic conditions:** The TEXTFAIL exception occurs if:

- the string value delivered by the format argument cannot be derived as a terminal production of the format control string; or
- an attempt to assign to the actual index a value which is less than 0 or greater than text length is made; or
- during the interpretation, the end of the format control string has been reached and the io list is not completely scanned, or no more elements can be taken from the io list and the format control string contains more conversion codes or variable clause widths; or
- an io clause is encountered and the text argument is not a text location; or
- a format text is encountered in READTEXT and the text record does not contain at the actual index position a string which is equal to the string delivered by format text.

Any exception defined for the READRECORD and WRITERECORD built-in routine call can occur if an i/o control function is executed and any one of the dynamic conditions defined is violated.

**Examples:**

26.18 WRITETEXT (output, "%B%",10) (1.2)

7.5.4 Format control string

**Syntax:**

```
<format control string> ::= 
  [ <format text> ] { <format specification> [ <format text> ] }* 
<format text> ::= 
  { <non-percent character> | <percent> }* 
<percent> ::= 
  % % 
<format specification> ::= 
  % [ <repetition factor> ] <format element> 
<repetition factor> ::= 
  { <digit> }+ 
<format element> ::= 
  <format clause> 
  | <parenthesized clause> 
<format clause> ::= 
  <control code> [ % . ]
```

ISO-IEC 9496 : 2002(E)

ITU-T Rec. Z.200 (1999 E)
NOTE – A format specification is terminated by the first character that cannot be part of the format element. Spaces and format effectors may not be used within format elements. A period (.) may be used to terminate a format clause. It belongs to the format clause and it has only a delimiting effect. To represent the character percent (%) within a format text, it has to be written twice (%%).

semantics: A format control string specifies the external form of the values being transferred and the layout of data within the records. A format control string is composed of format text occurrences, which denote fixed parts of the records and of format specification occurrences, which denote the external representations of CHILL values, allowing the editing of the text record or controlling the actual i/o operations.

If a format specification contains a repetition factor and a format clause, then it is equivalent to as many identical format specification occurrences of the format clause as the number delivered by repetition factor. A repetition factor can be 0, in which case the format specification is not considered. E.g. "%%C4" is equivalent to "%%C4%%C4%%C4".

The decimal notation is assumed for the digits in a repetition factor.

A format control string in a parenthesized clause is repeatedly scanned according to the repetition factor. If none is specified, 1 is assumed by default.

examples:

26.20 size = %C%/ (1.1)

7.5.5 Conversion

syntax:

<conversion clause> ::= (8)  
<conversion code> { <conversion qualifier> }* (8.1)  
| <editing clause> (8.2)  
| <io clause> (8.3)

<parenthesized clause> ::= (9)  
( <format control string> % ) (9.1)

A format control string in a parenthesized clause is repeatedly scanned according to the repetition factor. If none is specified, 1 is assumed by default.

examples:

26.20 size = %C%/ (1.1)

7.5.5 Conversion

syntax:

<conversion clause> ::= (1)  
<conversion code> { <conversion qualifier> }* (1.1)  
| <clause width> 

<conversion code> ::= (2)  
B | O | H | C | F (2.1)

<conversion qualifier> ::= (3)  
L | E | P <character> (3.1)

<clause width> ::= (4)  
{ <digit> }* | V [ <fractional width> ] [ <exponent width> ] (4.1)

<fractional width> ::= (5)  
. { <digit> }+ (5.1)

<exponent width> ::= (6)  
. { <digit> }+ (6.1)

derived syntax: A conversion clause in which a clause width is not present is derived syntax for a conversion clause in which a clause width that is 0 is specified.

semantics: A conversion in a READTEXT built-in routine call transforms a string which is an external representation into a CHILL value. A conversion in a WRITETEXT built-in routine call performs the opposite transformation. The conversion code together with the conversion qualifier specify the type of the conversion and the details of the requested operation such as justification, overflow handling and padding.

The external representation is a string whose length usually depends on the value being converted. That string may contain the minimum number of characters that are necessary to represent the CHILL value (free format) or may have a given length (fixed format).
In the fixed format a slice of width size starting from the actual index position is read from or written into the text record according to the justification and padding selected by conversion qualifiers, as follows:

- in READTEXT: all padding characters (to the left or to the right according to the justification), if any, are removed. However, when characters or fixed character strings are being read, the maximum number $N$ of padding characters that are removed is $\text{width} - L$, where $L$ is 1 or string length, respectively. No characters are removed if $N < 0$. The remaining characters are taken as the external representation;

- in WRITETEXT: if the length of the external representation is less than or equal to width, then the characters are justified to the left or to the right in the slice (according to the justification). The unused string elements, if any, are filled with the padding character. Otherwise the string is truncated (on the left if the justification to the right is selected, otherwise on the right), or width "overflow" indicator characters (*) are transferred, if the qualifier $E$ is present. The truncation is applied to the external representation, including the minus sign, the period (.) and the $E$ (scientific representation), if any.

In the free format the following holds:

- in READTEXT: padding characters, if any, are skipped except when a character or a character string is being read and the conversion qualifier $P$ is not specified. Then, the external representation is taken as the longest slice of characters that starts at the actual index and is made of all the subsequent characters that may lexically belong to it as defined below.

- in WRITETEXT: the string delivered by the conversion is inserted starting from the actual index position.

In WRITETEXT the string which is the external representation is transferred to the text record without regard to its actual length. After the transfer, the actual index is automatically advanced to the next available character position and the actual length is set to the maximum value between the actual index and the (old) actual length.

A clause width is constant if it is made of digits. The decimal notation is assumed. Otherwise it is variable.

If the width is zero, then the free format is chosen, otherwise the width is the length of the fixed format.

If the width is too small to contain the string, the appropriate action is taken depending on the conversion qualifier.

In a READTEXT the external representation that is applied is the one defined below for the mode of the location argument.

In a WRITETEXT the external representation that is applied is the one defined below for the mode M of the M-value or M-derived class of the value delivered by the value argument.

Conversion codes

Conversion codes are represented as single letters. The following conversion codes are defined:

- $B$: binary representation.
- $O$: octal representation.
- $H$: hexadecimal representation.
- $C$: conversion: indicates the default external representation of CHILL values, which depends on the mode of the value being converted (see below).
- $F$: scientific representation, i.e. the representation of floating point values with mantissa and exponent.

The external representation depends on the conversion code and the mode of the value being converted.

Conversion qualifiers

Conversion qualifiers are represented as single letters. The following conversion qualifiers are defined:

- $L$: left justification. Right justification is assumed if it is not present. In the free format the qualifier has no effect.


\[ E: \text{ overflow evidence. In } \text{WRITETEXT} \text{ the overflow indication is selected; if the qualifier is not present, then truncation is performed. In } \text{READTEXT} \text{ or in the free format this qualifier has no effect.} \]

\[ P: \text{ padding. The character that follows the qualifier specifies the padding character. If } P \text{ is not present, then the padding character is assumed to be space by default. In } \text{READTEXT} \text{ if the free format is selected, then spaces and HT (Horizontal Tabulation) are considered as the same character for skipping purposes, either when specified after the qualifier or when applied by default.} \]

**External representation**

The external representation of CHILL values is defined as follows:

a) **Integers**

Integer values are lexically represented as one or more digits in a decimal default base without leading zeros and with a leading sign if negative. Underline characters, a leading plus sign and leading zeros are discarded in **READTEXT**. The following conversion codes are available: \( B, \ O, \ C \) and \( H \). The conversion code \( C \) selects the decimal representation. The digits that may belong to the representation are only those that are selected by the conversion code.

b) **Floating point**

Floating point values can be represented in two ways:

- fixed point representation (selected by \( C \) conversion code);
- scientific representation (selected by \( F \) conversion code).

In the fixed point representation, the floating point value is lexically represented by a sequence of one or more digits (integer part) followed by an optional sequence of one or more digits (fractional part) separated from the integer part by a period (.). A leading minus sign is present if the value is negative.

In the scientific representation, the floating point value is represented by mantissa and exponent. The mantissa is lexically represented as a fixed point value with the integer part consisting of only one digit, greater than zero. The exponent is lexically represented by an \( E \) followed by a possible sign and a sequence of one or more digits. For both representations a leading plus sign and zeros are discarded in **READTEXT**.

If \( \text{fractional width} \) is present, the value delivered by \( \text{digits} \) contained in it indicates the length of the fractional part extended with trailing zeros if necessary, otherwise the fractional part contains the minimum number of digits that are necessary to represent it.

If \( \text{exponent width} \) is present, the value delivered by \( \text{digits} \) contained in it indicates the minimum number of digits to use to represent the exponent, including leading zeros if necessary, otherwise a default value of 3 is assumed.

The following conversion codes are available: \( C, F \).

c) **Booleans**

Boolean values are lexically represented as simple name string, that are \( \text{TRUE} \) and \( \text{FALSE} \) [in upper case (e.g. \( \text{TRUE} \)) or lower case (e.g. \( \text{true} \)) depending on the representation chosen by the implementation for the special simple name strings]. The following conversion code is available: \( C \).

d) **Characters**

Character values are lexically represented as strings of length 1. The following conversion code is available: \( C \).

e) **Sets**

Set mode values are lexically represented as simple name strings, that are the set literals. The following conversion code is available: \( C \).

f) **Ranges**

Range values have the same representation as the values of their \texttt{root} mode. However, only the representations of those values defined by the discrete range mode or floating point range mode belong to the set of external representations associated to the discrete range mode or floating point range mode. 

ITU-T Rec. Z.200  (1999 E)  117
g) character strings

Character string values are lexically represented as strings of characters of length \( L \). In \textit{WRITETEXT} \( L \) is the \textbf{actual length}. In \textit{READTEXT} \( L \) is the \textbf{string length} if the string is a \textbf{fixed} string, otherwise it is a \textbf{varying} string and \( L \) is the \textbf{string length}, unless there are less characters available in the (slice of) \textit{text record} at the \textbf{actual index} position, in which case \( L \) is the number of available characters. The following conversion code is available: \( C \).

h) bit strings

Bit string values are lexically represented as strings of binary digits. The same rules as for character strings apply to determine the number of digits. The following conversion code is available: \( C \).

dynamic properties: A \textit{clause width} has a \textbf{width}, which is the value delivered by \textit{digits} or by a value from the \textit{io list} if the \textit{clause width} is \textbf{variable}, otherwise it is zero if none is specified.

dynamic conditions: The \textbf{TEXTFAIL} exception occurs if:

• in \textit{READTEXT}, the \textit{text record} does not contain a string slice starting at the \textbf{actual index} that (after the removal or skipping of padding characters, see above) can be interpreted as an external representation of one of the values of the mode of the current \textit{location argument} (including an attempt to read a non-empty external representation from a \textit{text record} when \textbf{actual index} = \textbf{actual length}); or

• in \textit{WRITETEXT}, a string slice that is the external representation of the current \textit{value argument} can not be transferred to the \textit{text record} starting at the \textbf{actual index}; or

• in \textit{READTEXT} a \textit{conversion code} is encountered and the current element in the \textit{io list} is not a location, or the mode of the location has the \textbf{read-only property}; or

• the same \textit{conversion qualifier} is specified more than once; or

• a \textbf{variable clause width} is encountered and the corresponding \textit{io list element} in the \textit{io list} does not have an integer class or it is less than 0;

• a \textit{clause width} has a \textbf{fractional width} or an \textbf{exponent width} and the corresponding \textit{io list element} in the \textit{io list} does not have a floating point class, or it has an \textbf{exponent width} and the \textit{conversion code} is not \( F \).

derived syntax: An \textit{editing clause} in which a \textit{clause width} is not present is derived syntax for an \textit{editing clause} in which a \textit{clause width} that is 1 is specified if the \textit{editing code} is not \( T \), otherwise 0, respectively.

semantics: The following editing functions are defined:

\begin{itemize}
  \item \textit{X}: space: \textbf{width} space characters are inserted or skipped.
  \item \textit{>}: skip right: the \textbf{actual index} is moved rightward for \textbf{width} positions.
  \item \textit{<}: skip left: the \textbf{actual index} is moved leftward for \textbf{width} positions.
  \item \textit{T}: tabulation: the \textbf{actual index} is moved to the position \textbf{width}.
\end{itemize}

In \textit{WRITETEXT}, if the \textbf{actual index} is moved to a position which is greater than the \textbf{actual length}, then a string of \( N \) space characters, where \( N \) is the difference between the \textbf{actual index} and the (old) \textbf{actual length} is appended to the \textit{text record}. The \textbf{actual length} is set to the maximum value between the \textbf{actual index} and the (old) \textbf{actual length}.
dynamic conditions: The TEXTFAIL exception occurs if:

- the actual index is moved to a position which is less than 0 or greater than text length; or
- in READTEXT the actual index is moved to a position which is greater than the actual length; or
- in READTEXT the editing code $X$ is specified and a string of width space or HT (Horizontal Tabulation) characters is not present in the text record at the actual index position.

examples:

26.22 $X$  \hspace{1cm} (1.1)

7.5.7 I/O control

syntax:

$$<\text{io clause}> ::= \hspace{1cm} (1)$$
$$<\text{io code}> \hspace{1cm} (1.1)$$

$$<\text{io code}> ::= \hspace{1cm} (2)$$
$$/ | - | + | ? | ! | =$$

semantics: The i/o control functions (except %=) perform an i/o operation. They allow precise control over the transfer of the text record. In READTEXT, all the functions have the same effect, to read the next record from the file. In WRITETEXT, the text record and the appropriate representation of the carriage control information are transferred. The initial position of the carriage at the time the text location is connected is such that the first character of the first text record is printed at the beginning of the first unoccupied line (regardless of any positioning information attached to the text record).

The carriage placement is described by means of the following abstract operations on the current column, line and page ($x, y, z$) considering columns as being numbered from zero starting at the left margin, and lines from zero starting at the top margin.

$nl(w)$: the carriage is moved $w$ lines downward, at the beginning of the line (new position: \((0, (y + w) \mod p, z + (y + w)/p)\), where $p$ is the number of lines per page));

$np(w)$: the carriage is moved $w$ pages downward at the beginning of the line (new position: \((0, 0, z + w)\)).

The following control functions are provided:

$/$: next record: the record is printed on the next line ($nl(1)$, print record, $nl(0)$);

$+$: next page: the record is printed on the top of the next page ($np(1)$, print record, $nl(0)$);

$-$: current line: the record is printed on the current line (print record, $nl(0)$);

$?$: prompt: the record is printed on the next line. The carriage is left at the end of the line ($nl(1)$, print record);

$!$: emit: no carriage control is performed (print record);

$=$: end page: defines the positioning of the next record, if any, to be at the top of the next page (this overrides the positioning performed before the printing of the record). It does not cause any i/o operation.

The I/O transfer is performed as follows:

- in READTEXT, the semantics are as if a READRECORD $(A, I, R)$, where $A$ is the access sub-location of the text location, $I$ is the index expression (if any) and $R$ denotes the text record, were executed. After the I/O transfer the actual index is set to 0 and actual length to the string length of the string value that was read;

- in WRITETEXT, the semantics are as if a WRITERECORD $(A, I, R)$, where $A$ is the access sub-location of the text location, $I$ is the index expression (if any) and $R$ denotes the text record, were executed. The associated positioning information is also transferred. If the record mode of the access is not dynamic, then the text record is filled at the end with space characters and its actual length is set to text length before the transfer takes place. After the I/O transfer the actual index and actual length are set to 0.

examples:

26.21 $/$  \hspace{1cm} (1.1)
7.5.8 Accessing the attributes of a text location

syntax:

\[<\text{gettext built-in routine call}> ::=\]

\[\text{GETTEXTRECORD} ( \text{<text location>} )\]

\[\text{GETTEXTINDEX} ( \text{<text location>} )\]

\[\text{GETTEXTACCESS} ( \text{<text location>} )\]

\[\text{EOLN} ( \text{<text location>} )\]

\[<\text{settext built-in routine call}> ::=\]

\[\text{SETTEXTRECORD} ( \text{<text location>} , \text{<character string location>} )\]

\[\text{SETTEXTINDEX} ( \text{<text location>} , \text{<integer expression>} )\]

\[\text{SETTEXTACCESS} ( \text{<text location>} , \text{<access location>} )\]

semantics: \text{GETTEXTRECORD} returns the text record reference of \text{<text location>}. \text{GETTEXTINDEX} returns the actual index of \text{<text location>}. \text{GETTEXTACCESS} returns the access reference of \text{<text location>}. EOLN delivers TRUE if no more characters are available in the text record (i.e. if the actual index equals the actual length).

\text{SETTEXTRECORD} stores a reference to the location delivered by \text{character string location} into the text record reference of the \text{<text location>}. \text{SETTEXTINDEX} has the same semantics as an editing clause in \text{WRITETEXT} in which editing code is T and clause width delivers the same value as \text{integer expression}, applied to the text record denoted by \text{<text location>}. \text{SETTEXTACCESS} stores a reference to the location delivered by \text{access location} into the access reference of the \text{<text location>}. static properties: The class of the \text{GETTEXTRECORD} built-in routine call is the M-reference class, where M is the text record mode of the \text{<text location>}. The class of the \text{GETTEXTINDEX} built-in routine call is the \&INT-derived class. The class of the \text{GETTEXTACCESS} built-in routine call is the M-reference class, where M is the access mode of the \text{<text location>}. The class of the \text{EOLN} built-in routine call is the \text{BOOL}-derived class.

A \text{GETTEXTRECORD} or \text{GETTEXTACCESS} built-in routine call has the same regionality as the \text{<text location>}. static conditions: The mode of the \text{character string location} argument of \text{SETTEXTRECORD} must be read-compatible with the text record mode of the \text{<text location>}. The mode of the \text{access location} argument of \text{SETTEXTACCESS} must be read-compatible with the access mode of the \text{<text location>}. The \text{location} argument in \text{SETTEXTRECORD} and \text{SETTEXTACCESS} must have the same regionality as the \text{<text location>}. dynamic conditions: The \text{TEXTFAIL} exception occurs if the \text{integer expression} argument of \text{SETTEXTINDEX} delivers a value that is less than 0 or greater than the text length of the \text{<text location>}. examples:

26.23 GETTEXTINDEX (output) (1.2)

8 Exception handling

8.1 General

An exception is either a language defined exception, in which case it has a language defined exception name, a user defined exception, or an implementation defined exception. A language defined exception will be caused by the dynamic violation of a dynamic condition. Any exception can be caused by the execution of a cause action.
When an exception is caused, it may be handled, i.e. an action statement list of an appropriate handler will be executed.

Exception handling is defined such that at any statement it is statically known which exceptions might occur (i.e. it is statically known which exceptions cannot occur) and for which exceptions an appropriate handler can be found or which exceptions may be passed to the calling point of a procedure. If an exception occurs and no handler for it can be found, the program is in error.

When an exception occurs at an action statement or a declaration statement, the execution of the statement is performed up to an unspecified extent, unless stated otherwise in the appropriate section.

8.2 Handlers

syntax:

\[
\langle\text{handler}\rangle ::= \\
\quad \text{ON} \{ \langle\text{on-alternative}\rangle \} \ast \ [ \text{ELSE} \ \langle\text{action statement list}\rangle ] \ \text{END} (1) \\
\langle\text{on-alternative}\rangle ::= \\
\quad \langle\text{exception list}\rangle : \langle\text{action statement list}\rangle (2)
\]

semantics: A handler is entered if it is appropriate for an exception E according to 8.3. If E is mentioned in an exception list in an on-alternative in the handler, the corresponding action statement list is entered; otherwise ELSE is specified and the corresponding action statement list is entered.

When the end of the chosen action statement list is reached, the handler and the construct to which the handler is appended are terminated.

static conditions: All the exception names in all the exception list occurrences must be different.

dynamic conditions: The SPACEFAIL exception occurs if an action statement list is entered and storage requirements cannot be satisfied.

examples:

10.47 ON

\[(\text{ALLOCATEFAIL}): \text{CAUSE} \text{overflow};\] END (1.1)

8.3 Handler identification

When an exception E occurs at an action or module A, or a data statement or region D, the exception may be handled by an appropriate handler; i.e. an action statement list in the handler will be executed or the exception may be passed to the calling point of a procedure; or, if neither is possible, the program is in error.

For any action or module A, or data statement or region D, it can be statically determined whether for a given exception E at A or D an appropriate handler can be found or whether the exception may be passed to the calling point.

An appropriate handler for A or D with respect to an exception with exception name E is determined as follows:

1) if a handler which mentions E in an exception list or which specifies ELSE is appended to or included in A or D, and E occurs in the reach directly enclosing the handler, then that handler is the appropriate one with respect to E;

2) otherwise, if A or D is directly enclosed by a bracketed action, a module or a region, the appropriate handler (if present) is the appropriate handler for the bracketed action, module or region with respect to E;

3) otherwise, if A or D is placed in the reach of a procedure definition then:

- if a handler which mentions E in an exception list or specifies ELSE is appended to the procedure definition, then that handler is the appropriate handler;
- otherwise, if E is mentioned in the exception list of the procedure definition, then E is caused at the calling point;
- otherwise there is no user-defined handler; however, in this situation an implementation defined handler may be appropriate (see 13.5);
4) otherwise, if A or D is placed in the reach of a process definition, then:

- if a handler which mentions E in an exception list or specifies ELSE is appended to the process definition, then that handler is the appropriate handler;
- otherwise there is no user-defined handler; however, in this situation an implementation defined handler may be appropriate (see 13.5);

5) otherwise, if A is an action of an action statement list in a handler, then the appropriate handler is the appropriate handler for the action A' or data statement or region D' with respect to E which the handler is appended to or included in but considered as if that handler were not specified.

If an exception is caused and the transfer of control to the appropriate handler implies exiting from blocks, local storage will be released when exiting from the block.

9 Time supervision

9.1 General

It is assumed that a concept of time exists externally to a CHILL program (system). CHILL does not specify the precise properties of time, but provides mechanisms to enable a program to interact with the external world's view of time.

9.2 Timeoutable processes

The concept of a timeoutable process exists in order to identify the precise points during program execution where a time interrupt may occur, that is, when a time supervision may interfere with the normal execution of a process.

A process becomes timeoutable when it reaches a well-defined point in the execution of certain actions. CHILL defines a process to become timeoutable during the execution of specific actions; an implementation may define a process to become timeoutable during the execution of further actions.

9.3 Timing actions

syntax:

```
<timing action> ::= (1)
    <relative timing action> (1.1)
| <absolute timing action> (1.2)
| <cyclic timing action> (1.3)
```

semantics: A timing action specifies time supervisions of the executing process. A time supervision may be initiated, it may expire and it may cease to exist. Several time supervisions may be associated with a single process because of the cyclic timing action and because a timing action can itself contain other actions whose execution can initiate time supervisions.

A time interrupt occurs when a process is timeoutable and at least one of its associated time supervisions has expired. The occurrence of a time interrupt implies that the first expired time supervision ceases to exist; furthermore, it leads to the transfer of control associated with that time supervision in the supervised process. If the supervised process was delayed, it becomes re-activated.

Time supervisions also cease to exist when control leaves the timing action that initiated them.

NOTE – If the transfer of control causes the process to leave a region, the region will be released (see 11.2.1).

9.3.1 Relative timing action

syntax:

```
<relative timing action> ::= (1)
    AFTER <duration primitive value> [ DELAY ] IN <action statement list> <timing handler> END (1.1)
```

```
<timing handler> ::= (2)
    TIMEOUT <action statement list> (2.1)
```
semantics: The \textit{duration primitive value} is evaluated, a time supervision is initiated, and then the \textit{action statement list} is entered.

If \texttt{DELAY} is specified, the time supervision is initiated when the executing process becomes \texttt{timeoutable} at the point of execution specified by the \textit{action statement} in the \textit{action statement list}, otherwise it is initiated before the \textit{action statement list} is entered.

If \texttt{DELAY} is specified, the time supervision ceases to exist if it has been initiated and the executing process ceases to be \texttt{timeoutable}.

The time supervision expires if it has not ceased to exist when the specified period of time has elapsed since initiation.

The transfer of control associated with the time supervision is to the \textit{action statement list} of the \texttt{timing handler}.

\textbf{static conditions:} If \texttt{DELAY} is specified the \textit{action statement list} must consist of precisely one \textit{action statement} that may itself cause the executing process to become \texttt{timeoutable}.

\textbf{dynamic conditions:} The \texttt{TIMERFAIL} exception occurs if the initiation of the time supervision fails for an implementation defined reason.

\subsection{9.3.2 Absolute timing action}

\textbf{syntax:}

\begin{verbatim}
<absolute timing action> ::= \\
(1) AT <absolute time primitive value> IN \\
<action statement list> <timing handler> END
\end{verbatim}

\textbf{semantics:} The \textit{absolute time primitive value} is evaluated, a time supervision is initiated, and then the \textit{action statement list} is entered.

The time supervision expires if it has not ceased to exist at (or after) the specified point in time.

The transfer of control associated with the time supervision is to the \textit{action statement list} of the \texttt{timing handler}.

\textbf{dynamic condition:} The \texttt{TIMERFAIL} exception occurs if the initiation of the time supervision fails for an implementation defined reason.

\subsection{9.3.3 Cyclic timing action}

\textbf{syntax:}

\begin{verbatim}
<cyclic timing action> ::= \\
(1) CYCLE <duration primitive value> IN \\
<action statement list> END
\end{verbatim}

\textbf{semantics:} The cyclic timing action is intended to ensure that the executing process enters the action statement list at precise intervals without cumulated drifts (this implies that the execution time for the \textit{action statement list} on average should be less than the specified duration value). The \textit{duration primitive value} is evaluated, a relative time supervision is initiated, and then the \textit{action statement list} is entered.

The time supervision expires if it has not ceased to exist when the specified period of time has elapsed since initiation. Indivisibly with the expiration a new time supervision with the same duration value is initiated.

The transfer of control associated with the time supervision is to the beginning of the \textit{action statement list}.

Note that the cyclic timing action can only terminate by a transfer of control out of it.

\textbf{dynamic properties:} The executing process becomes \texttt{timeoutable} if and when control reaches the end of the \textit{action statement list}.

\textbf{dynamic conditions:} The \texttt{TIMERFAIL} exception occurs if any initiation of a time supervision fails for an implementation defined reason.
9.4 Built-in routines for time

**syntax:**

\[
\text{<time value built-in routine call> ::= (1)} \\
| \text{<duration built-in routine call> (1.1)} \\
| \text{<absolute time built-in routine call> (1.2)}
\]

**semantics:** Implementations are likely to have quite different requirements and capabilities in terms of precision and range of time values. The built-in routines defined below are intended to accommodate these differences in a portable manner.

9.4.1 Duration built-in routines

**syntax:**

\[
\text{<duration built-in routine call> ::= (1)} \\
| \text{MILLISECS ( <integer expression> ) (1.1)} \\
| \text{SECS ( <integer expression> ) (1.2)} \\
| \text{MINUTES ( <integer expression> ) (1.3)} \\
| \text{HOURS ( <integer expression> ) (1.4)} \\
| \text{DAYS ( <integer expression> ) (1.5)}
\]

**semantics:** A duration built-in routine call delivers a duration value with implementation defined and possibly varying precision (i.e. MILLISECS (1000) and SECS (1) may deliver different duration values); this value is the closest approximation in the chosen precision to the indicated period of time. The argument of MILLISECS, SECS, MINUTES, HOURS and DAYS indicate a point in time expressed in milliseconds, seconds, minutes, hours and days respectively.

**static properties:** The class of a duration built-in routine call is the DURATION-derived class.

**dynamic conditions:** The RANGEFAIL exception occurs if the implementation cannot deliver a duration value denoting the indicated period of time.

9.4.2 Absolute time built-in routine

**syntax:**

\[
\text{<absolute time built-in routine call> ::= (1)} \\
| \text{ABSTIME ( [ [ [ [ <year expression> , ] <month expression> , ] <day expression> , ] <hour expression> , ] <minute expression> , ] <second expression> ) (1.1)}
\]

\[
\text{<year expression> ::= (2)} \\
| \text{<integer expression> (2.1)}
\]

\[
\text{<month expression> ::= (3)} \\
| \text{<integer expression> (3.1)}
\]

\[
\text{<day expression> ::= (4)} \\
| \text{<integer expression> (4.1)}
\]

\[
\text{<hour expression> ::= (5)} \\
| \text{<integer expression> (5.1)}
\]

\[
\text{<minute expression> ::= (6)} \\
| \text{<integer expression> (6.1)}
\]

\[
\text{<second expression> ::= (7)} \\
| \text{<integer expression> (7.1)}
\]

**semantics:** The ABSTIME built-in routine call delivers an absolute time value denoting the point in time in the Gregorian calendar indicated in the parameter list. The parameters indicate the components of time in the following order: the year, the month, the day, the hour, the minute and the second. When higher order parameters are omitted, the point in time indicated is the next one that matches the low order parameters present (e.g. ABSTIME (15,12,00,00) denotes noon on the 15th in this or the next month).
When no parameters are specified, an absolute time value denoting the present point in time is delivered.

**static properties**: The class of the absolute time built-in routine call is the *TIME*-derived class.

**dynamic conditions**: The *RANGEFAIL* exception is caused if the implementation cannot deliver an absolute time value denoting the indicated point in time.

### 9.4.3 Timing built-in routine call

**syntax**:

\[
\langle \text{timing simple built-in routine call} \rangle ::= \\
(1) \quad \text{WAIT } ( ) \\
(1.1) \quad | \quad \text{EXPIRED } ( ) \\
(1.2) \quad | \quad \text{INTTIME} (<\text{absolute time primitive value}> , [ [ [ <\text{year location}> \\
(1.3) \quad <\text{month location}> ] <\text{day location}>, ] \\
(1.3) \quad <\text{hour location}>, ] <\text{minute location}>, ] \\
(1.3) \quad <\text{second location}> )
\]

\[
<\text{year location}> ::= \\
(2) \quad \langle \text{integer location} \rangle \\
(2.1)
\]

\[
<\text{month location}> ::= \\
(3) \quad \langle \text{integer location} \rangle \\
(3.1)
\]

\[
<\text{day location}> ::= \\
(4) \quad \langle \text{integer location} \rangle \\
(4.1)
\]

\[
<\text{hour location}> ::= \\
(5) \quad \langle \text{integer location} \rangle \\
(5.1)
\]

\[
<\text{minute location}> ::= \\
(6) \quad \langle \text{integer location} \rangle \\
(6.1)
\]

\[
<\text{second location}> ::= \\
(7) \quad \langle \text{integer location} \rangle \\
(7.1)
\]

**semantics**: *WAIT* unconditionally makes the executing process **timeoutable**: its execution can only terminate by a time interrupt. (Note that the process remains active in the CHILL sense.)

*EXPIRED* makes the executing process **timeoutable** if one of its associated time supervisions has expired; otherwise it has no effect.

*INTTIME* assigns to the specified integer locations an integer representation of the point in time in the Gregorian calendar specified by the *absolute time primitive value*. The locations passed as arguments receive the components of time in the following order: the year, the month, the day, the hour, the minute and the second.

**static conditions**: All specified integer locations must be **referable** and their modes may not have the **read-only** property.

**dynamic properties**: *WAIT* makes the executing process **timeoutable**.

*EXPIRED* makes the executing process **timeoutable** if there is an expired time supervision associated with it.

### 10 Program Structure

#### 10.1 General

The *if action, case action, do action, delay case action, begin-end block, module, region, spec module, spec region, context, receive case action, procedure definition* and *process definition* determine the program structure; i.e. they determine the scope of names and the lifetime of locations created in them.
The word block is used to denote:
- the action statement list in a do action including any loop counter and while control;
- the action statement list in a then clause in an if action;
- the action statement list in a case alternative in a case action;
- the action statement list in a delay alternative in a delay case action;
- a begin-end block;
- a procedure definition excluding the result spec and parameter spec of all formal parameters of the formal parameter list;
- a process definition excluding the parameter spec of all formal parameters of the formal parameter list;
- the action statement list in a buffer receive alternative or in a signal receive alternative, including any defining occurrences in a defining occurrence list after IN;
- the action statement list after ELSE in an if action or case action or a receive case action or handler;
- the on-alternative in a handler;
- the action statement list in a relative timing action, an absolute timing action, a cyclic timing action or in a timing handler.

The word modulion is used to denote:
- a module or region, excluding the context list and defining occurrence, if any;
- a spec module or spec region, excluding the context list, if any;
- a context;
- the specification together with the corresponding body of a moreta mode;
- a template together with the corresponding body.

The word group denotes either a block or a modulion.

The word reach or reach of a group denotes that part of the group that is not surrounded (see 10.2) by an inner group. If BM is a moreta mode and DM is a direct successor of BM then BM P -BMCD ∪ DMP form one reach. For the visibility of the internal components of moreta modes the reach of a successor is nested immediately in the specification part of its direct predecessor; this nesting occurs at the end of the specification part.

A group influences the scope of each name created in its reach. Names are created by defining occurrences:

- A defining occurrence in the defining occurrence list of a declaration, mode definition or synonym definition or appearing in a signal definition creates a name in the reach where the declaration, mode definition, synonym definition or signal definition, respectively, is placed.
- A defining occurrence in a set mode creates a name in the reach directly enclosing the set mode.
- A defining occurrence appearing in the defining occurrence list in a formal parameter list creates a name in the reach of the associated procedure definition or process definition.
- A defining occurrence in front of a colon followed by an action, region, procedure definition, or process definition creates a name in the reach where the action, region, procedure definition, process definition, respectively, is placed.
- A (virtual) defining occurrence introduced by a with part or in a loop counter creates a name in the reach of the block of the associated do action.
- A defining occurrence in the defining occurrence list of a buffer receive alternative or a signal receive alternative creates a name in the reach of the block of the associated buffer receive alternative or signal receive alternative, respectively.
- A (virtual) defining occurrence for a language predefined or an implementation defined name creates a name in the reach of the imaginary outermost process (see 10.8).
The places where a name is used are called applied occurrences of the name. The name binding rules associate a defining occurrence with each applied occurrence of the name (see 12.2.2).

A name has a certain scope, i.e. that part of the program where its definition or declarations can be seen and, as a consequence, where it may be freely used. The name is said to be visible in that part. Locations and procedures have a certain lifetime, i.e. that part of the program where they exist. Blocks determine both visibility of names and the lifetime of the locations created in them. Modulations determine only visibility; the lifetime of locations created in the reach of a modulation will be the same as if they were created in the reach of the first surrounding block. Modulations allow for restricting the visibility of names. For instance, a name created in the reach of a module will not automatically be visible in inner or outer modules, although the lifetime might allow for it.

10.2 Reaches and nesting

syntax:

<begin-end body> ::= (1)
<action statement list> <data statement list> (1.1)

<proc body> ::= (2)
<data statement list> <action statement list> (2.1)

<process body> ::= (3)
<data statement list> <action statement list> (3.1)

<module body> ::= (4)
{ <data statement> | <visibility statement> | <region> | <spec region> }* <action statement list> (4.1)

<region body> ::= (5)
{ <data statement> | <visibility statement> }* (5.1)

<spec module body> ::= (6)
{ <quasi data statement> | <visibility statement> | <spec module> | <spec region> }* (6.1)

<spec region body> ::= (7)
{ <quasi data statement> | <visibility statement> }* (7.1)

<context body> ::= (8)
{ <quasi data statement> | <visibility statement> | <spec module> | <spec region> }* (8.1)

<action statement list> ::= (9)
{ <action statement> }* (9.1)

<data statement list> ::= (10)
{ <data statement> }* (10.1)

<data statement> ::= (11)
<declaration statement> (11.1)
| <definition statement> (11.2)

<definition statement> ::= (12)
<synmode definition statement> (12.1)
| <newmode definition statement> (12.2)
| <synonym definition statement> (12.3)
| <procedure definition statement> (12.4)
| <process definition statement> (12.5)
| <signal definition statement> (12.6)
| <template> (12.7)
| <empty> ; (12.8)

semantics: When a reach of a block is entered, all the lifetime-bound initializations of the locations created when entering the block are performed. Subsequently, the reach-bound initializations in the block reach, the possibly dynamic evaluations in the loc-identity declarations, the reach-bound initializations in the regions and the actions are performed in the order they are textually specified.
When a reach of a modulion is entered, the reach-bound initializations, the possibly dynamic evaluations in the loc-
identity declarations, the reach-bound initializations in the regions and the actions (if the modulion is a module) that are
in the modulion reach are performed in the order they are textually specified.

A data statement, action, module or region, is terminated either by completing it, or by terminating a handler appended to
it.

When a group G is to be terminated first all TASK and REGION locations (RTL), which depend on G (see 12.2.6), are
closed. The termination of G is finished when all those RTL are completed (see 11.6).

When a reach-bound initialization, loc-identity declaration, action, module, region, procedure or process is terminated,
execution is resumed as follows, depending on the statement or the kind of termination:

• if the statement is terminated by completing the execution of a handler, then the execution is resumed with the
  subsequent statement;
• otherwise, if it is an action that implies a transfer of control, the execution is resumed with the statement defined for
  that action (see 6.5, 6.6, 6.8 and 6.9);
• otherwise, if it is a procedure, control is returned to the calling point (see 10.4);
• otherwise, if it is a process, the execution of that process (or the program, if it is the outermost process) ends
  (see 11.1) and execution is (possibly) resumed with another process;
• otherwise control will be given to the subsequent statement.

static properties: Any reach is directly enclosed in zero or more groups as follows:

• If the reach is the reach of a do action, begin-end block, procedure definition, process definition, then it is directly
  enclosed in the group in whose reach the do action, begin-end block, procedure definition or process definition,
  respectively, is placed, and only in that group.
• If the reach is the action statement list of a timing action or timing handler, or one of the action statement lists of an
  if action, case action or delay case action, then it is directly enclosed in the group in whose reach the timing action,
  timing handler, if action, case action or delay case action is placed, and only in that group.
• If the reach is the action statement list, or a buffer receive alternative, or signal receive alternative, or the action
  statement list following ELSE in a receive buffer case action or receive signal case action, then it is directly
  enclosed in the group in whose reach the receive buffer case action or receive signal case action is placed, and only
  in that group.
• If the reach is the action statement list in an on-alternative or the action statement list following ELSE in a handler
  which is not appended to a group, then it is directly enclosed in the group in whose reach the statement to which the
  handler is appended is placed, and only in that group.
• If the reach is an on-alternative or action statement list after ELSE of a handler which is appended to a group, then
  it is directly enclosed in the group to which the handler is appended, and only in that group.
• If the reach is a module, region, spec module or spec region, then it is directly enclosed in the group in whose reach it
  is placed, and also directly enclosed in the context directly in front of the module, region, spec module or spec
  region, if any. This is the only case where a reach has more than one directly enclosing group.
• If the reach is a context, then it is directly enclosed in the context directly in front of it. If there is no such context, it
  has no directly enclosing group.

A reach has directly enclosing reaches that are the reaches of the directly enclosing groups. A statement has a unique
directly enclosing group, namely, the group in which the statement is placed. A reach is said to directly enclose a group
(reach) if and only if the reach is a directly enclosing reach of the group (reach).

A statement (reach) is said to be surrounded by a group if and only if either the group is the directly enclosing group of
the statement (reach) or a directly enclosing reach is surrounded by the group.
A reach is said to be entered when:

- Module reach: the module is executed as an action (e.g. the module is not said to be entered when a goto action transfers control to a label name defined inside the module).
- Begin-end reach: the begin-end block is executed as an action.
- Region reach: the region is encountered (e.g. the region is not said to be entered when one of its critical procedures is called).
- Procedure reach: the procedure is entered via a procedure call.
- Process reach: the process is activated via the evaluation of a start expression.
- Do reach: the do action is executed as an action after the evaluation of the expressions or locations in the control part.
- Buffer-receive alternative reach, signal receive alternative reach: the alternative is executed on reception of a buffer value or signal.
- On-alternative reach: the on-alternative is executed on the cause of an exception.
- Other block reaches: the action statement list is entered.

An action statement list is said to be entered when and only when its first action, if present, receives control from outside the action statement list.

A reach is a **quasi** reach if it is the one of a spec module, spec region or context, otherwise it is a **real** reach.

A defining occurrence is a **quasi** defining occurrence if:

- it is surrounded by a context and not by a module or region; or
- it is surrounded by a simple spec module or a simple spec region; or
- it is not surrounded by one of the above-mentioned groups and it is surrounded by a module spec or a region spec and it is contained in a quasi declaration, a quasi procedure definition statement or a quasi process definition statement,

otherwise it is a **real** defining occurrence.

### 10.3 Begin-end blocks

**syntax:**

\[
\text{<begin-end block> ::= (1)} \\
\text{BEGIN <begin-end body> END (1.1)}
\]

**semantics:** A begin-end block is an action, possibly containing local declarations and definitions. It determines both visibility of locally created names and the lifetimes of locally created locations (see 10.9 and 12.2).

**dynamic conditions:** The SPACEFAIL exception occurs if storage requirements cannot be satisfied.

**examples:** see 15.73-15.90

### 10.4 Procedure specifications and definitions

**syntax:**

\[
\text{<procedure definition statement> ::= (1)} \\
\text{<defining occurrence> : <procedure definition>} \\
\text{[ <handler> ][ <simple name string> ] ; (1.1)} \\
\text{[ <generic procedure instantiation> (1.2)}
\]

\[
\text{<procedure definition> ::= (2)} \\
\text{PROC ([ <formal parameter list> ] [ <result spec> ] \\
[ EXCEPTIONS ( <exception list> ) ] <procedure attribute list> ; \\
<proc body> END (2.1)}
\]
<formal parameter list> ::= (formal parameter> { , <formal parameter> })*  
(3)  
(3.1)

<formal parameter> ::= <defining occurrence list> <parameter spec>  
(4)  
(4.1)

<procedure attribute list> ::= [ <generality> ]  
(5)  
(5.1)

<generality> ::= GENERAL  
SIMPLE  
INLINE  
(6)  
(6.1)  
(6.2)  
(6.3)

<guarded procedure signature statement> ::= <defining occurrence> : <guarded procedure signature > [ <simple name string> ] ;  
(7)  
(7.1)

<guarded procedure signature > ::= PROC ( [ <parameter list> ] ) [ < result spec> ] 
[EXCEPTIONS ( <exception list> )] <guarded procedure attribute list> END  
(8)  
(8.1)

<guarded procedure definition statement> ::= <defining occurrence> : <guarded procedure definition> 
[ [ <handler> ] [ <simple name string> ] ] ;  
(9)  
(9.1)

<guarded procedure definition> ::= PROC ( [<formal parameter list>] ) [result spec] 
[EXCEPTIONS ( <exception list> )] <guarded procedure attribute list> ;  
<prec body> END  
(10)  
(10.1)

<guarded procedure attribute list> ::=  
[ GENERAL ]  
(11)  
(11.1)

| [ SIMPLE ]  
[ <simple component procedure attribute list> ] <assertion part>  
(11.2)  

| [ INLINE ]  
[ <inline component procedure attribute list> ]  
(11.3)

<simple component procedure attribute list> ::=  
<inline component procedure attribute list>  
(12)  
(12.1)

| DESTR  
(12.2)

| INCOMPLETE  
(12.3)

| [ REIMPLEMENT ] [ FINAL ]  
(12.4)

<inline component procedure attribute list> ::=  
CONSTR  
(13)  
(13.1)

| FINAL  
(13.2)

<assertion part> ::=  
[ PRE ( <boolean expression> ) ]  
(14)  
(14.1)

[ POST ( <boolean expression> ) ]

derived syntax: A formal parameter, where defining occurrence list consists of more than one defining occurrence, is derived from several formal parameter occurrences, separated by commas, one for each defining occurrence and each with the same parameter spec. For example, i, j INT LOC is derived from i INT LOC, j INT LOC.

semantics: A procedure definition statement defines a (possibly) parameterized sequence of actions that may be called from different places in the program. The procedure is terminated and control is returned to the calling point either by executing a return action or by reaching the end of the proc body or by terminating a handler appended to the procedure definition (falling through). Different degrees of complexity of procedures may be specified as follows:

a) simple procedures (SIMPLE) are procedures that cannot be manipulated dynamically. They are not treated as values, i.e. they cannot be stored in a procedure location nor can they be passed as parameters to or returned as result from a procedure call;
b) **general** procedures (**GENERAL**) do not have the restrictions of **simple** procedures and may be treated as procedure values;

c) **inline** procedures (**INLINE**) have the same restrictions as **simple** procedures and they are not **recursive**. They have the same semantics as normal procedures, but the compiler may insert the generated object code at the point of invocation rather than generating code for actually calling the procedure.

Only **simple** and **general** procedures are **recursive**.

A guarded procedure definition statement defines a (possibly) parameterized sequence of actions that may be called from different places in the program. The procedure is terminated and control is returned to the calling point either by executing a **return action** or by reaching the end of the **proc body** or by terminating a **handler** appended to the procedure definition (falling through).

When the procedure is defined in a **moreta mode**, it is called a **component procedure**. Different kinds of **simple** and **inline** component procedures defined in moreta modes may be specified as follows:

a) a **constr** component procedure (**CONSTR**) is a constructor which can be used to initialize moreta locations automatically when they are created statically or dynamically;

b) a **destr** component procedure (**DESTR**) is a destructor which can be used to finalize moreta locations when they are destroyed statically or dynamically;

c) an **incomplete** component procedure (**INCOMPLETE**) has only a signature but no body;

d) a **reimplement** component procedure (**REIMPLEMENT**) which is given a new body and possibly new assertions;

e) a **final** component procedure (**FINAL**) is a procedure which cannot be reimplented in a derived moreta mode.

Different kinds of **assertion part** may be specified for **simple** component procedures:

a) a **pre** assertion part (**PRE**) which is checked automatically before the body of the corresponding procedure is executed;

b) a **post** assertion part (**POST**) which is checked automatically after the body of the corresponding procedure has been executed and before the return to the calling point.

Only **simple** (except for **component procedures** with the attributes **constr** or **destr** or with **public** visibility in a **region** mode) and **general** procedures are **recursive**.

A procedure may return a value or it may return a location (indicated by the **LOC** attribute in the result spec).

The **defining occurrence** in front of the procedure definition defines the name of the procedure.

**parameter passing**

There are basically two parameter passing mechanisms: the "pass by value" (**IN**, **OUT** and **INOUT**) and the "pass by location" (**LOC**).

**pass by value**

In pass by value parameter passing, a value is passed as a parameter to the procedure and stored in a local location of the specified parameter mode. The effect is as if, at the beginning of the procedure call, the location declaration:

```
DCL <defining occurrence> <mode> := <actual parameter>;
```

were encountered for the **defining occurrences** of the **formal parameter**. However, the procedure is entered after the actual parameters have been evaluated. Optionally, the keyword **IN** may be specified to indicate pass by value explicitly.

If the attribute **INOUT** is specified, the actual parameter value is obtained from a location and just before returning the current value of the formal parameter is restored in the actual location.

The effect of **OUT** is the same as for **INOUT** with the exception that the initial value of the actual location is not copied into the formal parameter location upon procedure entry; therefore, the formal parameter has an **undefined** initial value. The store-back operation need not be performed if the procedure causes an exception at the calling point.
pass by location

In pass by location parameter passing, a (possibly dynamic mode) location is passed as a parameter to the procedure body. Only referable locations can be passed in this way. The effect is as if at the entry point of the procedure the loc-identity declaration statement:

DCL <defining occurrence> <mode>

LOC [DYNAMIC] := <actual parameter> ;

were encountered for the defining occurrences of the formal parameter. However, the procedure is entered after the actual parameters have been evaluated.

If a value is specified that is not a location, a location containing the specified value will be implicitly created and passed at the point of the call. The lifetime of the created location is the procedure call. The mode of the created location is dynamic if the value has a dynamic class.

result transmission

Both a value and a location may be returned from the procedure. In the first case, a value is specified in any result action, in the latter case, a location (see 6.8). If the attribute NONREF is not given in the result spec, the location must be referable. The returned value or location is determined by the most recently executed result action before returning. If a procedure with a result spec returns without having executed a result action, the procedure returns an undefined value or an undefined location. In this case the procedure call may not be used as a location procedure call (see 4.2.11) nor as a value procedure call (see 5.2.13), but only as a call action (see 6.7).

static properties: A defining occurrence in a procedure definition statement defines a procedure name.

A procedure name has a procedure definition attached that is the procedure definition in the statement in which the procedure name is defined.

A procedure name has the following properties attached, as defined by its procedure definition:

• It has a list of parameter specs that are defined by the parameter spec occurrences in the formal parameter list, each parameter consisting of a mode and possibly a parameter attribute.

• It has possibly a result spec, consisting of a mode and an optional result attribute.

• It has a possibly empty list of exception names, which are the names mentioned in exception list.

• It has a generality that is, if generality is specified, either general or simple or inline, depending on whether GENERAL, SIMPLE or INLINE is specified; otherwise an implementation default specifies general or simple. If the procedure name is defined inside a block or a region, its generality is simple. If a procedure is defined in a moreta mode and has public visibility, its generality is simple or inline.

• It has a recursivity which is recursive. However, if the generality is inline or if the procedure name is critical (see 11.2.1) the recursivity is non-recursive.

• A component procedure has the generality inline if the attribute INLINE is specified. Otherwise it has the generality SIMPLE by default.

A procedure name that is general is a general procedure name. A general procedure name has a procedure mode attached, formed as:

PROC ( [ <parameter list> ] ) [ <result spec> ]

[ EXCEPTIONS ( <exception list> ) ]

where <result spec>, if present, and <exception list> are the same as in its procedure definition and parameter list is the sequence of <parameter spec> occurrences in the formal parameter list, separated by commas.

A name defined in a defining occurrence list in the formal parameter is a location name if and only if the parameter spec in the formal parameter does not contain the LOC attribute. If it does contain the LOC attribute, it is a loc-identity name. Any such a location name or loc-identity name is referable.
A moreta mode component procedure of a moreta mode M has a complete postcondition CPM which is defined as follows:

a) if M has no immediate base mode then CPM = post part;
b) if M has the immediate base mode B then CPM = CPB \( \wedge \) post part, where CPB is the complete postcondition of B.

**static conditions:** If a procedure name is **intra-regional** (see 11.2.2) or is a public procedure of a moreta mode, its procedure definition must not specify **GENERAL**.

If a procedure name is **critical** (see 11.2.1), its definition may not specify **GENERAL**.

If a simple component procedure has any assertion part, the name of the procedure must have **public** visibility.

If a simple or inline guarded component procedure has the attribute **FINAL**, the name of the procedure must not have **private** visibility.

The defining occurrence of a constr component procedure must be the same as that of its attached moreta mode. A constr component procedure must not specify a result spec and must be **non-recursive**.

The defining occurrence of a destr component procedure must be the same as that of its attached moreta mode. A destr component procedure must neither specify a formal parameter list nor a result spec and must be **non-recursive**.

If specified, the simple name string must be equal to the name string of the defining occurrence in front of the procedure definition.

Only if LOC is specified in the parameter spec or result spec may the mode in it have the **non-value property**.

All exception names mentioned in exception list must be different.

If P1 and P2 are component procedures or component processes then P1 matches P2 if and only if:

a) P1 and P2 are of the same kind; and
b) P1 and P2 have the same simple name string; and
c) the formal parameter lists of P1 and P2 are syntactically and semantically equivalent; and
d) the result specs of P1 and P2 are syntactically and semantically equivalent.

If P is a component procedure or a component process then PB corresponds to PS if and only if:

a) PB matches PS; and
b) the exception lists of PS and PB are syntactically and semantically equivalent; and
c) the attribute lists of PS and PB are syntactically and semantically equivalent.

Two procedures P1 and P2 conform to each other if and only if:

a) they both have the same number of parameters and the names of the modes of corresponding parameters conform to each other; and
b) (they both have a result mode and the names of those result modes conform to each other or they both have no result mode).

**examples:**

1.4  
add:

```plaintext
PROC (i,j INT) RETURNS (INT) EXCEPTIONS (OVERFLOW);
RESULT i+j;
END add;

put:

PROC(p RANGE(1:10)) PRE((p > 0) AND (p < 11));
...;
END put;
```

(1.1)  
(10.1)
10.5 Process specifications and definitions

syntax:

\[
<\text{process definition statement}> ::= \quad (1) \\
\langle \text{defining occurrence} \rangle : <\text{process definition}> \\
\text{[ [ <\text{handler}> ] [ <\text{simple name string}> ] ; \quad (1.1) \\
\text{[ <\text{generic process instantiation}> ] ; \quad (1.2) \\
\text{<process definition> } ::= \quad (2) \\
\text{PROCESS ( [ [ <\text{formal parameter list}> ] ) <process body> END \quad (2.1) }
\]

semantics: A process definition statement defines a possibly parameterized sequence of actions that may be started for concurrent execution from different places in the program (see clause 11).

static properties: A defining occurrence in a process definition statement defines a process name.

A process name has the following property attached, as defined by its process definition:

• It has a list of parameter specs that are defined by the parameter spec occurrences in the formal parameter list, each parameter consisting of a mode and possibly a parameter attribute.

static conditions: If specified, the simple name string must be equal to the name string of the defining occurrence in front of the process definition.

A process definition statement must not be surrounded by a region or by a block other than the imaginary outermost process definition (see 10.8).

The parameter attributes in the formal parameter list must not be INOUT nor OUT.

Only if LOC is specified in the parameter spec in a formal parameter in the formal parameter list may the mode in it have the non-value property.

examples:

14.13 \text{PROCESS ();} \\
\text{wait;} \\
\text{PROC (x INT);} \\
\text{/*some wait action*/} \\
\text{END wait;} \\
\text{DO \text{FOR \text{EVER;}}} \\
\text{wait(10 /* seconds */);} \\
\text{CONTINUE \text{operator is ready;}} \\
\text{OD;} \\
\text{END \quad (2.1) }

10.6 Modules

syntax:

\[
<\text{module}> ::= \quad (1) \\
\text{[ <context list> ] [ <defining occurrence> : ]} \\
\text{\text{MODULE [ BODY ] <module body> END \quad (1.1) \\
\text{[ [ <handler> ] [ <simple name string> ] ; \quad (1.2) \\
\text{\text{<remote modulion> \quad (1.3)}} \\
\text{\text{<generic module instantiation>}}}
\]

semantics: A module is an action statement possibly containing local declarations and definitions. A module is a means of restricting the visibility of name strings; it does not influence the lifetime of the locally declared locations.
The detailed visibility rules for modules are given in 12.2.

**static properties:** A *defining occurrence* in a module defines a *module* name as well as a *label* name. The name has the module (seen as a modulion, i.e. excluding the context list and defining occurrence, if any) attached.

A module is developed piecewisely if and only if a context list is specified.

A module is a *module body* if and only if BODY is specified.

**static conditions:** If specified, the *simple name string* must be equal to the name string of the defining occurrence.

A remote modulion in a module must refer to a module.

**examples:**

7.48 MODULE

SEIZE convert;
DCL n INT INIT:= 1979;
DCL rn CHAR(20) INIT:=(20) " ";
GRANT n,rn;
convert();
ASSERT rn = "MDCCCLXXVIIII"!(6) " ";

END

10.7 Regions

**syntax:**

<region> ::= 

[ <context list> ] [ <defining occurrence> : ]
REGION [ BODY ] <region body> END
[ <handler> ] [ <simple name string> ] ;
| <remote modulion>
| <generic region instantiation>

**semantics:** A region is a means of providing mutually exclusive access to its locally declared data objects for the concurrent executions of processes (see clause 11). It determines visibility of locally created names in the same way as a module.

**static properties:** A *defining occurrence* in a region defines a *region* name. It has the region (seen as a modulion, i.e. excluding the context list and defining occurrence, if any) attached.

A region is developed piecewisely if and only if a context list is specified.

A region is a *region body* if and only if BODY is specified.

**static conditions:** If specified, the *simple name string* must be equal to the name string of the defining occurrence.

A region must not be surrounded by a block other than the imaginary outermost process definition.

**examples:** see 13.1-13.28

10.8 Program

**syntax:**

<program> ::= 

{ <module> | <spec module> | <region> | <spec region>
| <moreta declaration statement>
| <moreta synmode definition statement>
| <moreta newmode definition statement>
| <template> } +

**semantics:** A program consists of a list of program units (as given in the syntax rule) surrounded by an imaginary outermost process definition.
The definitions of the CHILL pre-defined names (see III.2) and the implementation defined built-in routines and integer modes are considered, for lifetime purposes, to be defined in the reach of the imaginary outermost process definition. For their visibility, see 12.2.

10.9 Storage allocation and lifetime

The time during which a location or procedure exists within its program is its lifetime.

A location is created by a declaration or by the execution of a GETSTACK or an ALLOCATE built-in routine call.

The lifetime of a location declared in the reach of a block is the time during which control lies in that block or in a procedure whose call originated from that block, unless it is declared with the attribute STATIC. The lifetime of a location declared in the reach of a modulion is the same as if it were declared in the reach of the closest surrounding block of the modulion. The lifetime of a location declared with the attribute STATIC is the same as if it were declared in the reach of the imaginary outermost process definition. This implies that for a location declaration with the attribute STATIC storage allocation is made only once, namely, when starting the imaginary outermost process. If such a declaration appears inside a procedure definition or process definition, only one location will exist for all invocations or activations.

The lifetime of a location created by executing a GETSTACK built-in routine call ends when the directly enclosing block terminates.

The lifetime of a location created by an ALLOCATE built-in routine call is the time starting from the ALLOCATE call until the time that the location cannot be accessed anymore by any CHILL program. The latter is always the case if a TERMINATE built-in routine is applied to an allocated reference value that references the location.

The lifetime of an access created in a loc-identity declaration is the directly enclosing block of the loc-identity declaration.

The lifetime of a procedure is the directly enclosing block of the procedure definition.

static properties: A location is said to be static if and only if it is a static mode location of one of the following kinds:

- A location name that is declared with the attribute STATIC or whose definition is not surrounded by a block other than the imaginary outermost process definition.
- A string element or string slice where the string location is static and either the left element and right element, or start element and slice size are constant.
- An array element where the array location is static and the expression is constant.
- An array slice where the array location is static and either the lower element and upper element or the first element and slice size are constant.
- A structure field where the structure location is static.
- A location conversion where the location occurring in it is static.

10.10 Constructs for piecewise programming

Modules and regions are the elementary units (pieces) in which a complete CHILL program that is developed piecewisely can be subdivided. The text of such pieces is indicated by remote constructs (see 10.10.1). CHILL defines the syntax and semantics of complete programs, in which all occurrences of remote pieces have been virtually replaced by the referred text.

10.10.1 Remote pieces

syntax:

<remote modulion> ::= 
[ <simple name string> : ] REMOTE <piece designator> ;   (1)

<remote spec> ::= 
[ <simple name string> : ] SPEC REMOTE <piece designator> ;   (2)

<remote context> ::= 
CONTEXT REMOTE <piece designator>
[ <context body> ] FOR   (3)
<context module> ::= (CONTEXT MODULE REMOTE <piece designator> ;)

<piece designator> ::= (character string literal)  
| (text reference name)  
| (empty)

<remote program unit> ::= (<simple name string> : ] REMOTE <piece designator> ;)

derived syntax: The notation:

CONTEXT MODULE REMOTE <piece designator>

is derived syntax for:

CONTEXT REMOTE <piece designator> FOR

MODULE SEIZE ALL; END;

NOTE – This construct is redundant but can be used for consistence checking.

semantics: Remote modulions, remote specs, remote contexts, context modules, and remote program units are means to represent the source text of a program as a set of (interconnected) files.

A piece designator refers in an implementation defined way to a description of a piece of CHILL source text, as follows:

- If the piece designator is empty, the source text is retrieved from a place determined by the structure of the program.
- If the piece designator contains a character string literal, the character string literal is used to retrieve the source text.
- If the piece designator contains a text reference name, the text reference name is interpreted in an implementation defined way to retrieve the source text.

A program with 1. remote modulions, 2. remote specs, 3. remote program units is equivalent to the program built by replacing each 1. remote modulion, 2. remote spec, 3. remote program unit by the piece of CHILL text referred to by its piece designator.

A program with remote contexts is equivalent to the program built by replacing each remote context by the piece of CHILL text referred to by its piece designator in which the context body has been virtually inserted immediately after the last occurrence of context body in the context list referred to by the piece designator.

If the designated piece is not available as CHILL text, then the piece designator in it is considered to refer to an equivalent piece of CHILL text which is introduced virtually.

Although the semantics of a remote piece is defined in terms of replacement, CHILL does not imply any textual substitution.

static conditions: The piece designator in a 1. remote modulion, 2. remote spec, 3. remote context, 4. context module, 5. remote program unit must refer to a description of a piece of source text which is a terminal production of a 1. module or region that is not a remote modulion, 2. spec module or spec region that is not a remote spec, 3., 4. context list which is not a remote context, 5. a program unit which is not remote.

When the source text referred to by the piece designator in a remote modulion starts with a defining occurrence, then the remote modulion must start with a simple name string which is the name string of that defining occurrence.

When the source text referred to by the piece designator in a remote spec starts with a simple name string, then the remote spec must start with the same simple name string.

When the source text referred to by the piece designator in a remote program unit starts with a simple name string, then the first defining occurrence in the remote program unit must be the same simple name string.

examples:

25.9  stack: REMOTE "example 27 or 28";

25.9  "example 27 or 28"
10.10.2 Spec modules, spec regions and contexts

syntax:

```
<spec module> ::= (1)
  <simple spec module> (1.1)
  | <module spec> (1.2)
  | <remote spec> (1.3)

<simple spec module> ::= (2)
  [ <context list> ] [ <simple name string> : ] SPEC MODULE
  <spec module body> END [ <simple name string> ]; (2.1)

<module spec> ::= (3)
  [ <context list> ] <simple name string> : MODULE SPEC
  <spec module body> END [ <simple name string> ]; (3.1)

<spec region> ::= (4)
  <simple spec region> (4.1)
  | <region spec> (4.2)
  | <remote spec> (4.3)

<simple spec region> ::= (5)
  [ <context list> ] [ <simple name string> : ] SPEC REGION
  <spec region body> END [ <simple name string> ]; (5.1)

<region spec> ::= (6)
  [ <context list> ] <simple name string> : REGION SPEC
  <spec region body> END [ <simple name string> ]; (6.1)

<context list> ::= (7)
  <context> { <context> }* (7.1)
  | <remote context> (7.2)

<context> ::= (8)
  CONTEXT <context body> FOR (8.1)
```

semantics: Simple spec modules, simple spec regions and contexts are used to specify static properties of names. They may be redundant but they can be used for piecewise programming.

Simple name strings in spec modules and spec regions are not names, they are not bound, and they have no visibility rules.

1. spec modules, 2. spec regions in a real reach indicate the properties of one or more 1. modules, 2. regions that are piecewisely compiled and that are considered to be enclosed in that reach. The texts of such 1. modules, 2. regions are indicated by occurrences of remote modulions. A context list indicates the surrounding reaches (note that a module or a region that is developed piecewisely always has a context list in front of it).

For each name string OP ! NS visible in the reach of a 1. module spec, 2. region spec and linked there to a quasi s defining occurrence and that is granted into a real reach as NP ! NS, a (virtual) grant statement with the same old name string OP ! NS and new name string NP ! NS is considered to be introduced in the reach of the corresponding 1. module body, 2. region body.

static conditions: In a spec module or a spec region, the optional simple name string following END may only be present if the optional simple name string before SPEC is present. When both are present, they must have equal name strings.

A context which has no directly enclosing group may not contain visibility statements.

A real reach that contains a 1. spec module, 2. spec region must also contain at least a remote modulion and vice versa.

If a real r reach contains a 1. module which is a module body, 2. region which is a region body, then it must contain also a 1. module spec, 2. region spec such that the simple name strings in front of them have equal name strings. The 1. module spec, 2. region spec is said to have a corresponding 1. module body, 2. region body.

A remote spec in a 1. spec module, 2. spec region must refer to a 1. spec module, 2. spec region.
A spec module or a spec region may not be surrounded by a block other than the imaginary outermost process definition.

**examples:**

23.2  
**letter_count:**

**SPEC MODULE**

SEIZE max;

*count: PROC (input ROW CHARS (max) IN,

output ARRAY ('A':'Z') INT OUT) END;

GRANT count;

END letter_count;

10.10.3 Quasi statements

**syntax:**

<quasi data statement> ::=  
| <quasi declaration statement> | <quasi definition statement> |

<quasi declaration statement> ::=  
| <quasi location declaration> | <quasi loc-identity declaration> |

<quasi location declaration> ::=  
| <defining occurrence list> <mode> |

<quasi loc-identity declaration> ::=  
| <defining occurrence list> <mode> |

<quasi definition statement> ::=  
| <synmode definition statement> | <newmode definition statement> |

<quasi synonym definition statement> ::=  
| <quasi synonym definition statement> | <quasi procedure definition statement> |

<quasi synonym definition statement> ::=  
| <quasi signal definition statement> | <signal definition statement> |

<quasi synonym definition> ::=  
| <quasi synonym definition> | <defining occurrence list> 

<quasi formal parameter list> ::=  
| <quasi formal parameter> | <defining occurrence list> 

<quasi formal parameter> ::=  
| <simple name string> | <parameter spec> |

<quasi process definition statement> ::=  
| <defining occurrence> : PROCESS ( [ <quasi formal parameter list> ] ) |

| END [ <simple name string> ] |
semantics: Quasi statements are used in spec modules, spec regions and contexts to specify static properties of names. Spec modules, spec regions and contexts may contain quasi statements and real statements. Quasi statements may be redundant, but are used for piecewise programming.

An implementation that can not guarantee the equality of the values between quasi constant synonym names and the corresponding real ones may disallow the indication of the constant value.

Note that in CHILL no quasi defining occurrences exist for label names.

static properties: Quasi statements are restricted forms of the corresponding statements, and have the same static properties.

The name defined by a defining occurrence in a quasi loc-identity declaration is referable if NONREF is not specified.

static conditions: Quasi statements are restricted forms of the corresponding statements and are subject to their static conditions.

A quasi synonym definition statement or a quasi signal definition statement may only be directly enclosed in a simple spec module, simple spec region or context. A synonym definition statement or a signal definition statement in a quasi definition statement may only be directly enclosed in a module spec or region spec.

10.10.4 Matching between quasi defining occurrences and defining occurrences

Two defining occurrences are said to match if they have identical semantic categories and:

- If they are synonym names, then they must have the same regionality and value, the root mode of their classes must be alike, they must both have an M-value, M-derived, M-reference, null or all class, and if the one which is quasi is literal, then so the other one must be.

- If they are newmode names or synmode names, then their modes must be alike.

- If they are location names or loc-identity names, then they must have the same regionality, they both must be or both not be referable, and their modes must be alike.

- If they are procedure names, then they must have the same regionality and generality, they both must be or both not be critical, they must satisfy the same conditions of alikeness as procedure modes, and corresponding (by position) simple name strings in the formal parameter list and quasi formal parameter list must be the same.

- If they are process names, then the parameters of their process definitions must satisfy the same conditions of matching and alikeness as the parameters of procedure names.

- If they are signal names, then they must both specify or both not specify TO, their lists of modes must have the same number of modes, and corresponding modes must be alike.

If two structure modes are novelty bound in a reach R, then they must have the same set of visible field names in R.

The following rules apply:

- If a name string in a reach that is not the reach of a spec module, spec region or context is bound to a quasi defining occurrence, then it must also be bound to a defining occurrence which is not a quasi defining occurrence, and further:
  
  - Let a name string be bound to a quasi defining occurrence QD and be bound also to a real defining occurrence RD in reach R, then:
    
    1) QD and RD must match as defined above; and
    
    2) RD and QD must both be enclosed in an enclosed group of R or both not be enclosed in the group of R or, if R is the reach of a module or region which is a module body or region body, then QD must be enclosed in the group of the corresponding module spec or region spec and RD must be enclosed in the group of R.
If a name string in a real reach R is bound to a quasi defining occurrence that is enclosed in the group of R (i.e., surrounded by a spec modulation), then it must also be bound to a real defining occurrence that is surrounded by the group of a module or region that are indicated by a remote modulation directly enclosed in R (informally, if the interface grants, so must the implementation). If the quasi defining occurrence is enclosed in the group of a module spec or a region spec, then the real one must be enclosed in the group of the corresponding modulation.

For each name string in the reach Q of a spec module or spec region directly enclosed in a real reach R that is bound to a defining occurrence not surrounded by Q, there must be an identical name string in the reach of a module or region that is indicated by a remote modulation directly enclosed in R that is bound to the same defining occurrence (informally, if the interface seizes, so must the implementation).

• If two name strings are bound to the same 1. real, 2. quasi defining occurrence in a reach, then both name strings must be bound to the same 1. quasi, 2. real defining occurrence, or both not be further bound.

A real novelty may not be novelty bound to two quasi novelties in any reach.

Let a quasi novelty QN and a real novelty RN be novelty bound to each other in a reach R; then RN and QN must both be enclosed in an enclosed group of R or both not be enclosed in the group of R, or if R is the reach of a module or region which is a module body or region body, then RN must be enclosed in the group of R and QN must be enclosed in the group of the corresponding module spec or region spec.

10.11 Genericity

Many algorithms solve problems on similarly structured data items whose component modes are different. Genericity provides a means to implement such algorithms as program schemes which are instantiated by substituting formal mode definitions by actual ones.

syntax:

```
<template> ::= (1)
    | <generic module template> (1.1)
    | <generic region template> (1.2)
    | <generic procedure template> (1.3)
    | <generic process template> (1.4)
    | <generic module mode template> (1.5)
    | <generic region mode template> (1.6)
    | <generic task mode template> (1.7)
    | <generic interface mode template> (1.8)
    | <remote program unit> (1.9)

<generic module template> ::= (2)
    [ <context list> ] [ <defining occurrence> : ]
    <generic part> MODULE [ BODY ] <module body> END
    [ <handler> ][ <simple name string> ] ;

<generic region template> ::= (3)
    [ <context list> ] [ <defining occurrence> : ]
    <generic part> REGION [ BODY ] <region body> END
    [ <handler> ][ <simple name string> ] ;

<generic procedure template> ::= (4)
    <defining occurrence> : <generic part> <procedure definition>
    [ <handler> ][ <simple name string> ] ;

<generic process template> ::= (5)
    <defining occurrence> : <generic part> <process definition>
    [ <handler> ][ <simple name string> ] ;

<generic module mode template> ::= (6)
    <generic part> <module mode specification>

<generic region mode template> ::= (7)
    <generic part> <region mode specification>
```
<generic task mode template> ::= (8)
<generic part> <task mode specification>
(8.1)
<generic interface mode template> ::= (9)
<generic part> <interface mode>
(9.1)
<generic part> ::= (10)
GENERIC { <seize statement> }* <formal generic parameter list>
(10.1)
<formal generic parameter list> ::= (11)
{ <formal generic parameter> }*
(11.1)
<formal generic parameter> ::= (12)
SYN <formal generic synonym list> ;
(12.1)
MODE <formal generic mode list> ;
(12.2)
PROC <formal generic procedure spec> ;
(12.3)
<formal generic synonym list> ::= (13)
<formal generic synonym> { ,<formal generic synonym> }*
(13.1)
<formal generic mode list> ::= (14)
<formal generic mode> { ,<formal generic mode> }*
(14.1)
<formal generic synonym> ::= (15)
<defining occurrence list> =

<mode> | ANY_DISCRETE | ANY_INT | ANY_REAL }
(15.1)
<formal generic mode> ::= (16)
<defining occurrence list> = <formal generic mode indication>
(16.1)
<formal generic mode indication> ::= (17)
ANY
(17.1)
| ANY_ASSIGN
(17.2)
| ANY_DISCRETE
(17.3)
| ANY_INT
(17.4)
| ANY_REAL
(17.5)
| <moreta mode name>
(17.6)
<formal generic procedure spec> ::= (18)
<simple name string> ( [ <formal parameter list> ] ) [ <result spec> ]
[ EXCEPTIONS ( <exception list> ) ]
(18.1)
<generic module instantiation> ::= (19)
<simple name string> : MODULE = NEW <generic module name>
{ <seize statement> }*
<actual generic parameter list> END [ <simple name string> ] ;
(19.1)
<generic region instantiation> ::= (20)
<simple name string> : REGION = NEW <generic region name>
{ <seize statement> }*
<actual generic parameter list> END [ <simple name string> ] ;
(20.1)
<generic procedure instantiation> ::= (21)
<simple name string> : PROC = NEW <generic procedure name>
{ <seize statement> }*
<actual generic parameter list> END [ <simple name string> ] ;
(21.1)
<generic process instantiation> ::= (22)
<simple name string> : PROCESS = NEW <generic process name>
{ <seize statement> }*
<actual generic parameter list> END [ <simple name string> ] ;
(22.1)
<generic moreta mode instantiation> ::= (23)
NEW <generic moreta mode name>
{ <seize statement> }*
<actual generic parameter list> END [ <simple name string> ] ;
(23.1)
<actual generic parameter list> ::= (24)
{ <actual generic parameter> }*
(24.1)
semantics: The word unit means either a module, a region, a procedure, a process, or a moreta mode.

A generic unit is a unit which contains a generic part.

A generic unit is a template from which nongeneric units may be obtained by a process called generic instantiation.

A generic unit may contain formal generic parameters. During generic instantiation a copy of the generic unit is made and the formal generic parameters are replaced by the actual generic parameters throughout the whole unit. After this replacement the generic part is deleted and thus a nongeneric unit is obtained.

static properties: The formal generic synonyms are characterized by two properties:

a) the properties which a formal generic parameter has inside the generic unit;
b) the properties which a corresponding actual generic parameter must have to be accepted:

mode:

<table>
<thead>
<tr>
<th>formal prop:</th>
<th>act prop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>properties of the given mode which must not have the non-value property.</td>
<td>value of the actual generic parameter must be a value of the mode.</td>
</tr>
</tbody>
</table>

ANY_DISCRETE:

<table>
<thead>
<tr>
<th>formal prop:</th>
<th>act prop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations available: :=, relational, PRED, SUCC, NUM, SIZE.</td>
<td>value of the actual generic parameter must be a value of a discrete mode.</td>
</tr>
</tbody>
</table>

ANY_INT:

<table>
<thead>
<tr>
<th>formal prop:</th>
<th>act prop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY_DISCRETE and +, −, *, /, mod, abs, rem.</td>
<td>value of the actual generic parameter must be a value of an integer mode.</td>
</tr>
</tbody>
</table>

ANY_REAL:

<table>
<thead>
<tr>
<th>formal prop:</th>
<th>act prop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations available: ANY_ASSIGN and relational, +, −, *, /</td>
<td>value of the actual generic parameter must be a value of a real mode.</td>
</tr>
</tbody>
</table>

The formal generic modes are characterized by two properties:

a) the properties which a formal generic parameter has inside the generic unit;
b) the properties which a corresponding actual generic parameter must have to be accepted:

ANY:

<table>
<thead>
<tr>
<th>formal prop:</th>
<th>actual prop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZE; cannot be used as the mode of a location or of a parameter; (can be used as a referenced mode).</td>
<td>any mode acceptable.</td>
</tr>
</tbody>
</table>

ANY_ASSIGN:

<table>
<thead>
<tr>
<th>formal prop:</th>
<th>act prop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations available: :=, comparison, SIZE.</td>
<td>mode must posses formal prop.</td>
</tr>
</tbody>
</table>

ANY_DISCRETE:

<table>
<thead>
<tr>
<th>formal prop:</th>
<th>act prop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations available: :=, relational, PRED, SUCC, NUM, SIZE.</td>
<td>mode must posses formal prop.</td>
</tr>
</tbody>
</table>

ANY_INT:

<table>
<thead>
<tr>
<th>formal prop:</th>
<th>act prop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY_DISCRETE and +, −, *, /, mod, abs, rem.</td>
<td>mode must posses formal prop.</td>
</tr>
</tbody>
</table>

ANY_REAL:

<table>
<thead>
<tr>
<th>formal prop:</th>
<th>act prop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>operations available: ANY_ASSIGN and relational, +, −, *, /</td>
<td>mode must posses formal prop.</td>
</tr>
</tbody>
</table>

moreta mode name:

<table>
<thead>
<tr>
<th>formal prop:</th>
<th>act prop:</th>
</tr>
</thead>
<tbody>
<tr>
<td>those of the mode.</td>
<td>same mode or any successor.</td>
</tr>
</tbody>
</table>
The formal generic procedures are characterized by two properties:

a) the properties which a formal generic parameter has inside the generic unit;

b) the properties which a corresponding actual generic parameter must have to be accepted:

- **formal prop:** according to the given formal generic procedure spec.
- **act prop:** the given formal generic procedure spec must be **compatible** with the class of the actual generic parameter.

**static conditions:** For derivation involving generic moreta mode templates the following restrictions apply: if the base is a template then any derived entity must also be a template. If the base is not a template a derived entity may be a template.

In a generic instantiation there must be exactly one actual generic parameter for each formal generic parameter of the generic unit being instantiated.

The restrictions on nesting of groups are given in the following table. It applies to plain groups, generic groups and generic instantiations.

<table>
<thead>
<tr>
<th>inner group</th>
<th>MODULE</th>
<th>REGION</th>
<th>PROC</th>
<th>PROCESS</th>
<th>Module Mode</th>
<th>Region Mode</th>
<th>Task Mode</th>
<th>Interface Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin-End</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>PROC</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>PROCESS</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MODULE</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>REGION</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Module Mode</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Region Mode</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Task Mode</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Interface Mode</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Program</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The table is based on the following correspondence between templates and entities of CHILL. For a template in the left column the restrictions of the corresponding entity in the right column apply:

- generic module template: procedure definition statement
- generic region template: region
- generic procedure template: procedure definition statement
- generic process template: process definition statement
- generic module mode template: procedure definition statement
- generic region mode template: region
- generic task mode template: process definition statement
- generic interface mode template: procedure definition statement

### 11 Concurrent execution

#### 11.1 Processes, tasks, threads and their definitions

A thread is either a process or a task. A process is the sequential execution of a series of statements. It may be executed concurrently with other threads. The behaviour of a process is described by a process definition (see 10.5), that describes the objects local to a process and the series of action statements to be executed sequentially.

A process is created by the evaluation of a start expression (see 5.2.15). It becomes active (i.e. under execution) and is considered to be executed concurrently with other threads. The created process is an activation of the definition indicated by the **process** name of the process definition. An unspecified number of processes with the same definition may be created and may be executed concurrently. Each process is uniquely identified by an instance value, yielded as the result
of the start expression or the evaluation of the **THIS** operator. The creation of a process causes the creation of its locally declared locations, except those declared with the attribute **STATIC** (see 10.9), and of locally defined values and procedures. The locally declared locations, values and procedures are said to have the same activation as the created process to which they belong. The imaginary outermost process (see 10.8), which is the whole CHILL program under execution, is considered to be created by a start expression executed by the system under whose control the program is executing. At the creation of a process, its formal parameters, if present, denote the values and locations as delivered by the corresponding actual parameters in the start expression.

A process is terminated by the execution of a stop action, by reaching the end of the process body or by terminating a handler specified at the end of the process definition (falling through). If the imaginary outermost process executes a stop action or falls through, the termination will be completed when and only when all other threads in the program are terminated.

A task is a sequential execution of a series of statements. It may be executed concurrently with other threads. The behaviour of a task is described by a task mode definition.

A task is created as part of the creation and initialization of a task mode location (see 4.1). It is called to belong to this task mode location. A task is terminated if its task mode location is destroyed (see 10.2). A thread is, at the CHILL programming level, always in one of two states: it is either active (i.e. under execution) or delayed (see 11.3). The transition from active to delayed is called the delaying of the thread; the transition from delayed to active is called the re-activation of the thread.

### 11.2 Mutual exclusion and regions

#### 11.2.1 General

Regions (see 10.7) and region locations (see 3.15) are a means of providing threads with mutually exclusive indirect access to locations declared inside the regions or region locations by granted procedures. Static context conditions (see 11.2.2) are made such that accesses by a thread other than the imaginary outermost process to locations declared inside a region can be made only by calling procedures that are defined inside the region or region mode and granted by the region or region mode.

**NOTE** – The only situation when the locations declared inside a region or region location can be directly accessed by a thread T is when the region or the region location is entered and its reach-bound initializations (if any) are performed by T. A **procedure** name is said to denote a **critical** procedure (and it is a **critical procedure** name) if it is defined inside a region and granted by the region or region mode.

A **component procedure** name is said to denote a **critical** component procedure (and it is a **critical component procedure** name) if it is defined inside a region mode and granted by the region mode. A region is said to be free if and only if control lies in none of its **critical** procedures or in the region itself performing reach-bound initializations.

A region location is said to be free if and only if control lies in none of its **critical** component procedures or in the region location itself performing reach-bound initializations. The region will be locked (to prevent concurrent execution) if:

- The region is entered (note that because regions are not surrounded by a block, no concurrent attempts can be made to enter the region).
- A **critical** procedure of the region is called.
- A process, delayed in the region, is re-activated.

The region location will be locked (to prevent concurrent execution) if:

- The region location is entered.
- A critical component procedure of the region location is called.
- A thread, which is delayed in the region location, is re-activated.

The region will be released, becoming free again, if:

- The region is left after having its reach-bound initializations performed.
- A **critical** procedure returns.
- A **critical** procedure executes an action that causes the executing process to become delayed (see 11.3). In the case of dynamically nested **critical** procedure calls, only the latest locked region will be released.
- A process executing a **critical** procedure terminates. In the case of dynamically nested **critical** procedure calls, all the regions locked by the process will be released.
The region location will be released, becoming free again, if:

- The region location is left after having its reach-bound initializations performed.
- A critical component procedure returns.
- A critical component procedure executes an action that causes the executing thread to become delayed (see 11.3). In the case of dynamically nested critical procedure calls, only the latest locked region will be released.
- A thread executing a critical component procedure terminates. In the case of dynamically nested critical component procedure calls, all the region locations locked by the thread will be released. If, while the region is locked, a thread attempts to call one of its critical procedures or a thread delayed in the region is re-activated, the thread is suspended until the region is released (note that the thread remains active in the CHILL sense).

If, while the region location is locked, a thread attempts to call one of its critical component procedures or a thread delayed in the region location is re-activated, the thread is suspended until the region location is released (note that the thread remains active in the CHILL sense). When a region is released and more than one thread has been suspended while attempting to call one of its critical component procedures or to be re-activated in one of its critical component procedures, only one thread will be selected to lock the region according to an implementation defined scheduling algorithm.

When a region location is released and more than one thread has been suspended while attempting to call one of its critical component procedures or to be re-activated in one of its critical component procedures, only one thread will be selected to lock the region location according to an implementation defined scheduling algorithm.

11.2.2 Regionality

To allow for checking statically that a location declared in a region can only be accessed by calling critical procedures or by entering the region for performing reach-bound initializations, the following static context conditions are enforced:

- the *regionality* requirements mentioned in the appropriate sections (assignment action, procedure call, send action, result action, etc.);
- *intra-regional* procedures are not general (see 10.4);
- critical procedures are neither general nor recursive (see 10.4).

To allow for checking statically that a component location declared in a region location can only be accessed by calling critical component procedures or by entering the region location for performing reach-bound initializations, the following static context conditions are enforced:

- the *regionality* requirements mentioned in the appropriate sections (assignment action, procedure call, send action, result action, etc.);
- intra-regional component procedures are not general (see 10.4);
- critical component procedures are neither general nor recursive (see 10.4).
- critical component procedures are also not inline (see 3.15).

A location and procedure call have a *regionality* which is *intra-regional* or *extra-regional*. A value has a *regionality* which is *intra-regional* or *extra-regional* or nil. These properties are defined as follows:

1) Location

A location is *intra-regional* if and only if any of the following conditions are fulfilled:

- It is an access name that is either:
  - a location name declared textually inside a region or spec region and not defined in a formal parameter of a critical procedure,
  - a location name declared textually inside a region mode and not defined in a formal parameter of a critical component procedure,
  - a loc-identity name, where the location in its declaration is intra-regional or that is defined in a formal parameter of an intra-regional procedure,
– a `loc-identity` name, where the `location` in its declaration is `intra-regional` or that is defined in a formal parameter of an `intra-regional` component procedure,
– a `location enumeration` name, where the `array location` or `string location` in the associated do action is `intra-regional`,
– a `location do-with` name, where the `structure location` in the associated do action is `intra-regional`.

• It is a dereferenced `bound reference`, where the `bound reference` `primitive value` in it is `intra-regional`.
• It is a dereferenced free reference, where the `free reference` `primitive value` in it is `intra-regional`.
• It is a dereferenced row, where the `row` `primitive value` in it is `intra-regional`.
• It is an `array element` or `array slice`, where the `array location` in it is `intra-regional`.
• It is a `string element` or `string slice`, where the `string location` in it is `intra-regional`.
• It is a `structure field`, where the `structure location` in it is `intra-regional`.
• It is a `location procedure call`, where in the `location procedure call` a `procedure name` is specified which is `intra-regional`.
• It is a `location` `built-in routine call`, that the CHILL definition or the implementation specifies to be `intra-regional`.
• It is a `location conversion`, where the `static mode location` in it is `intra-regional`.

A `location` which is not `intra-regional` is `extra-regional`.

2) `Value`

A `value` has a `regionality` depending on its class. If it has the M-derived class or the `all` class or the `null` class then it has `regionality nil`. Otherwise it has the M-value class or the M-reference class and it has a `regionality` depending on the mode M as follows:

If the `value` has the M-value class and M does not have the `referencing property` then the `regionality` is `nil`; otherwise the value is an `operand–7` (and has the `referencing property`) or a `conditional expression`:

If it is a `primitive value` then:

• If it is a `location contents` that is a `location`, then it is that of the `location`.
• If it is a `component location contents` that is a `component location`, then it is that of the component `location`.
• If it is a `name` value, then:
  – if it is a `synonym` `name` then it is that of the `constant` `value` in its definition;
  – if it is a `value do-with` `name` then it is that of the `structure` `primitive value` in the associated do action;
  – if it is a `value receive` `name` then it is `extra-regional`.
• If it is a `tuple` then if one of the `value` occurrences in it has `regionality` not `nil`, then it is that of that `value` (it does not matter which choice is made, see 5.2.5 static conditions); otherwise it is `nil`.
• If it is a `value array element` or a `value array slice` then it is that of the `array primitive value` in it.
• If it is a `value structure field` then it is that of the `structure primitive value` in it.
• If it is an `expression conversion` then it is that of the `expression` in it.
• If it is a `value procedure call` then it is that of the `procedure call` in it.
• If it is a `value component procedure call` then it is that of the component `procedure call` in it.
• If it is a `value` `built-in routine call` that the CHILL definition or the implementation specifies to be `intra-regional` or `extra-regional`.

If it is a `referenced location` then it is that of the `location` in it.

If it is a `conditional expression`, then if one of the `sub expression` occurrences in it has `regionality` not `nil`, then it is that of that `sub expression` (it does not matter which choice is made, see 5.3.2 static conditions); otherwise it is `nil`.

3) `Procedure name`

A `procedure name` is `intra-regional` if and only if it is defined inside a `region` or `spec region` and it is not `critical` (i.e. not granted by the region). Otherwise it is `extra-regional`.
A component procedure name is intra-regional if and only if it is defined inside a region mode and it is not critical (i.e. not granted by the region mode). Otherwise it is extra-regional.

4) Procedure call

A procedure call is intra-regional if it contains a procedure name which is intra-regional; otherwise it is extra-regional.

A component procedure call is intra-regional if it contains a component procedure name which is intra-regional; otherwise it is extra-regional.

A value is regionally safe for a non-terminal (used only for location, procedure call and procedure name) if and only if:

- the non-terminal is extra-regional and the value is not intra-regional;
- the non-terminal is intra-regional and the value is not extra-regional;
- the non-terminal has regionality nil.

11.3 Delaying of a thread

An active thread may become delayed by executing one of the following actions:

- delay action (see 6.16);
- delay case action (see 6.17);
- receive signal case action (see 6.19.2);
- receive buffer case action (see 6.19.3);
- send buffer action (see 6.18.3);
- call action to a component procedure of a region location (see 3.15.3);
- call action to a component procedure of a task location in case there is not enough storage to perform step c) 2) in 6.7 (see 3.15.4).

When a thread becomes delayed while its control lies within a critical procedure or a critical component procedure, the associated region is released. The dynamic context of the thread is retained until it is re-activated. The thread then attempts to lock the region or the region location again, which may cause it to be suspended.

11.4 Re-activation of a thread

A delayed thread may become re-activated if it is time supervised and a time interrupt occurs (see clause 9). It may also become re-activated if another thread executes one of the following actions:

- continue action (see 6.15);
- send signal action (see 6.18.2);
- send buffer action (see 6.18.3);
- receive buffer case action (see 6.19.3);
- release of a region location (see 3.15.3);
- at the beginning of the execution of an externally called component procedure of a task location (see 3.15.4).

When a thread, while having locked a region or region location, re-activates another thread, it remains active, i.e. it will not release the region or region location at that point.

11.5 Signal definition statements

syntax:

\[
\langle\text{signal definition statement}\rangle ::= \langle\text{SIGNAL}\rangle <\text{signal definition}> \{, <\text{signal definition}> \}^* ; \quad (I)
\]

\[
\langle\text{signal definition}\rangle ::= \langle\text{defining occurrence}\rangle [ = (<\text{mode}> \{, <\text{mode}> \}^*) ] \{ \text{TO} <\text{process name}> \} \quad (2.1)
\]

semantics: A signal definition defines a composing and decomposing function for values to be transmitted between processes. If a signal is sent, the specified list of values is transmitted. If no process is waiting for the signal in a receive case action, the values are kept until a process receives the values.
static properties: A defining occurrence in a signal definition defines a signal name.

A signal name has the following properties:

- It has an optional list of modes attached, that are the modes mentioned in the signal definition.
- It has an optional process name attached that is the process name specified after TO.

static conditions: No mode in a signal definition may have the non-value property.

examples:

15.27  SIGNAL initiate = (INSTANCE),
        terminate;

11.6 Completion of Region and Task locations

Both a REGION location L and a TASK location L contain waiting queues which contain threads waiting to execute a procedure of L.

After creation L is open, i.e. calls which cannot be executed immediately are put into a waiting queue of L. If L is a REGION object the calling thread is also blocked.

An RTL (REGION or TASK location) can be closed. If it is closed no calls are queued but instead an exception is caused at the corresponding call action.

A closed RTL L is executed until all its queues are empty. Then L is put into the state empty. If all RTLs which depend on L (see 12.2.6) are completed then L itself is completed.

A completed RTL may be destroyed. If an RTL is in one of the states "open", "closed" or "empty" it may not be destroyed.

12 General semantic properties

12.1 Mode rules

12.1.1 Properties of modes and classes

12.1.1.1 Read-only property

Informal

A mode has the read-only property if it is a read-only mode or contains a component or a sub-component, etc. which is a read-only mode.

Definition

A mode has the read-only property if and only if it is:

- an array mode with an element mode that has the read-only property;
- a structure mode where at least one of its field modes has the read-only property, where the field is not a tag field with an implicit read-only mode of a parameterized structure mode;
- a read-only mode.

12.1.1.2 Parameterizable modes

Informal

A mode is parameterizable if it can be parameterized.

Definition

A mode is parameterizable if and only if it is:

- a string mode;
- an array mode;
- a parameterizable variant structure mode.
12.1.1.3 Referencing property

Informal
A mode has the referencing property if it is a reference mode or contains a component or a sub-component, etc. which is a reference mode.

Definition
A mode has the referencing property if and only if it is:
- a reference mode;
- an array mode with an element mode that has the referencing property;
- a structure mode where at least one of its field modes has the referencing property.

12.1.1.4 Tagged parameterized property

Informal
A mode has the tagged parameterized property if it is a tagged parameterized structure mode or contains a component or a sub-component etc. which is a tagged parameterized structure mode.

Definition
A mode has the tagged parameterized property if and only if it is:
- an array mode with an element mode which has the tagged parameterized property;
- a structure mode where at least one of its field modes has the tagged parameterized property;
- a tagged parameterized structure mode.

12.1.1.5 Non-value property

Informal
A mode has the non-value property if no expression or primitive value denotation exists for the mode.

Definition
A mode has the non-value property if and only if it is:
- an event mode, a buffer mode, an access mode, an association mode or a text mode;
- an array mode with an element mode that has the non-value property;
- a structure mode where at least one of its field modes has the non-value property;
- a not_assignable moreta mode;
- an abstract moreta mode;
- a moreta mode where at least one of its components has the non-value property.

12.1.1.6 Root mode

Any mode M has a root mode defined as:
- if M is not a discrete range mode nor a floating point range mode;
- the parent mode of M, if M is a discrete range mode or a floating point range mode.

Any M-value class or M-derived class has a root mode which is the root mode of M.

12.1.1.7 Resulting class

Given two compatible classes (see 12.1.2.16), where the first one is either the all class, an M-value class or an M-derived class, where M and N are either a discrete mode, a floating point mode, a powerset mode or a string mode, the resulting class is defined as:
- the resulting class of the M-value class and the N-value class is the R-value class;
- the resulting class of the M-value class and the N-derived class or the all class is the P-value class;
- the resulting class of the M-derived class and the N-derived class is the R-derived class;
• the resulting class of the M-derived class and the all class is the P-derived class;

• the resulting class of the all class and the all class is the all class,

where R is the resulting mode of M and N, and P is the root mode of M.

Given two similar modes M and N, the resulting mode R is defined as:

• if the root mode of one is a fixed string mode and the other one is a varying string mode, then it is the root mode of the one (between M and N) whose root mode is a varying string mode;

• otherwise it is P.

Given a list C_i of pairwise compatible classes (i = 1, ..., n), the resulting class of the list of classes is recursively defined as the resulting class of the resulting class of the list C_i (i = 1, ..., n – 1) and the class C_n if n > 1; otherwise as the resulting class of C_1 and C_1.

12.1.2 Relations on modes and classes

12.1.2.1 General

In the following subclauses, the compatibility relations are defined between modes, between classes, and between modes and classes. These relations are used throughout this Recommendation | International Standard to define static conditions.

The compatibility relations themselves are defined in terms of other relations which are mainly used in this clause for the above-mentioned purpose.

12.1.2.2 Equivalence relations on modes

Informal

The following equivalence relations play a role in the formulation of the compatibility relations:

• Two modes are similar if they are of the same kind; i.e. they have the same hereditary properties.

• Two modes are v-equivalent (value-equivalent) if they are similar and also have the same novelty.

• Two modes are equivalent if they are v-equivalent and also possible differences in value representation in storage or minimum storage size are taken into account.

• Two modes are l-equivalent (location-equivalent) if they are equivalent and also have the same read-only specification.

• Two modes are alike if they are indistinguishable; i.e. if all operations that can be applied to objects of one of the modes can be applied to the other one as well, provided that novelty is not taken into account.

• Two modes are novelty bound if they are alike and have equal novelty specification.

Definition

In the following subclauses, the equivalence relations on modes are given in the form of a (partial) set of relations. The full equivalence algorithms are obtained by taking the symmetric, reflexive and transitive closure of this set of relations. The modes mentioned in the relations may be virtually introduced or dynamic. In the latter case, the complete equivalence check can only be performed at run time. Check failure of the dynamic part will result in the RANGEFAIL or TAGFAIL exception (see appropriate subclauses).

Checking two recursive modes for any equivalence requires the checking of associated modes in the corresponding paths of the set of recursive modes by which they are defined. Equivalence between the modes holds if no contradiction is found. (As a consequence, a path of the checking algorithm stops successfully if two modes which have been compared before, are compared.)

12.1.2.3 The relation similar

Two modes are similar if and only if:

• they are integer modes;

• they are floating point modes;

• they are boolean modes;
they are character modes;

they are set modes such that:
1) they define the same number of values;
2) for each set element name defined by one mode there is a set element name defined by the other mode which has the same name string and the same representation value;
3) they both are numbered set modes or both are unnumbered set modes;

they are discrete range modes with similar parent modes;

they are floating point range modes;

one is a discrete range mode or a floating point range mode whose parent mode is similar to the other mode;

they are powerset modes such that their member modes are equivalent;

they are bound reference modes such that their referenced modes are equivalent;

they are free reference modes;

they are row modes such that their referenced origin modes are equivalent;

they are procedure modes such that:
1) they have the same number of parameter specs and corresponding (by position) parameter specs have l-equivalent modes and the same parameter attributes, if present;
2) they both have or both do not have a result spec. If present, the result specs must have l-equivalent modes and the same attributes, if present;
3) they have the same list of exception names;
4) they have the same recursivity;

they are instance modes;

they are event modes such that they both have no event length or both have the same event length;

they are buffer modes such that:
1) they both have no buffer length or both have the same buffer length;
2) they have l-equivalent buffer element modes;

they are association modes;

they are access modes such that:
1) they both have no index mode or both have index modes which are equivalent;
2) at least one has no record mode, or both have record modes that are l-equivalent and that are both static record modes or both dynamic record modes;

they are text modes such that:
1) they have the same text length;
2) they have l-equivalent text record modes;

they are duration modes;

they are absolute time modes;

they are string modes such that their element modes are equivalent;

they are array modes such that:
1) their index modes are v-equivalent;
2) their element modes are equivalent;
3) their element layouts are equivalent;
4) they have the same number of elements. This check is dynamic if one or both modes is (are) dynamic. Check failure will result in the RANGEFAIL exception;
• they are structure modes which are not parameterized structure modes such that:
  1) in the strict syntax, they have the same number of fields and corresponding (by position) fields are equivalent;
  2) if they are both parameterizable variant structure modes, their lists of classes must be compatible;
• they are parameterized structure modes such that:
  1) their origin variant structure modes are similar;
  2) their corresponding (by position) values are the same. This check is dynamic if one or both modes is (are) dynamic. Check failure will result in the TAGFAIL exception;
• they are moreta modes whose mode names are synonymous.

12.1.2.4 The relation v-equivalent

Two modes are v-equivalent if and only if they are similar and have the same novelty.

12.1.2.5 The relation equivalent

Two modes are equivalent if and only if they are v-equivalent and:
• if one is a discrete range mode, the other must also be a discrete range mode and both upper bounds must be equal and both lower bounds must be equal;
• if one is a floating point range mode, the other must also be a floating point range mode and both upper bounds must be equal and both lower bounds must be equal and they must have the same precision;
• if one is a fixed string mode, the other one must also be a fixed string mode, and they must have the same string length. This check is dynamic in the case that one or both modes is (are) dynamic. Check failure will result in the RANGEFAIL exception;
• if one is a varying string mode, the other one must also be a varying string mode, and they must have the same string length. This check is dynamic in the case that one or both modes is (are) dynamic. Check failure will result in the RANGEFAIL exception.

12.1.2.6 The relation l-equivalent

Two modes are l-equivalent if and only if they are equivalent and if one is a read-only mode, the other must also be a read-only mode, and:
• if they are bound reference modes, their referenced modes must be l-equivalent;
• if they are row modes, their referenced origin modes must be l-equivalent;
• if they are array modes, their element modes must be l-equivalent;
• if they are structure modes which are not parameterized structure modes, corresponding (by position) fields in the strict syntax must be l-equivalent; if they are parameterized structure modes, their origin variant structure modes must be l-equivalent.

12.1.2.7 The relations equivalent and l-equivalent for fields

Two fields (both fields in the context of two given structure modes) are 1. equivalent, 2. l-equivalent if and only if both fields are fixed fields which are 1. equivalent, 2. l-equivalent or both are alternative fields which are 1. equivalent, 2. l-equivalent.

The relations equivalent and l-equivalent are recursively defined for corresponding fixed fields, variant fields, alternative fields and variant alternatives, respectively, in the following way:
• Fixed fields and variant fields
  1) Both fixed fields or variant fields must have equivalent field layout.
  2) Both field modes must be 1. equivalent, 2. l-equivalent.
• Alternative fields
  1) Both alternative fields have tag lists or both have no tag lists. In the former case, the tag lists must have the same number of tag field names and corresponding (by position) tag field names must denote corresponding fixed fields.
2) Both must have the same number of variant alternatives and corresponding (by position) variant alternatives must be 1. equivalent, 2. l-equivalent.
3) Both must have no ELSE specified or both must have ELSE specified. In the latter case, the same number of variant fields must follow and corresponding (by position) variant fields must be 1. equivalent, 2. l-equivalent.

- Variant alternatives
  1) Both variant alternatives must have the same number of case label lists and corresponding (by position) case label lists must either be both irrelevant, or both define the same set of values.
  2) Both variant alternatives must have the same number of variant fields and corresponding (by position) variant fields must be 1. equivalent, 2. l-equivalent.

12.1.2.8 The relation equivalent for layout

In the rest of the subclause, it will be assumed that each pos is of the form:

\[
\text{POS} (<\text{number}> , <\text{start bit}> , <\text{length}> )
\]

and that each step is of the form:

\[
\text{STEP} (<\text{pos}> , <\text{step size}> )
\]

Subclause 3.13.5 gives the appropriate rules to bring pos or step in the required form.

- Field layout

Two field layouts are equivalent if they are both NOPACK, or both PACK, or both pos. In the latter case the one pos must be equivalent to the other one (see below).

- Element layout

Two element layouts are equivalent if they are both NOPACK, both PACK, or both step. In the latter case the pos in the one step must be equivalent to the pos in the other one (see below) and step size must deliver the same values for the two element layouts.

- Pos

A pos is equivalent to another pos if and only if both word occurrences deliver the same value, both start bit occurrences deliver the same value and both length occurrences deliver the same value.

12.1.2.9 The relation alike

Two modes are alike if and only if they both are or both are not read-only modes and they both have novelty nil or both have the same novelty and:

- they are integer modes;
- they are boolean modes;
- they are character modes;
- they are similar set modes;
- they are discrete range modes with equal upper bounds and equal lower bounds;
- they are floating point range modes with equal upper bounds, equal lower bounds and equal precision;
- they are powerset modes such that their member modes are alike;
- they are bound reference modes such that their referenced modes are alike;
- they are free reference modes;
- they are row modes such that their referenced origin modes are alike;
they are procedure modes such that:
1) they have the same number of parameter specs and corresponding (by position) parameter specs have alike modes and the same parameter attributes, if present;
2) they both have or both do not have a result spec. If present, the result specs must have alike modes and the same attributes, if present;
3) they have the same list of exception names;
4) they have the same recursivity;

they are instance modes;

they are event modes such that they both have no event length or both have the same event length;

they are buffer modes such that:
1) they both have no buffer length or both have the same buffer length;
2) they have buffer element modes which are alike;

they are association modes;

they are access modes such that:
1) they both have no index mode or both have index modes that are alike;
2) at least one has no record mode or both have record modes that are alike and that are both static record modes or both dynamic record modes;

they are text modes such that:
1) they have the same text length;
2) their text record modes are alike;
3) their access modes are alike;

they are duration modes;

they are absolute time modes;

they are string modes such that:
1) their element modes are alike;
2) they have the same string length;
3) they both are fixed string modes or both are varying string modes;

they are array modes such that:
1) their index modes are alike;
2) their element modes are alike;
3) their element layouts are equivalent;
4) they have the same number of elements;

they are structure modes that are not parameterized structure modes such that:
1) in the strict syntax they have the same number of fields and corresponding (by position) fields are alike;
2) if they are both parameterizable variant structure modes, their lists of classes must be compatible;

they are parameterized structure modes such that:
1) their origin variant structure modes are alike;
2) their corresponding (by position) values are the same.

12.1.2.10 The relation alike for fields

Two fields (both fields in the context of two given structure modes) are alike if and only if both fields are fixed fields which are alike or both are alternative fields which are alike.
The relation alike is recursively defined for corresponding fixed fields, variant fields, alternative fields and variant alternatives, respectively, in the following way:

- **Fixed fields and variant fields**
  1) Both fixed fields or variant fields must have equivalent field layout.
  2) Both field modes must be alike.
  3) Both fixed fields or variant fields must have the same name string attached.

- **Alternative fields**
  1) Both alternative fields have tag lists or both have no tag lists. In the former case, the tag lists must have the same number of tag field names and corresponding (by position) tag field names must denote corresponding fixed fields.
  2) Both must have the same number of variant alternatives and corresponding (by position) variant alternatives must be alike.
  3) Both must have no ELSE specified or both must have ELSE specified. In the latter case, the same number of variant fields must follow and corresponding (by position) variant fields must be alike.

- **Variant alternatives**
  1) Both variant alternatives must have the same number of case label lists and corresponding (by position) case label lists must either be both irrelevant, or both define the same set of values.
  2) Both variant alternatives must have the same number of variant fields and corresponding (by position) variant fields must be alike.

12.1.2.11 The relation novelty bound

**Informal**

In a program, each quasi newmode must represent at most one real newmode. This is established as follows: when a name string is bound to both a real and a quasi defining occurrence all the newmodes involved are paired. The relation novelty bound is then established between novelties.

**Definition**

The relation novelty paired applies between two modes and a reach. For each name string bound in a reach R to both a real and a quasi defining occurrence:

- if they are synonym names, then the root modes of their classes are novelty paired in R;
- if they are location or loc-identity names, then their location modes are novelty paired in R;
- if they are procedure names, then the modes of the parameter specs and result spec, if present, are novelty paired in R;
- if they are process names, then the modes of the parameter specs are novelty paired in R;
- if they are signal names, then the modes in the list of modes are novelty paired in R.

If two modes are novelty paired in a reach R, then:

- if they are powerset modes, their member modes are novelty paired in R;
- if they are boundset reference modes, their referenced modes are novelty paired in R;
- if they are row modes, their referenced origin modes are novelty paired in R;
- if they are procedure modes, the modes of their parameter specs and result spec, if present, are novelty paired in R;
- if they are buffer modes, their buffer element modes are novelty paired in R;
- if they are access modes, their index modes, if present, and record modes, if present, are novelty paired in R;
- if they are text modes, their index modes, if present, are novelty paired in R;
- if they are array modes, their index modes and element modes are novelty paired in R;
• if they are parameterized structure modes, their origin variant structure modes are novelty paired in R;
• if they are parameterizable variant structure modes, their field modes and the modes of the classes in their list of classes are novelty paired in R;
• otherwise if they are structure modes, their field modes are novelty paired in R.

If two modes are novelty paired in a reach R and their novelties are not equal, then the real and quasi novelties of the modes are novelty bound to each other in R.

Two novelties are considered the same if they are:
• the same real novelty, or
• a real novelty and a quasi novelty that are novelty bound.

12.1.2.12 The relation read-compatible

Informal

The relation read-compatible is relevant for equivalent modes. A mode M is said to be read-compatible with a mode N if it or its possible (sub-)components have equal or more restrictive read-only specifications and, if they are reference modes, refer to l-equivalent locations. This relation is therefore non-symmetric.

Example:

READ REF READ CHAR is read-compatible with REF READ CHAR

Definition

A mode M is said to be read-compatible with a mode N (a non-symmetric relation) if and only if M and N are equivalent and, if N is a read-only mode, then M must also be a read-only mode and further:
• if M and N are bound reference modes, the referenced mode of M must be l-equivalent with the referenced mode of N;
• if M and N are row modes, the referenced origin mode of M must be l-equivalent with the referenced origin mode of N;
• if M and N are array modes, the element mode of M must be read-compatible with the element mode of N;
• if M and N are structure modes which are not parameterized structure modes, any field mode of M must be read-compatible with the corresponding field mode of N. If M and N are parameterized structure modes, the origin variant structure mode of M must be read-compatible with the origin variant structure mode of N.

12.1.2.13 The relations dynamic equivalent and read-compatible

Informal

The relations 1. dynamic equivalent, 2. dynamic read-compatible, are relevant only for modes that can be dynamic, i.e. string, array and variant structure modes. A parameterizable mode M is said to be 1. dynamic equivalent, 2. dynamic read-compatible with a (possibly dynamic) mode N, if there exists a dynamically parameterized version of M which is 1. equivalent, 2. read-compatible with N.

Definition

A mode M is 1. dynamic equivalent to a mode N, 2. dynamic read-compatible with a mode N (a non-symmetric relation) if and only if one of the following holds:
• M and N are string modes such that M(p) is 1. equivalent, 2. read-compatible with N, where p is the (possibly dynamic) length of N. The value p must not be greater than the string length of M. This check is dynamic if N is a dynamic mode. Check failure will result in a RANGEFAIL exception;
• M and N are array modes such that M(p) is 1. equivalent, 2. read-compatible with N, where p is such that \( NUM(p) - LOWER(M) + 1 \) is the (possibly dynamic) number of elements of N. The value p must not be greater than the upper bound of M. This check is dynamic if N is a dynamic mode. Check failure will result in a RANGEFAIL exception;
• M is a parameterizable variant structure mode and N is a parameterized structure mode such that M\( (p_1, \ldots, p_n) \) is 1. equivalent, 2. read-compatible with N, where \( p_1, \ldots, p_n \) denote the list of values of N.
12.1.2.14 The relation restrictable

Informal

The relation restrictable is relevant for equivalent modes with the referencing property. A mode M is said to be restrictable to a mode N if it or its possible (sub-)components refer to locations with equal or more restrictive read-only specification than those referenced by N. This relation is therefore non-symmetric.

Example:

REF READ INT is restrictable to REF INT
STRUCT (P REF READ BOOL) is restrictable to STRUCT (Q REF BOOL)

Definition

A mode M is restrictable to a mode N (a non-symmetric relation) if and only if M and N are equivalent and further:

- if M and N are bound reference modes, the referenced mode of M must be read-compatible with the referenced mode of N;
- if M and N are row modes, the referenced origin mode of M must be read-compatible with the referenced origin mode of N;
- if M and N are array modes, the element mode of M must be restrictable to the element mode of N;
- if M and N are structure modes, each field mode of M must be restrictable to the corresponding field mode of N.

12.1.2.15 Compatibility between a mode and a class

- Any mode M is compatible with the all class.
- A mode M is compatible with the null class if and only if M is a reference mode or a procedure mode or an instance mode.
- A mode M is compatible with the N-reference class if and only if M is a reference mode and one of the following conditions is fulfilled:
  1) N is a static non-moreta mode and M is a bound reference mode whose referenced mode is read-compatible with N;
  2) N is a static moreta mode and M is a bound reference mode MM and MM and N are on the same path;
  3) N is a static mode and M is a free reference mode;
  4) M is a row mode whose referenced origin mode is dynamic read-compatible with N.
- A mode M is compatible with the N-derived class if and only if M and N are similar.
- A mode M is compatible with the N-value class if and only if one of the following holds:
  1) if M does not have the referencing property, M and N must be v-equivalent;
  2) if M does have the referencing property, M must be restrictable to N.

12.1.2.16 Compatibility between classes

- Any class is compatible with itself.
- The all class is compatible with any other class.
- The null class is compatible with any M-reference class.
- The null class is compatible with the M-derived class or M-value class if and only if M is a reference mode, procedure mode or instance mode.
- The M-reference class is compatible with the N-reference class if and only if M and N are equivalent. If M and/or N is (are) a dynamic mode, the dynamic part of the equivalence check is ignored, i.e. no exceptions can occur.
- The M-reference class is compatible with the N-value class if and only if N is a reference mode and one of the following conditions is fulfilled:
  1) M is a static mode and N is a bound reference mode whose referenced mode is equivalent to M.
  2) M is a static mode and N is a free reference mode.
  3) N is a row mode whose referenced origin mode is dynamic equivalent with M.
• The M-derived class is compatible with the N-derived class or N-value class if and only if M and N are similar.
• The M-value class is compatible with the N-value class if and only if M and N are v-equivalent.

Two lists of classes are compatible if and only if both lists have the same number of classes and corresponding (by position) classes are compatible.

12.1.2.17 Conformance of mode names

Two mode names A and B conform to each other if and only if:
• either they both denote modes of the kind "REF MM", where MM is a moreta mode, and A and B are on the same path;
• or A syn B.

12.1.3 Definitions for moreta modes

If M is a moreta mode, then:

\[ M_S = \text{the specification part of } M \text{ (also the set of components in this part);} \]
\[ M_B = \text{the body part of } M \text{ (also the set of components in this part);} \]
\[ M_p = \text{the set of public components of } M_S \text{ defined directly in } M_S; \]
\[ M_{p+} = \text{the set of all public components of } M_S \text{ (including the inherited ones);} \]
\[ M_I = \text{the set of internal components of } M_S; \]
\[ M_{I+} = \text{the set of all internal components of } M_S \text{ (including the inherited ones);} \]
\[ M_R = \text{the set of private components of } M_B; \]
\[ M_{R+} = \text{the set of all private components of } M_B \text{ (including the inherited ones);} \]
\[ M_{CD} = \text{the set of constructors and destructors of } M_S; \]
\[ M_{inv} = \text{the invariant of } M_S; \]
\[ M_O = \text{the set of components (logically) contained in a location of mode } M. \]

If P is a component procedure of a moreta mode, then:

\[ PS = \text{the signature part of } P; \]
\[ PD = \text{the (complete) definition of } P; \]
\[ P_{Pre} = \text{the precondition of } P; \]
\[ P_{Post} = \text{the postcondition of } P; \]
\[ PE = \text{the set of exceptions specified in } PS. \]

If X is a procedure or a moreta mode, then:

\[ \text{attr}(X, A) = X \text{ contains the attribute } A: \text{ e.g. attr}(P, \text{INLINE}); \]
\[ \text{prop}(X, P) = X \text{ has the property } P: \text{ e.g. prop}(P, \text{assignable}); \]
\[ \text{GRANTed} = \text{explicitly exported}; \]
\[ \text{granted} = \text{GRANTed} \lor \text{implicitly exported}. \]

12.1.3.1 Qualified names of components of moreta modes and moreta locations

If M is the simple name string of a moreta mode, L is the simple name string of a moreta location, and C is the simple name of a component of M or of a public component of L then the name M.C or L.C can be used as a unique name for C in order to distinguish C from components with the same simple name string. If necessary the qualified name is assumed.

12.1.3.2 Successor and predecessor relations for moreta mode names

A moreta mode name DM is a direct successor (dsucc) of a moreta mode name BM if and only if there exist moreta mode names D and B: (B syn BM) \land (D syn DM) \land (B is mentioned in the inheritance clause of D).

A moreta mode name DM is a successor (succ) of a moreta mode name BM if and only if either DM syn BM or (\exists MM: (DM succ MM) \land (MM dsucc BM)).

The relation "predecessor" is the inverse of "successor".
Two moreta mode names A and B are on the same path if and only if \((A \ succ \ B) \lor (B \ succ \ A)\).

These relations hold isomorphically for modes of the kind "REF MM", where MM is a moreta mode.

12.1.3.3 Matching between procedure signatures and procedure definitions

A guarded procedure signature S matches a guarded procedure definition D if and only if:

- \(S.\langle\text{parameter list}\rangle\) matches \(D.\langle\text{formal parameter list}\rangle\) ∧
- \(S.\langle\text{result spec}\rangle\) and \(D.\langle\text{result spec}\rangle\) differ at most in the occurrence of \(\text{RESULT}\) ∧
- \(S.\langle\text{exception list}\rangle = D.\langle\text{exception list}\rangle\) ∧
- \(S.\langle\text{guarded procedure attribute list}\rangle = D.\langle\text{guarded procedure attribute list}\rangle\)

A parameter list P matches a formal parameter list F with strict syntax \(F'\) if and only if:

\(|P| = |F'|\) ∧

all corresponding elements of P and F' have the same mode and the same parameter attributes.

12.2 Visibility and name binding

The definition of visibility and name binding is based on the following terminology:

- **name string**: denotes a terminal string that has attached a *canonical* name string (see 2.7) and visibility properties;
- **name**: denotes a *simple name string* associated with the *defining occurrence* that has created it (see 10.1);
- **name**: denotes an applied occurrence of a name (with a possibly prefixed name string).

12.2.1 Degrees of visibility

The binding rules are based on the visibility of *name strings* in the reaches of a program. Within a reach, each *name string* has one of the following degrees of visibility:

<table>
<thead>
<tr>
<th>Visibility</th>
<th>Properties (informal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>directly visible</td>
<td><em>Name string</em> is visible by creation, granting or seizing or inheritance from spec to body</td>
</tr>
<tr>
<td>indirectly visible</td>
<td><em>Name string</em> is predefined or inherited via block nesting</td>
</tr>
<tr>
<td>publicly visible</td>
<td><em>Name string</em> is name of a public component of a <em>moreta mode</em> and is used in a moreta component name, or name string is name of a component of a moreta mode M and is used in a moreta component name which occurs inside M or any successor of M</td>
</tr>
<tr>
<td>privately visible</td>
<td><em>Name string</em> is name of a guarded procedure definition statement P contained in a moreta mode body B and the moreta mode specification of B does not contain a corresponding guarded procedure signature statement</td>
</tr>
<tr>
<td>invisible</td>
<td><em>Name string</em> may not be applied</td>
</tr>
</tbody>
</table>

A *name string* is said to be *visible* in a reach if it is either *directly visible* or *indirectly visible* in that reach. Otherwise the *name string* is said to be *invisible* in that reach. The program structuring statements and visibility statements determine uniquely to which visibility class each *name string* belongs.

When a *name string* is *visible* in a reach, it can be *directly linked* to another *name string* in another reach, or *directly linked* to a *defining occurrence* in the program. The rules for *direct linkage* are in 12.2.3. Notice that any application of a rule introduces a new *direct linkage* for a *name string*.
Based on direct linkage, the notion of (not necessarily direct) linkage is defined as follows:

A name string \( N_1 \), visible in reach \( R_1 \), is said to be linked to name string \( N_2 \) in reach \( R_2 \) or to defining occurrence \( D \), if and only if one of the following conditions holds:

- \( N_1 \) in \( R_1 \) is directly linked to \( N_2 \) in \( R_2 \) or to \( D \). However, if \( N_1 \) is directly linked to more than one defining occurrence in \( R_1 \), then all but one of these defining occurrences are superfluous, and \( N_1 \) is linked to an arbitrary one of them in \( R_1 \). This does not apply if \( N_1 \) is the name string of a simple guarded procedure signature statement in a moreta mode specification.

- \( N_1 \) in \( R_1 \) is directly linked to some \( N \) in some \( R \), and \( N \) in \( R \) is linked to \( N_2 \) in \( R_2 \) or to \( D \).

### 12.2.2 Visibility conditions and name binding

In each reach of a program, the following conditions must be satisfied:

- If a name string is visible in a reach and has more than one direct linkage, then it must be linked to exactly one real defining occurrence and one quasi defining occurrence, or to exactly one real defining occurrence in a simple guarded procedure signature statement in a mode \( M \) which is not an interface mode and exactly one real defining occurrence in a corresponding simple guarded procedure definition statement and possibly several real defining occurrences in a simple guarded procedure signature statement in one or more interface modes which are base modes of \( M \), or to possibly several real defining occurrences in a simple guarded procedure signature statement in one or more interface modes which are base modes of a moreta mode \( M \) and where the name string has no direct linkage in \( M \).

A name string \( NS \), visible in reach \( R \), is said to be bound in \( R \) to several defining occurrences according to the following rules:

- If \( NS \) is visible in \( R \), \( NS \) is bound to the defining occurrences to which it is linked in \( R \) (as a visible name string). If it is bound both to a quasi defining occurrence and a real defining occurrence, then the quasi one is redundant and does not participate further to visibility and name binding (i.e. it is not seized, granted nor inherited). If it is linked to exactly one real defining occurrence in a simple guarded procedure signature statement in a mode \( M \) which is not an interface mode and to exactly one real defining occurrence in a corresponding simple guarded procedure definition statement and to possibly several real defining occurrences in a simple guarded procedure signature statement in one or more interface modes which are base modes of \( M \) then it is bound to the occurrence in \( M \).

- Otherwise \( NS \) is not bound in \( R \).

*static condition*: The name string attached to each name directly enclosed in a reach must be bound in that reach.

*binding of names*: A name \( N \) with attached name string \( NS \) in a reach \( R \) is bound to the defining occurrences to which \( NS \) is bound in \( R \).

### 12.2.3 Visibility in reaches

#### 12.2.3.1 General

A name string is directly visible in a reach according to the following rules:

- the name string is seized into the reach (see 12.2.3.5);
- the name string is granted into the reach (see 12.2.3.4);
- there is a defining occurrence with that name string in the reach. In that case, the name string in the reach is directly linked to the defining occurrence. (Note that the name string may be directly linked to several defining occurrences in the reach);
- inside a constructor or destructor \( CD \) of a moreta mode \( M \), the name string of \( M \) is not hidden by the defining occurrence of the same name string in the definition of \( CD \) (but it may still be hidden by other defining occurrences of the same name string);
- at a place inside a constructor or destructor \( CD \) of a moreta mode \( M \), where the name string \( S \) of \( M \) is not hidden, \( S \) denotes either \( M \) or \( CD \) depending on the context;
- the reach is a 1. module body, 2. region body and the name string is directly visible in the reach of a corresponding 1. module spec, 2. region spec. The name string is directly linked to the name string in the corresponding reach.
A name string which is not directly visible in a reach is indirectly visible in it according to the following rules:

- The reach is a block, and the name string is visible in the directly enclosing reach. The name string is said to be inherited by the block, and is directly linked to the same name string in the directly enclosing reach.
- The reach is not a block in which the name string is inherited and the name string is a language (see III.2) or implementation defined name string. The name string is considered to be directly linked to a defining occurrence in the reach of the imaginary outermost process definition for its predefined meaning.

12.2.3.2 Visibility statements

syntax:

```
<visibility statement> ::= (1)
   <grant statement> (1.1)
   | <seize statement> (1.2)
```

semantics: Visibility statements are only allowed in modulion reaches and moreta mode reaches, and control the visibility of the name strings mentioned in them.

static properties: A visibility statement has one or two origin reaches (see 10.2) and one or two destination reaches attached, defined as follows:

- If the visibility statement is a seize statement, its destination reach is the reach directly enclosing the seize statement, and its origin reaches are the reaches directly enclosing that reach.
- If the visibility statement is a grant statement, then its origin reach is the reach directly enclosing the grant statement, and its destination reaches are the reaches directly enclosing that reach.
- If the visibility statement is a grant statement in a moreta mode specification, then its origin reach is the reach directly enclosing the grant statement, and its destination reaches are not the reaches directly enclosing that reach.

12.2.3.3 Prefix rename clause

syntax:

```
<prefix rename clause> ::= (1)
   ( <old prefix> –> <new prefix> ) ! <postfix> (1.1)
```

<old prefix> ::= (2)
```
   <prefix> (2.1)
   | <empty> (2.2)
```

<new prefix> ::= (3)
```
   <prefix> (3.1)
   | <empty> (3.2)
```

<postfix> ::= (4)
```
   <seize postfix> { , <seize postfix> }* (4.1)
   | <grant postfix> { , <grant postfix> }* (4.2)
```

derived syntax: A prefix rename clause where the postfix consists of more than one seize postfix (grant postfix) is derived syntax for several prefix rename clauses, one for each seize postfix (grant postfix), separated by commas, with the same old prefix and new prefix.

For example:

```
GRANT (p –> q) ! a , b ;
```

is derived syntax for

```
GRANT (p –> q) ! a , (p –> q) ! b ;
```

semantics: Prefix rename clauses are used in visibility statements to express change of prefix in prefixed name strings that are granted or seized. (Since prefix rename clauses can be used without prefix changes – when both the old prefix and the new prefix are empty – they are taken as the semantic base for visibility statements.)
static properties: A prefix rename clause has one or two origin reaches attached, which are the origin reaches of the visibility statement in which it is written.

A prefix rename clause has one or two destination reaches attached, which are the destination reaches of the visibility statement in which it is written.

A postfix has a set of name strings attached, which is the set of name strings attached to its seize postfix or the set of name strings attached to its grant postfix. These name strings are the postfix name strings of the prefix rename clause.

A prefix rename clause has a set of old name strings and a set of new name strings attached. Each postfix name string attached to the prefix rename clause gives both an old name string and a new name string attached to the prefix rename clause, as follows: the new name string is obtained by prefixing the postfix name string with the new prefix; the old name string is obtained by prefixing the postfix name string with the old prefix.

When a new name string and an old name string are obtained from the same postfix name string, the old name string is said to be the source of the new name string.

visibility rules: The new name strings attached to a prefix rename clause are visible in their destination reaches and are directly linked in those reaches to their sources in the origin reaches. If the prefix rename clause is part of a seize statement (grant statement), those name strings are seized (granted) in their destination reach (reaches).

A name string NS is said to be seizable by modulion M directly enclosed in reach R if and only if it is visible in R and it is neither linked in R to any name string in the reach of M nor directly linked to the defining occurrence of a predefined name string.

A name string NS is said to be grantable by modulion M directly enclosed in reach R if and only if it is visible in the reach of M and it is neither linked in it to any name string in R nor directly linked in it to the defining occurrence of a predefined name string.

static conditions: If a prefix rename clause is in a seize statement directly enclosed in the reach of modulion M then each of its old name strings must be:

• bound to several defining occurrences in the reach directly enclosing the reach of M; and
• seizable by M.

If a prefix rename clause is in a grant statement directly enclosed in the reach of modulion M then each of its old name strings must be:

• bound to several defining occurrences in the reach of M; and
• grantable by M.

A prefix rename clause that occurs in a grant statement (seize statement) must have a postfix that is a grant postfix (seize postfix).

examples:

25.35 (stack ! int – stack) ! ALL (1.1)

12.2.3.4 Grant statement

syntax:

<grant statement> ::= (1)
  GRANT <prefix rename clause> { , <prefix rename clause> }* ; (1.1)
  | GRANT <grant window> [ <prefix clause> ] [ <friend clause> ] ; (1.2)

<grant window> ::= (2)
  <grant postfix> { , <grant postfix> }* (2.1)

<grant postfix> ::= (3)
  <name string> [ ( < parameter list> ) [ [RETURNS] ( <result spec> ) ] ] (3.1)
  | <newmode name string> <forbid clause> (3.2)
  | [ <prefix> ! ] ALL (3.3)

<prefix clause> ::= (4)
  PREFIXED [ <prefix> ] (4.1)
semantics: Grant statements are a means of extending the visibility of name strings in a modulion reach into the directly enclosing reaches. **FORBID** can be specified only for newmode names which are structure modes. It means that all locations and values of that mode have fields which may be selected only inside the granting modulion, not outside.

The following visibility rules apply:

- If the grant statement contains prefix rename clause(s), the grant statement has the effect of its prefix rename clause(s) (see 12.2.3.3).
- If the grant statement contains grant windows, it is shorthand notation for a set of grant statements with prefix rename clauses constructed as follows:
  - for each grant postfix in the grant window, there is a corresponding grant statement;
  - the old prefix in their prefix rename clause is empty;
  - the new prefix in their prefix rename clause is the prefix attached to the prefix clause in the grant statement, or it is empty if there is no prefix clause in the original grant statement;
  - the postfix in the prefix rename clause is the corresponding postfix in the grant window.
- The notation **FORBID ALL** is shorthand notation for forbidding all the field names of the newmode name (see 12.2.5).
- If a prefix rename clause in a grant statement has a grant postfix which contains a prefix and ALL, then it is of the form:

  \[(OP \rightarrow NP) \! P \! ALL\]

  where \(OP\) and \(NP\) are the possibly empty old prefix and new prefix, respectively, and \(P\) is the prefix in the grant postfix. The prefix rename clause is then shorthand notation for a clause of the form:

  \[(OP \! P \rightarrow NP \! P) \! ALL\]

- If a friend clause is given, the visibility of the GRANTed objects is extended to only those groups which are mentioned in the friend name list.

*static properties:* A prefix clause has a prefix attached, defined as follows:

- If the prefix clause contains a prefix, then that prefix is attached.
- Otherwise the attached prefix is a simple prefix whose name string is determined as follows:
  - If the reach directly enclosing the prefix is a module or region, then the name string is the same as the one of the module name or region name of that modulion.
  - If the reach directly enclosing the prefix is a spec region or spec module, then the name string is the name string in front of SPEC.
A grant postfix has a set of name strings attached, defined as follows:

- If it is a name string, or contains a newmode name string, then the set containing only that name string.
- Otherwise, let OP be the (possibly empty) old prefix of the prefix rename clause in which the grant postfix is placed, the set contains all name strings of the form OP ! N (i.e. obtained by prefixing N with OP) for any name string N such that OP ! N is visible in the reach of the modulion in which the grant postfix is placed and grantable by this modulion.

**static conditions:** The newmode name string with forbid clause must be visible in the reach R of the modulion in which the grant statement is placed. The newmode name string must be bound in R to the defining occurrence of a newmode which must be a structure mode, and each field name in the forbid name list must be a field name of that mode. The newmode defining occurrence must be directly enclosed in R. All field names in a forbid name list must have different name strings.

If the grant statement is placed in the reach of a region or spec region, it must not grant a name string which is bound in that reach to the defining occurrence of:

- a location name; or
- a loc-identity name, where the location in its declaration is intra-regional; or
- a synonym name whose value is intra-regional.

The prefix rename clause in a grant statement must have a grant postfix.

If a grant statement contains a prefix clause which does not contain a prefix, then its directly enclosing modulion must not be a context and:

- if its directly enclosing modulion is a module or region, then it must be named (i.e. it must be headed by a defining occurrence followed by a colon);
- if its directly enclosing modulion is a spec module or a spec region, then it must be headed by a simple name string.

A name N contained in a friend name list of a grant statement, which is placed immediately in the reach of a group G, must be defined immediately in the reach of the group directly surrounding G.

If the grant statement occurs immediately inside a moreta specification then no prefixing must occur.

**examples:**

25.7 GRANT (–> stack ! char) ! ALL; (1.1)
6.44 gregorian_date, julian_day_number (2.1)

### 12.2.3.5 Seize statement

**syntax:**

```plaintext
<seize statement> ::= (1)
   SEIZE <prefix rename clause> { , <prefix rename clause> }* ; (1.1)
   | SEIZE <seize window> [ <prefix clause> ] ; (1.2)

<seize window> ::= (2)
   <seize postfix> { , <seize postfix> }* (2.1)

<seize postfix> ::= (3)
   <name string> [ ( <formal parameter list> ) [ [RETURNS] ( <result spec> ) ] ] (3.1)
   | [ <prefix> ! ] ALL (3.2)
```

**semantics:** Seize statements are a means of extending the visibility of name strings in group reaches into the reaches of directly enclosed modulions.

The following visibility rules apply:

- If the seize statement contains prefix rename clause(s), the seize statement has the effect of its prefix rename clause(s) (see 12.2.3.3).
• If the seize statement contains a seize window, it is shorthand notation for a set of seize statements with prefix rename clauses constructed as follows:
  – for each seize postfix in the seize window, there is a corresponding seize statement;
  – the old prefix in their prefix rename clause is the prefix attached to the prefix clause in the seize statement, or is empty if there is no prefix clause in the original seize statement;
  – the new prefix in their prefix rename clause is empty;
  – the postfix in their prefix rename clause is the corresponding postfix of the seize window.

• If a prefix rename clause in a seize statement has a seize postfix which contains a prefix and ALL, then it is of the form:

\[(OP \rightarrow NP) ! P \! \text{ALL}\]

where OP and NP are the possibly empty old prefix and new prefix, respectively, and P is the prefix in the seize postfix. The prefix rename clause is then shorthand notation for a clause of the form:

\[(OP \! P \rightarrow NP \! P) \! \text{ALL}\]

**static properties:** A seize postfix has a set of name strings attached, defined as follows:

• If the seize postfix is a name string, the set containing only the name string.

• Else, if the seize postfix is ALL, let OP be the (possibly empty) old prefix of the prefix rename clause of which the seize postfix is part, the set contains all name strings of the form OP ! S, for any name string S, such that:
  – OP!S is visible in the reach directly enclosing the modulion in which the seize statement is placed; and
  – it is seizable by this modulion; and
  – it is bound to a quasi defining occurrence if this modulion has a context in front of it.

**static conditions:** The prefix rename clause in a seize statement must have a seize postfix.

If a seize statement contains a prefix clause which does not contain a prefix, then its directly enclosing modulion must not be a context, and:

• if its directly enclosing modulion is a module or region, then it must be named (i.e. it must be headed by a defining occurrence followed by a colon);

• if its directly enclosing modulion is a spec module or a spec region, then it must be headed by a simple name string.

**examples:**

25.35 SEIZE (stack ! int \rightarrow stack) ! \text{ALL};

(1.1)

12.2.4 Visibility of set element names

A set element name may occur only in the context of a set literal.

If a set mode name is specified in the set literal, then the name string of a set element name can be bound to a set element name defining occurrence in the mode of the class of the set literal.

Otherwise, a set mode name is not specified, and then the name string can be bound to a set element name defining occurrence only if it is not visible in the reach in which the set literal is placed.

12.2.5 Visibility of field names

Field names may occur only in the following contexts:

• structure fields and value structure fields;

• labelled structure tuples;

• forbid clauses in grant statements.

Note that a field name may not occur in a grant postfix or in a seize postfix.
In each of these cases, the name string of the field name can be bound to a field name defining occurrence in the mode M or in the defining mode of M, obtained as follows:

- M is the mode of the structure location or (strong) structure primitive value;
- M is the mode of the structure tuple;
- M is the mode of the defining occurrence to which the newmode name string is bound in the reach in which the forbid clause is placed.

However, if the novelty of M is a defining occurrence that defines a newmode name that has been granted by a grant statement in a modulion as a grant postfix with a forbid clause, then the field names mentioned in the forbid name list are only visible:

- in the group of the granting modulion;
- if the novelty of M is novelty bound to a quasi novelty N, then in the group of the reach in which N is directly enclosed;
- if the modulion is a module spec or region spec, then in the reach of the corresponding modulion.

Outside these reaches the field names mentioned in the forbid name list are invisible and cannot be used.

12.2.6 Dependence of locations

An instance LI of a directly declared location L depends on that execution of its immediately surrounding group which created LI.

example:

SYNMODE TM = TASK SPEC ....
SYNMODE MM = MODULE SPEC
DCL T1 TM;
END MM;
DCL M1 MM;

The current instance M1-I of M1 contains an instance M1-I.T1 of MM.T1. M1-I.T1 has been created during the execution of "DCL M1 MM;". Therefore M1-I.T1 depends on M1-I.

Dependence of heap locations: GETSTACK and ALLOCATE create a new location L and deliver a reference value R for L. There are two cases:

a) the mode of R is known to be RM. In this case L depends on the creator of the relevant instance of RM;
b) the mode of R is not known (IF ALLOCATE(...) = ALLOCATE( ... ) .....). In this case L depends on the creator of the relevant instance of LM, where LM is the mode of L.

A location Lc which is a subcomponent of a location L depends on L.

12.3 Case selection

syntax:

\[
\text{<case label specification> ::=}< \text{case label list} \{ , <\text{case label list}> \}^* \quad (1) \\
\text{<case label list> ::=}( <\text{case label} \{ , <\text{case label}> \}^* ) \quad (2) \\
\text{ | <irrelevant>} \quad (2.1) \\
\text{<case label> ::=}<\text{discrete literal expression}> \quad (3) \\
\text{ | <literal range>} \quad (3.1) \\
\text{ | <discrete mode name>} \quad (3.2) \\
\text{ | ELSE} \quad (3.3) \\
\text{<irrelevant> ::=} (*) \quad (4) \\
\text{ | (3.4) }
\]

semantics: Case selection is a means of selecting an alternative from a list of alternatives. The selection is based upon a specified list of selector values. Case selection may be applied to:

- alternative fields (see 3.13.4), in which case a list of variant fields is selected;
- labelled array tuples (see 5.2.5), in which case an array element value is selected;
• conditional expressions (see 5.3.2), in which case an expression is selected;

• case action (see 6.4), in which case an action statement list is selected.

In the first, third and fourth situations, each alternative is labelled with a case label specification; in the labelled array tuple, each value is labelled with a case label list. For ease of description, the case label list in the labelled array tuple will be considered in this section as a case label specification with only one case label list occurrence.

Case selection selects that alternative which is labelled by the case label specification which matches the list of selector values. (The number of selector values will always be the same as the number of case label list occurrences in the case label specification.) A list of values is said to match a case label specification if and only if each value matches the corresponding (by position) case label list in the case label specification.

A value is said to match a case label list if and only if:

• the case label list consists of case labels and the value is one of the values explicitly indicated by one of the case labels or implicitly indicated in the case of ELSE;

• the case label list consists of irrelevant.

The values explicitly indicated by a case label are the values delivered by any discrete literal expression, or defined by the literal range or discrete mode name. The values implicitly indicated by ELSE are all the possible selector values which are not explicitly indicated by any associated case label list (i.e. belonging to the same selector value) in any case label specification.

static properties:

• An alternative fields with case label specification, a labelled array tuple, a conditional expression, or a case action has a list of case label specifications attached, formed by taking the case label specification in front of each variant alternative, value or case alternative, respectively.

• A case label has a class attached, which is, if it is a discrete literal expression, the class of the discrete literal expression; if it is a literal range, the resulting class of the classes of each discrete literal expression in the literal range; if it is a discrete mode name, the resulting class of the M-value class where M is the discrete mode name; if it is ELSE, the all class.

• A case label list has a class attached, which is, if it is irrelevant, then the all class, otherwise the resulting class of the classes of each case label.

• A case label specification has a list of classes attached, which are the classes of the case label lists.

• A list of case label specifications has a resulting list of classes attached. This resulting list of classes is formed by constructing, for each position in the list, the resulting class of all the classes that have that position.

A list of case label specifications is complete if, and only if, for all lists of possible selector values, a case label specification is present, which matches the list of selector values. The set of all possible selector values is determined by the context as follows:

• For a tagged variant structure mode it is the set of values defined by the mode of the corresponding tag field.

• For a tag-less variant structure mode it is the set of values defined by the root mode of the corresponding resulting class (this class is never the all class, see 3.13.4).

• For an array tuple, it is the set of values defined by the index mode of the mode of the array tuple.

• For a case action with a range list, it is the set of values defined by the corresponding discrete mode in the range list.

• For a case action without a range list, or a conditional expression, it is the set of values defined by M where the class of the corresponding selector is the M-value class or the M-derived class.

static conditions: For each case label specification the number of case label list occurrences must be equal.

For any two case label specification occurrences, their lists of classes must be compatible.

The list of case label specification occurrences must be consistent, i.e. each list of possible selector values matches at most one case label specification.
If the root mode of the class of a case label list is an integer mode, there must exist a predefined integer mode that contains all the values delivered by each case label.

examples:

11.9 (occupied)  (2.1)
11.58 (rook),(*)  (1.1)
8.26 (ELSE)  (2.1)

12.4 Definition and summary of semantic categories

This subclause gives a summary of all semantic categories which are indicated in the syntax description by means of an underlined part. If these categories are not defined in the appropriate subclauses, the definition is given here, otherwise the appropriate subclause will be referenced.

12.4.1 Names

Mode names

**absolute time mode name**: a name defined to be an absolute time mode.

**access mode name**: a name defined to be an access mode.

**array mode name**: a name defined to be an array mode.

**association mode name**: a name defined to be an association mode.

**boolean mode name**: a name defined to be a boolean mode.

**bound reference mode name**: a name defined to be a bound reference mode.

**buffer mode name**: a name defined to be a buffer mode.

**character mode name**: a name defined to be a character mode.

**discrete mode name**: a name defined to be a discrete mode.

**discrete range mode name**: a name defined to be a discrete range mode.

**duration mode name**: a name defined to be a duration mode.

**event mode name**: a name defined to be an event mode.

**floating point mode name**: a name defined to be a floating point mode.

**floating point range mode name**: a name defined to be a floating point range mode.

**free reference mode name**: a name defined to be a free reference mode.

**generic moreta mode name**: a name defined to be a generic moreta mode.

**interface mode name**: a name defined to be an interface mode.

**instance mode name**: a name defined to be an instance mode.

**integer mode name**: a name defined to be an integer mode.

**mode name**: see 3.2.1.

**module mode name**: a name defined to be a module mode.

**modulion or moreta mode name**: a name defined to be a modulion mode or a moreta mode.

**moreta mode name**: a name defined to be a moreta mode.

**parameterized array mode name**: a name defined to be a parameterized array mode.

**parameterized string mode name**: a name defined to be a parameterized string mode.

**parameterized structure mode name**: a name defined to be a parameterized structure mode.

**powerset mode name**: a name defined to be a powerset mode.

**procedure mode name**: a name defined to be a procedure mode.
region mode name: a name defined to be a region mode.
row mode name: a name defined to be a row mode.
set mode name: a name defined to be a set mode.
string mode name: a name defined to be a string mode.
structure mode name: a name defined to be a structure mode.
task mode name: a name defined to be a task mode.
text mode name: a name defined to be a text mode.
variant structure mode name: a name defined to be a variant structure mode.

Access names
location name: see 4.1.2.
location do-with name: see 6.5.4.
location enumeration name: see 6.5.2.
loc-identity name: see 4.1.3.

Value names
boolean literal name: see 5.2.4.4.
emptiness literal name: see 5.2.4.7.
synonym name: see 5.1.
value do-with name: see 6.5.4.
value enumeration name: see 6.5.2.
value receive name: see 6.19.2, 6.19.3.

Miscellaneous names
built-in routine name: any CHILL or implementation defined name denoting a built-in routine.
friend procedure or process name: see 12.2.3.4.
general procedure name: a procedure name whose generality is general.
generic module name: see 10.11.
generic procedure name: see 10.11.
generic process name: see 10.11.
generic region name: see 10.11.
label name: see 6.1, 10.6.
modulion name: see 12.2.3.4.
newmode name string: a name string bound to the defining occurrence of a newmode name.
on-reserved name: a name which is none of the reserved names mentioned in Appendix III.
procedure name: see 10.4.
process name: see 10.5.
set element name: see 3.4.5.
signal name: see 11.5.
tag field name: see 3.13.4.
undefined synonym name: see 5.1.
12.4.2 Locations

- **access location**: a *location* with an access mode.
- **array location**: a *location* with an array mode.
- **association location**: a *location* with an association mode.
- **buffer location**: a *location* with a buffer mode.
- **character string location**: a *location* with a *character* string mode.
- **discrete location**: a *location* with a discrete mode.
- **event location**: a *location* with an event mode.
- **floating point location**: a *location* with a floating point mode.
- **instance location**: a *location* with an instance mode.
- **integer location**: a *location* with an integer mode.
- **moreta location**: a *location* with a moreta mode.
- **static mode location**: a *location* with a static mode.
- **string location**: a *location* with a string mode.
- **structure location**: a *location* with a structure mode.
- **text location**: a *location* with a text mode.

12.4.3 Expressions and values

- **absolute time primitive value**: a *primitive value* whose class is *compatible* with an absolute time mode.
- **array expression**: an *expression* whose class is *compatible* with an array mode.
- **array primitive value**: a *primitive value* whose class is *compatible* with an array mode.
- **boolean expression**: an *expression* whose class is *compatible* with a boolean mode.
- **bound reference moreta location primitive value**: see 6.7.
- **bound reference primitive value**: a *primitive value* whose class is *compatible* with a bound reference mode.
- **character string expression**: an *expression* whose class is *compatible* with a *character* string mode.
- **constant value**: a *value* which is *constant*.
- **discrete expression**: an *expression* whose class is *compatible* with a discrete mode.
- **discrete literal expression**: a *discrete expression* which is *literal*.
- **duration primitive value**: a *primitive expression* whose class is *compatible* with a duration mode.
- **floating point expression**: an *expression* whose class is *compatible* with a floating point mode.
- **floating point literal expression**: a *floating point expression* which is *literal*.
- **free reference primitive value**: a *primitive value* whose class is *compatible* with a free reference mode.
- **instance primitive value**: a *primitive value* whose class is *compatible* with an instance mode.
- **integer expression**: an *expression* whose class is *compatible* with an integer mode.
- **integer literal expression**: an *integer expression* which is *literal*. 
literal expression:  
an expression which is literal.

powerset expression:  
an expression whose class is compatible with a powerset mode.

procedure primitive value:  
a primitive value whose class is compatible with a procedure mode.

reference primitive value:  
a primitive value whose class is compatible with either a bound reference mode, a free reference mode or a row mode.

row primitive value:  
a primitive value whose class is compatible with a row mode.

string expression:  
an expression whose class is compatible with a string mode.

string primitive value:  
a primitive value whose class is compatible with a string mode.

structure primitive value:  
a primitive value whose class is compatible with a structure mode.

12.4.4 Miscellaneous semantic categories

array mode:  
a mode in which the composite mode is an array mode.

constructor actual parameter list:  
see 4.1.2.

discrete mode:  
a mode in which the non-composite mode is a discrete mode.

inline guarded procedure definition statement:  
see 10.4.

location built-in routine call:  
see 6.7.

location procedure call:  
see 6.7.

moreta component procedure call:  
see 2.7.

moreta declaration statement:  
see 3.15.

moreta newmode definition statement:  
see 3.15.

moreta synmode definition statement:  
see 3.15.

non-percent character:  
a character which is not a percent (%).

non-reserved character:  
a character which is neither a quote (") nor a circumflex (^).

non-reserved wide character:  
a wide character which is neither a quote (") nor a circumflex (^).

non-special character:  
a character which is neither a circumflex (^) nor an open parenthesis (\).

simple guarded procedure definition statement:  
see 10.4.

simple guarded procedure signature statement:  
see 10.4.

string mode:  
a mode in which the composite mode is a string mode.

value built-in routine call:  
see 6.7.

value procedure call:  
see 6.7.

variant structure mode:  
a mode in which the non-composite mode is a variant structure mode.
13 Implementation options

13.1 Implementation defined built-in routines

semantics: An implementation may provide for a set of implementation defined built-in routines in addition to the set of language defined built-in routines.

The parameter passing mechanism is implementation defined.

predefined names: The name of an implementation defined built-in routine is predefined as a built-in routine name.

static properties: A built-in routine name may have a set of implementation defined exception names attached. A built-in routine call is a value (location) built-in routine call if and only if the implementation specifies that for a given choice of static properties of the parameters and the given static context of the call, the built-in routine call delivers a value (location).

The implementation specifies also the regionality of the value (location).

13.2 Implementation defined integer modes

An implementation defines the upper bound and lower bound of the integer mode INT. An implementation may define integer modes other than the ones defined by INT; e.g. short integers, long integers, unsigned integers. These integer modes must be denoted by implementation defined integer mode names. These names are considered to be newmode names, similar to INT. Their value ranges are implementation defined. These integer modes may be defined as root modes of appropriate classes.

13.3 Implementation defined floating point modes

An implementation defines the upper bound and the lower bound, the negative upper limit and the positive lower limit, the precision of the floating point mode FLOAT. An implementation may define floating point modes other than the ones defined by FLOAT; e.g. short float, long float. These floating point modes must be denoted by implementation defined floating point mode names. These names are considered to be newmode names, similar to FLOAT. Their value ranges, lower limits and precision are implementation defined. These floating point modes may be defined as root modes of appropriate classes.

13.4 Implementation defined process names

An implementation may define a set of implementation defined process names; i.e. process names whose definition is not specified in CHILL. The definition is considered to be placed in the reach of the imaginary outermost process or in any context. Processes of this name may be started and instance values denoting such processes may be manipulated.

13.5 Implementation defined handlers

An implementation may specify that an implementation defined handler is appended to a process or procedure definition; such a handler may handle any exception.

13.6 Implementation defined exception names

An implementation may define a set of exception names.

13.7 Other implementation defined features

- Static check of dynamic conditions (see 2.1.2)
- implementation directive (see 2.6)
- case of special simple name strings
• text reference name (see 2.7 and 10.10.1)
• default generality (see 10.4)
• set of values of duration modes (see 3.12.2)
• set of values of absolute time modes (see 3.12.3)
• default element layout (see 3.13.3)
• comparison of tag-less variant structure values (see 3.13.4)
• number of bits in a word (see 3.13.5)
• minimum bit occupancy (see 3.13.5)
• additional referable (sub-)locations (see 4.2.1)
• semantics of a location do-with name and value do-with name which is a variant field of a tag-less variant structure location (see 4.2.2 and 5.2.3)
• semantics of variant fields of tag-less variant structures (see 4.2.10, 5.2.14 and 6.2)
• semantics of location conversion (see 4.2.13)
• semantics of expression conversion and additional conditions (see 5.2.11)
• additional actual parameters in a start expression (see 5.2.15)
• ranges of values for literal and constant expressions (see 5.3.1)
• scheduling algorithm (see 6.15, 6.18.2, 6.18.3, 6.19.2, 6.19.3 and 11.2.1)
• releasing of storage in TERMINATE (see 6.20.4)
• denotation for files (see 7.1)
• operations on associations (see 7.1 and 7.2.1)
• non-exclusive associations (see 7.1)
• additional attributes of association values (see 7.2.2)
• semantics of associate parameters (see 7.4.2)
• ASSOCIATEFAIL exception (see 7.4.2)
• semantics of modify parameters (see 7.4.5)
• CREATEFAIL, DELETEFAIL and MODIFYFAIL exception (see 7.4.5)
• CONNECTFAIL exception (see 7.4.6)
• semantics of reading of records that are not legal values according to the record mode (see 7.4.9)
• additional timeoutable actions (see 9.2)
• TIMERFAIL exception (see 9.3.1, 9.3.2 and 9.3.3)
• precision of duration values (see 9.4.1 and 9.4.2)
• indication of constant value in quasi synonym definitions (see 10.10.3)
• regionality of built-in routines (see 11.2.2).
Appendix I

Character set for CHILL

The character set of CHILL is an extension of the CCITT Alphabet No. 5, International Reference Version, Recommendation V3. For the values whose representations are greater than 127, no graphical representation is defined.

The integer representation is the binary number formed by bits $b_8$ to $b_1$, where $b_1$ is the least significant bit.

<table>
<thead>
<tr>
<th>$b_2b_3b_4$</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_0b_1b_2b_3$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
<td>NUL</td>
<td>TC₂ (DLE)</td>
<td>SP</td>
<td>0</td>
<td>@</td>
<td>P</td>
<td>'</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>TC₁ (SOH)</td>
<td>DC₁</td>
<td>!</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>TC₂ (STX)</td>
<td>DC₂</td>
<td>&quot;</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>TC₃ (ETX)</td>
<td>DC₃</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>TC₄ (EOT)</td>
<td>DC₄</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>TC₅ (ENQ)</td>
<td>TC₆ (NAK)</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>TC₆ (ACK)</td>
<td>TC₇ (SYN)</td>
<td>&amp;</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>BEL</td>
<td>TC₈ (ETB)</td>
<td>'</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>FE₀ (BS)</td>
<td>CAN</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>FE₁ (HT)</td>
<td>EM</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>FE₂ (LF)</td>
<td>SUB</td>
<td>*</td>
<td>:</td>
<td>J</td>
<td>Z</td>
<td>j</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>FE₃ (VT)</td>
<td>ESC</td>
<td>+</td>
<td>;</td>
<td>K</td>
<td>[</td>
<td>k</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>FE₄ (FF)</td>
<td>IS₄ (FS)</td>
<td>,</td>
<td>&lt;</td>
<td>L</td>
<td>\</td>
<td>l</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>FE₅ (CR)</td>
<td>IS₅ (GS)</td>
<td>-</td>
<td>=</td>
<td>M</td>
<td>]</td>
<td>m</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>SO</td>
<td>IS₆ (RS)</td>
<td>.</td>
<td>&gt;</td>
<td>N</td>
<td>^</td>
<td>n</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>SI</td>
<td>IS₇ (US)</td>
<td>/</td>
<td>?</td>
<td>O</td>
<td>_</td>
<td>o</td>
</tr>
</tbody>
</table>
## Appendix II

### Special symbols

<table>
<thead>
<tr>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>semicolon terminator for statements etc.</td>
</tr>
<tr>
<td>,</td>
<td>comma separator in various constructs</td>
</tr>
<tr>
<td>(</td>
<td>left parenthesis opening parenthesis of various constructs</td>
</tr>
<tr>
<td>)</td>
<td>right parenthesis closing parenthesis of various constructs</td>
</tr>
<tr>
<td>[</td>
<td>left square bracket opening bracket of a tuple</td>
</tr>
<tr>
<td>]</td>
<td>right square bracket closing bracket of a tuple</td>
</tr>
<tr>
<td>(:</td>
<td>left tuple bracket opening bracket of a tuple</td>
</tr>
<tr>
<td>:)</td>
<td>right tuple bracket closing bracket of a tuple</td>
</tr>
<tr>
<td>:</td>
<td>colon label indicator, range indicator</td>
</tr>
<tr>
<td>.</td>
<td>dot field selection symbol</td>
</tr>
<tr>
<td>:=</td>
<td>assignment symbol assignment, initialization</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than relational operator</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal relational operator</td>
</tr>
<tr>
<td>=</td>
<td>equal relational operator, assignment, initialization, definition indicator</td>
</tr>
<tr>
<td>/=</td>
<td>not equal relational operator</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal relational operator</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than relational operator</td>
</tr>
<tr>
<td>+</td>
<td>plus addition operator</td>
</tr>
<tr>
<td>–</td>
<td>minus subtraction operator</td>
</tr>
<tr>
<td>*</td>
<td>asterisk multiplication operator, undefined value, unnamed value, irrelevant symbol</td>
</tr>
<tr>
<td>/</td>
<td>solidus division operator</td>
</tr>
<tr>
<td>//</td>
<td>double solidus concatenation operator</td>
</tr>
<tr>
<td>-&gt;</td>
<td>arrow referencing and dereferencing, prefix renaming</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>diamond start or end of a directive clause</td>
</tr>
<tr>
<td>/*</td>
<td>comment opening bracket start of a comment</td>
</tr>
<tr>
<td>*/</td>
<td>comment closing bracket end of a comment</td>
</tr>
<tr>
<td>'</td>
<td>apostrophe start or end symbol in various literals</td>
</tr>
<tr>
<td>#</td>
<td>sharp location and expression conversion</td>
</tr>
<tr>
<td>&quot;</td>
<td>quote start or end symbol in character string literals</td>
</tr>
<tr>
<td>!</td>
<td>prefixing operator prefixing of names</td>
</tr>
<tr>
<td>B'</td>
<td>literal qualification binary base for literal</td>
</tr>
<tr>
<td>b'</td>
<td>literal qualification binary base for literal</td>
</tr>
<tr>
<td>D'</td>
<td>literal qualification decimal base for literal</td>
</tr>
<tr>
<td>d'</td>
<td>literal qualification decimal base for literal</td>
</tr>
<tr>
<td>H'</td>
<td>literal qualification hexadecimal base for literal</td>
</tr>
<tr>
<td>h'</td>
<td>literal qualification hexadecimal base for literal</td>
</tr>
<tr>
<td>O'</td>
<td>literal qualification octal base for literal</td>
</tr>
<tr>
<td>o'</td>
<td>literal qualification octal base for literal</td>
</tr>
<tr>
<td>W'</td>
<td>literal qualification wide character or character string literal</td>
</tr>
<tr>
<td>w'</td>
<td>literal qualification wide character or character string literal</td>
</tr>
<tr>
<td>--</td>
<td>line end line end delimiter of in-line comments</td>
</tr>
</tbody>
</table>
Appendix III

Special simple name strings

III.1 Reserved simple name strings

<table>
<thead>
<tr>
<th>ABSTRACT</th>
<th>DO</th>
<th>MODULE</th>
<th>RETURNS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS</td>
<td>DOWN</td>
<td>NEW</td>
<td>ROW</td>
</tr>
<tr>
<td>AFTER</td>
<td>DYNAMIC</td>
<td>NEWMODE</td>
<td>SEIZE</td>
</tr>
<tr>
<td>ALL</td>
<td>ELSE</td>
<td>NONREF</td>
<td>SELF</td>
</tr>
<tr>
<td>AND</td>
<td>ELSIF</td>
<td>NOT_ASSIGNABLE</td>
<td>SEND</td>
</tr>
<tr>
<td>ANDIF</td>
<td>END</td>
<td>NOPACK</td>
<td>SET</td>
</tr>
<tr>
<td>ANY</td>
<td>ESAC</td>
<td>NOT</td>
<td>SIGNAL</td>
</tr>
<tr>
<td>ANY_ASSIGN</td>
<td>EVENT</td>
<td>OD</td>
<td>SIMPLE</td>
</tr>
<tr>
<td>ANY_DISCRETE</td>
<td>EVER</td>
<td>OF</td>
<td>SPEC</td>
</tr>
<tr>
<td>ANY_INT</td>
<td>EXCEPTIONS</td>
<td>ON</td>
<td>START</td>
</tr>
<tr>
<td>ANY_REAL</td>
<td>EXIT</td>
<td>OR</td>
<td>STATIC</td>
</tr>
<tr>
<td>ARRAY</td>
<td>EXIT</td>
<td>ORIF</td>
<td>STEP</td>
</tr>
<tr>
<td>Assignable</td>
<td>FOR</td>
<td>OUT</td>
<td>STOP</td>
</tr>
<tr>
<td>ASSERT</td>
<td>FOR</td>
<td>PACK</td>
<td>STRUCT</td>
</tr>
<tr>
<td>AT</td>
<td>FORBID</td>
<td>POST</td>
<td>SYN</td>
</tr>
<tr>
<td>BASED_ON</td>
<td>GENERAL</td>
<td>POST</td>
<td>SYNMODE</td>
</tr>
<tr>
<td>BEGIN</td>
<td>GENERIC</td>
<td>POWERSET</td>
<td>TASK</td>
</tr>
<tr>
<td>BIN</td>
<td>GOTO</td>
<td>PRE</td>
<td>TEXT</td>
</tr>
<tr>
<td>BODY</td>
<td>GRANT</td>
<td>PRIORITY</td>
<td>THEN</td>
</tr>
<tr>
<td>BOOLS</td>
<td>IF</td>
<td>PROC</td>
<td>THIS</td>
</tr>
<tr>
<td>BUFFER</td>
<td>IMPLEMENTS</td>
<td>PROCESS</td>
<td>TIMEOUT</td>
</tr>
<tr>
<td>BY</td>
<td>IN</td>
<td>RANGE</td>
<td>TO</td>
</tr>
<tr>
<td>CASE</td>
<td>INCOMPLETE</td>
<td>READ</td>
<td>UP</td>
</tr>
<tr>
<td>CAUSE</td>
<td>INIT</td>
<td>RECEIVE</td>
<td>VARYING</td>
</tr>
<tr>
<td>CHARs</td>
<td>INLINE</td>
<td>REF</td>
<td>WCHARS</td>
</tr>
<tr>
<td>CONSTR</td>
<td>INOUT</td>
<td>REGION</td>
<td>WHILE</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>INTERFACE</td>
<td>REIMPLEMENT</td>
<td>WITH</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>INTERFACE</td>
<td>REIMPLEMENT</td>
<td>WITH</td>
</tr>
<tr>
<td>CYCLE</td>
<td>INTERFACE</td>
<td>REM</td>
<td>WITH</td>
</tr>
<tr>
<td>DCL</td>
<td>LOC</td>
<td>REMOTE</td>
<td>WTEXT</td>
</tr>
<tr>
<td>DELAY</td>
<td>MOD</td>
<td>RESULT</td>
<td>XOR</td>
</tr>
<tr>
<td>DESTR</td>
<td>MODE</td>
<td>RETURN</td>
<td></td>
</tr>
</tbody>
</table>
### III.2 Predefined simple name strings

<table>
<thead>
<tr>
<th>ABS</th>
<th>EXP</th>
<th>LOWER</th>
<th>SETTEXTRECORD</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTIME</td>
<td>EXPIRED</td>
<td>MAX</td>
<td>SIN</td>
</tr>
<tr>
<td>ALLOCATE</td>
<td>FALSE</td>
<td>MILLISECS</td>
<td>SIZE</td>
</tr>
<tr>
<td>ARCCOS</td>
<td>FIRST</td>
<td>MIN</td>
<td>SUCC</td>
</tr>
<tr>
<td>ARCSIN</td>
<td>FLOAT</td>
<td>MINUTES</td>
<td>SQRT</td>
</tr>
<tr>
<td>ARCTAN</td>
<td>GETASSOCIATION</td>
<td>MODIFY</td>
<td>TAN</td>
</tr>
<tr>
<td>ASSOCIATE</td>
<td>GETSTACK</td>
<td>NULL</td>
<td>TERMINATE</td>
</tr>
<tr>
<td>ASSOCIATION</td>
<td>GETTEXTACCESS</td>
<td>NUM</td>
<td>TIME</td>
</tr>
<tr>
<td>BOOL</td>
<td>GETTEXTINDEX</td>
<td>OUTOFFILE</td>
<td>TRUE</td>
</tr>
<tr>
<td>CARD</td>
<td>GETTEXTRECORD</td>
<td>PRED</td>
<td>UPPER</td>
</tr>
<tr>
<td>CHAR</td>
<td>GETUSAGE</td>
<td>PTR</td>
<td>USAGE</td>
</tr>
<tr>
<td>CONNECT</td>
<td>HOURS</td>
<td>READABLE</td>
<td>VARIABLE</td>
</tr>
<tr>
<td>COS</td>
<td>INDEXABLE</td>
<td>READONLY</td>
<td>WAIT</td>
</tr>
<tr>
<td>CREATE</td>
<td>INSTANCE</td>
<td>READRECORD</td>
<td>WCHAR</td>
</tr>
<tr>
<td>DAYS</td>
<td>INT</td>
<td>READTEXT</td>
<td>WHERE</td>
</tr>
<tr>
<td>DELETE</td>
<td>INTTIME</td>
<td>READWRITE</td>
<td>WRITEABLE</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>ISASSOCIATED</td>
<td>SAME</td>
<td>WRITEONLY</td>
</tr>
<tr>
<td>DISSOCIATE</td>
<td>LAST</td>
<td>SECS</td>
<td>WRITERECORD</td>
</tr>
<tr>
<td>DURATION</td>
<td>LENGTH</td>
<td>SEQUENCIBLE</td>
<td>WRITETEXT</td>
</tr>
<tr>
<td>EOLN</td>
<td>LN</td>
<td>SETTEXTACCESS</td>
<td></td>
</tr>
<tr>
<td>EXISTING</td>
<td>LOG</td>
<td>SETTEXTINDEX</td>
<td></td>
</tr>
</tbody>
</table>

### III.3 Exception names

<table>
<thead>
<tr>
<th>ALLOCATEFAIL</th>
<th>EMPTY</th>
<th>PREFAIL</th>
<th>THIS_FAIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSERTFAIL</td>
<td>INVFAIL</td>
<td>RANGEFAIL</td>
<td>TIMERFAIL</td>
</tr>
<tr>
<td>ASSOCIATEFAIL</td>
<td>MODIFYFAIL</td>
<td>READFAIL</td>
<td>UNDERFLOW</td>
</tr>
<tr>
<td>CONNECTFAIL</td>
<td>NOTASSOCIATED</td>
<td>SENDFAIL</td>
<td>WRITEFAIL</td>
</tr>
<tr>
<td>CREATEFAIL</td>
<td>NOTCONNECTED</td>
<td>SPACEFAIL</td>
<td></td>
</tr>
<tr>
<td>DELAYFAIL</td>
<td>OVERFLOW</td>
<td>TAGFAIL</td>
<td></td>
</tr>
<tr>
<td>DELETEFAIL</td>
<td>POSTFAIL</td>
<td>TEXTFAIL</td>
<td></td>
</tr>
</tbody>
</table>
Appendix IV

Program examples

IV.1 Operations on integers

integer_operations:

MODULE

add:

PROC (i,j INT) RETURNS (INT) EXCEPTIONS (OVERFLOW);
RESULT i+j;
END add;

mult:

PROC (i,j INT) RETURNS (INT) EXCEPTIONS (OVERFLOW);
RESULT i*j;
END mult;

GRANT add, mult;

SYNMODE operand_mode=INT;
GRANT operand_mode;
SYN neutral_for_add=0,
neutral_for_mult=1;
GRANT neutral_for_add,
neutral_for_mult;

END integer_operations;

IV.2 Same operations on fractions

fraction_operations:

MODULE

NEWMODE fraction=STRUCT (num,denum INT);

add:

PROC (f1,f2 fraction) RETURNS (fraction) EXCEPTIONS (OVERFLOW);
RETURN [f1.num*f2.denum+f2.num*f1.denum,f1.denum*f2.denum];
END add;

mult:

PROC (f1,f2 fraction) RETURNS (fraction) EXCEPTIONS (OVERFLOW);
RETURN [f1.num*f2.num,f2.denum*f1.denum];
END mult;

GRANT add, mult;
SYNMODE operand_mode=fraction;
GRANT operand_mode;
SYN neutral_for_add fraction=[ 0,1 ],
neutral_for_mult fraction=[ 1,1 ];
GRANT neutral_for_add,
neutral_for_mult;

END fraction_operations;
IV.3 Same operations on complex numbers

```plaintext
complex_operations:
MODULE
NEWMODE complex=STRUCT (re,im FLOAT);

PROC (c1,c2 complex) RETURNS (complex) EXCEPTIONS (OVERFLOW);
RETURN [c1.re+c2.re,c1.im+c2.im];
END add;

PROC (c1,c2 complex) RETURNS (complex) EXCEPTIONS (OVERFLOW);
RETURN [c1.re*c2.re-c1.im*c2.im,c1.re*c2.im+c1.im*c2.re];
END mult;

GRANT add, mult;
SYNMODE operand_mode=complex;
GRANT operand_mode;
SYN neutral_for_add=complex [ 0.0,0.0 ],
neutral_for_mult=complex [ 1.0,0.0 ];
GRANT neutral_for_add,
neutral_for_mult;
END complex_operations;
```

IV.4 General order arithmetic

```plaintext
general_order_arithmetic: /* from collected algorithms from CACM no. 93 */
MODULE
PROC (a INT INOUT, b,c,order INT)
EXCEPTIONS (wrong_input);
DCL d INT;
ASSERT b>0 AND c>0 AND order>0
ON (ASSERTFAIL):
CAUSE wrong_input;
END;
CASE order OF
(1):  a := b+c;
RETURN;
(2):  d := 0;
(ELSE): d := 1;
ESAC;
DO FOR i := 1 TO c;
op (a,b,d,order–1);
d := a;
OD;
RETURN;
END op;
GRANT op;
END general_order_arithmetic;
```

IV.5 Adding bit by bit and checking the result

```plaintext
add_bit_by_bit:
MODULE
PROC (a STRUCT (a2,a1 BOOL) IN, b STRUCT (b2,b1 BOOL) IN)
RETURNS (STRUCT (c4,c2,c1 BOOL));
```
DCL c STRUCT (c4,c2,c1 BOOL);
DCL k2,x,w,t,s,r BOOL;
DO WITH a,b,c;
    k2 := a1 AND b1;
    c1 := NOT k2 AND (a1 OR b1);
    x := a2 AND b2 AND k2;
    w := a2 OR b2 OR k2;
    t := b2 AND k2;
    s := a2 AND k2;
    r := a2 AND b2;
    c4 := r OR s OR t;
    c2 := x OR (w AND NOT c4);
OD;
RETURN c;
END adder;
GRANT adder;
END add_bit_by_bit;

exhaustive_checker:
MODULE
    SEIZE adder;
    SYNMODE res=ARRAY (1:16) STRUCT (c4,c2,c1 BOOL);
    DCL r INT, results res;
    r := 0;
    DO FOR a2 IN BOOL;
        DO FOR a1 IN BOOL;
            DO FOR b2 IN BOOL;
                DO FOR b1 IN BOOL;
                    r+ := 1;
                    results (r) := adder ([a2,a1], [b2,b1]);
                OD;
            OD;
        OD;
    OD;
ASSERT
    results=res [ [FALSE,FALSE,FALSE ],[FALSE,FALSE,TRUE ],
        [FALSE,TRUE,FALSE ],[FALSE,TRUE,TRUE ],
        [FALSE,TRUE,FALSE ],[FALSE,TRUE,TRUE ],
        [FALSE,TRUE,FALSE ],[FALSE,TRUE,TRUE ],
        [FALSE,TRUE,FALSE ],[FALSE,TRUE,TRUE ],
        [FALSE,TRUE,FALSE ],[FALSE,TRUE,TRUE ],
        [FALSE,TRUE,FALSE ],[TRUE,FALSE,FALSE ],
        [FALSE,TRUE,FALSE ],[FALSE,TRUE,TRUE ],
        [TRUE,TRUE,FALSE ],[TRUE,TRUE,FALSE ],
        [TRUE,TRUE,FALSE ],[TRUE,TRUE,FALSE ]];
END exhaustive_checker;

IV.6 Playing with dates

playing_with_dates:
MODULE /* from collected algorithms from CACM no. 199 */
SYNMODE  month=SET (jan,feb,mar,apr,may,jun,
        jul,aug,sep,oct,nov,dec);
NEWMODE date=STRUCT (day INT (1:31), mo month, year INT);

gregorian_date:
PROC (julian_day_number INT) RETURNS (date);
DCL  j INT;= julian_day_number,
    d,m,y INT;
    j:= 1_721_119;
    y:= (4 * j – 1) / 146_097;
    j:= 4 * j – 1 – 146_097 * y;
    d:= j / 4;
    j:= (4 * d + 3) / 1_461;
IV.7 Roman numerals

 Roman:

 MODULE
 SEIZE n,rn;
 GRANT convert;
 convert:
 PROC () EXCEPTIONS (string_too_small);
 DCL r INT:= 0;
 DO WHILE n>=1_000;
   rn(r) := 'M';
   n := n - 1_000;
   r + := 1;
 OD;
 IF n>500 THEN rn(r) := 'D';
   n := n - 500;
   r + := 1;
 FI;
 DO WHILE n>=100;
   rn(r) := 'C';
   n := n - 100;
   r + := 1;
 OD;
 IF n>=50 THEN rn(r) := 'L';

\[ n := 50; \]
\[ r := 1; \]
\[ \text{FI}; \]
\[ \text{DO } \text{WHILE } n \geq 10; \]
\[ \text{rn}(r) := 'X'; \]
\[ n := 10; \]
\[ r := 1; \]
\[ \text{OD}; \]
\[ \text{IF } n \geq 5 \text{ THEN } \text{rn}(r) := 'V'; \]
\[ n := 5; \]
\[ r := 1; \]
\[ \text{FI}; \]
\[ \text{DO } \text{WHILE } n \geq 1; \]
\[ \text{rn}(r) := 'I'; \]
\[ n := 1; \]
\[ r := 1; \]
\[ \text{OD}; \]
\[ \text{RETURN}; \]
\[ \text{END} \text{ON} \text{(RANGEFAIL): } \text{DO FOR } i := 0 \text{ TO UPPER} \text{(rn); } \]
\[ \text{rn}(i) := ';'; \]
\[ \text{OD}; \]
\[ \text{CAUSE } \text{stringtoo_small}; \]
\[ \text{END} \text{convert}; \]
\[ \text{END} \text{Roman}; \]
\[ \text{test: } \]
\[ \text{MODULE } \]
\[ \text{SEIZE } \text{convert}; \]
\[ \text{DCL } n \text{ INT INIT:= 1979; } \]
\[ \text{DCL } \text{rn CHARS} \text{(20) INIT:= (20)" ";} \]
\[ \text{GRANT } n, \text{rn}; \]
\[ \text{convert}(); \]
\[ \text{ASSERT } \text{rn}="MDCCCCLXXVIIII"//(6)" "; \]
\[ \text{END test}; \]

IV.8 Counting letters in a character string of arbitrary length

1  letter_count:
2  MODULE
3    SEIZE max;
4  DCL letter POWERSET CHAR INIT:= ['A': 'Z'];
5  count:
6  PROC (input ROW CHARS (max) IN, output ARRAY ('A': 'Z') INT OUT);
7  output := [(ELSE) : 0];
8  DO FOR i := 0 TO UPPER (input ->);
9  IF input -> (i) IN letter
10    THEN
11      output (input -> (i)) + := 1;
12  FI;
13  OD;
14  END count;
15  GRANT count;
16  END letter_count;
17  test:
18  MODULE
19    SYNMODE results=ARRAY ('A': 'Z')INT;
20  DCL c CHARS (10) INIT:= "A-B<ZAA9K' ";
21  DCL output results;
22  SYN max=10_000;
23  GRANT max;
24  SEIZE count;
25  count (-> c,output);
26  ASSERT output=results [( 'A') : 3, ('B', 'K', 'Z') : 1, (ELSE) : 0];
27  END test;
IV.9 Prime numbers

```plaintext
prime: MODULE

SYN max = H'7FFF;
NEWMODE number_list = POWERSET INT (2:max);
SYN empty = number_list [* ];
DCL sieve number_list INIT:=[ 2:max ],
    primes number_list INIT:= empty;
GRANT primes;
DO WHILE sieve/=empty;
    primes OR:=[MIN (sieve)];
    DO FOR j := MIN (sieve) BY MIN (sieve) TO max;
        sieve – := [j];
    OD;
OD;
END prime;
```

IV.10 Implementing stacks in two different ways, transparent to the user

```plaintext
stack: MODULE

NEWMODE element =STRUCT (a INT, b BOOL);
stacks_1: MODULE

SEIZE element;
SYN max=10,000,min=1;
DCL stack ARRAY (min : max) element,
    stackindex INT INIT:= min;

push: PROC (e element) EXCEPTIONS (overflow);
    IF stackindex=max
        THEN CAUSE overflow;
    FI;
    stackindex + := 1;
    stack (stackindex) := e;
    RETURN;
END push;

pop: PROC () EXCEPTIONS (underflow);
    IF stackindex=min
        THEN CAUSE underflow;
    FI;
    stackindex – := 1;
    RETURN;
END pop;

elem: PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds);
    IF i<min OR i>max
        THEN CAUSE bounds;
    FI;
    RETURN stack (i);
END elem;

GRANT push,pop,elem;
END stacks_1;
stacks_2: MODULE
```
SEIZE element;
NEWMODE cell=STRUCT (pred,succ REF cell,info element);
DCL p,last,first REF cell INIT:= NULL;

push:
PROC (e element) EXCEPTIONS (overflow);
p := ALLOCATE (cell) ON (ALLOCATEFAIL) : CAUSE overflow;
END;
IF last=NULL THEN first := p;
last := p;
ELSE last->.succ := p;
p ->. pred := last;
last := p;
FI;
last->. info := e;
RETURN;
END push;

pop:
PROC () EXCEPTIONS (underflow);
IF last=NULL THEN CAUSE underflow;
p := last;
last := last->. pred;
IF last = NULL THEN first := NULL;
ELSE last->. succ := NULL;
FI;
TERMINATE(p);
RETURN;
END pop;

elem:
PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds);
IF first=NULL THEN CAUSE bounds;
FI;
p := first;
DO FOR j := 2 TO i;
IF p->. succ=NULL THEN CAUSE bounds;
FI;
p := p->. succ;
OD;
RETURN p->. info;
END elem;
/* GRANT push,pop,elem; */
END stacks_2;
END stack;

IV.11 Fragment for playing chess

chess_fragments:
MODULE
NEWMODE piece=STRUCT (color SET (white,black),
kind SET (pawn,rook,knight,bishop,queen,king));
NEWMODE column=SET (a,b,c,d,e,f,g,h);
NEWMODE line=INT (1 : 8);
ISO-IEC 9496 : 2002(E)

186

ITU-T Rec. Z.200     (1999 E)

NEWMODE square=STRUCT (status SET (occupied,free),
CASE status OF
(occupied) : p piece,
(free) :  
ESAC);

NEWMODE board=ARRAY (line) ARRAY (column) square;
NEWMODE move=STRUCT (lin_1,lin_2 line,
col_1,col_2 column);

initialize:
PROC (bd board INOUT):
bd := [ (1): 
(a,h):  [.status: occupied, .p : [white,rook]],
(b,g):  [.status: occupied, .p : [white,knight]],
(c,f):  [.status: occupied, .p : [white,bishop]],
(d):  [.status: occupied, .p : [white,queen]],
(e):  [.status: occupied, .p : [white,king]],
(2): [ (ELSE): [.status: occupied, .p : [white,pawn]],
(3:6): [ (ELSE): [.status: free]],
(7): [ (ELSE): [.status: occupied, .p : [black,pawn]],
(8): [ (a,h):  [.status: occupied, .p : [black,rook]],
(b,g):  [.status: occupied, .p : [black,knight]],
(c,f):  [.status: occupied, .p : [black,bishop]],
(d):  [.status: occupied, .p : [black,queen]],
(e):  [.status: occupied, .p : [black,king]] )
];
RETURN;
END initialize;

register_move:
PROC (b board LOC,m move) EXCEPTIONS (illegal);
DCL starting square LOC:= b (m.lin_1)(m.col_1),
arriving square LOC:= b (m.lin_2)(m.col_2);
DO WITH m;
IF starting.status=free THEN CAUSE illegal; FI;
IF arriving.status=free THEN
IF arriving.p.kind=king THEN CAUSE illegal; FI;
FI;
CASE starting.p.kind, starting.p.color OF
(pawn),(white):
IF col_1 = col_2 AND (arriving.status=free
OR NOT (lin_2=lin_1+1 OR lin_2=lin_1+2 AND lin_2=2))
OR (col_2=PRED (col_1) OR col_2=SUCCE (col_1))
AND arriving.status=free THEN CAUSE illegal; FI;
IF arriving.status=/=free THEN
IF arriving.p.color=white THEN CAUSE illegal; FI, FI;
(pawn),(black):
IF col_1 =col_2 AND (arriving.status=free
OR NOT (lin_2=lin_1–1 OR lin_2=lin_1–2 AND lin_1=7))
OR (col_2=PRED (col_1) OR col_2=SUCCE (col_1))
AND arriving.status=free THEN CAUSE illegal; FI;
IF arriving.status=/=free THEN
IF arriving.p.color=black THEN CAUSE illegal; FI, FI;
(rook),(*):
IF NOT ok_rook (b,m)
THEN CAUSE illegal;
FI;
(bishop),(*):
IF NOT ok_bishop (b,m)
THEN CAUSE illegal;
FI;
(queen),(*):
IF NOT ok_rook (b,m) AND NOT ok_bishop (b,m)
THEN CAUSE illegal;
FI;
(knight),(*):
IF ABS ( ABS (NUM (col_2)-NUM (col_1))
    -ABS (lin_2- lin_1)) /= 1
   OR ABS (NUM (col_2)-NUM (col_1))
   +ABS (lin_2- lin_1) =/ 3 THEN CAUSE illegal; FI;
IF arriving.status/=free THEN
   IF arriving.p.color=starting.p.color THEN
      CAUSE illegal; FI; FI;
(king),(*):
IF ABS (NUM (col_2)-NUM (col_1)) > 1
   OR ABS (lin_2- lin_1) > 1
   OR lin_2=lin_1 AND col_2=col_1 THEN CAUSE illegal; FI;
IF arriving.status/=free THEN
   IF arriving.p.color=starting.p.color THEN
      CAUSE illegal; FI; FI;/* checking king moving to check not implemented */
ESAC;
OD:
arriving := starting;
starting := [.status:free];
RETURN;
END register_move;
ok_rook:
PROC (b board,m move) RETURNS (BOOL);
DCL starting square := b (m.lin_1)(m.col_1),
   arriving square := b (m.lin_2)(m.col_2);
DO WITH m;
   IF NOT (col_2=col_1 OR lin_1=lin_2) THEN RETURN FALSE; FI;
   IF arriving.status/=free THEN
      IF arriving.p.color=starting.p.color THEN;
         RETURN FALSE; FI; FI;
   IF col_1=col_2 THEN IF lin_1<lin_2
      THEN DO FOR lin := lin_1+1 TO lin_2–1;
         IF b (lin)(col_1).status/=free
            THEN RETURN FALSE;
      OD;
   ELSE DO FOR lin := lin_1–1 DOWN TO lin_2+1;
      IF b (lin)(col_1).status/=free
         THEN RETURN FALSE;
   OD;
ELSIF col_1<col_2
   THEN DO FOR col := SUCC (col_1) TO PRED (col_2);
      IF b (lin_1)(col).status/=free
         THEN RETURN FALSE;
   FI;
   OD;
ELSIF col_1<col_2
   THEN DO FOR col := SUCC (col_1) TO PRED (col_2);
      IF b (lin_1)(col).status/=free
         THEN RETURN FALSE;
   FI;
   OD;
ELSE DO FOR col := SUCC (col_2) DOWN TO PRED (col_1);
      IF b (lin_1)(col).status/=free
         THEN RETURN FALSE;
   FI;
   OD;
FI;
RETURN TRUE;
OD:
END ok_rook;
ok_bishop:
PROC (b board,m move) RETURNS (BOOL);
ISO-IEC 9496 : 2002(E)

DCL starting square := b (m.lin_1)(m.col_1),
arriving square := b (m.lin_2)(m.col_2),
col column;

DO WITH m;
CASE lin_2>lin_1, col_2>col_1 OF
  (TRUE), (TRUE): col := col_1;
    DO FOR lin := lin_1 +1 TO lin_2-1;
      col := SUCCESS (col);
    IF b (lin)(col).status /= free
      THEN RETURN FALSE;
    FI;
  OD:
  IF SuCC (col) /= col_2
    THEN RETURN FALSE;
  FI;
  (TRUE), (FALSE): col := col_1;
    DO FOR lin := lin_1 +1 TO lin_2-1;
      col := PREDECESS (col);
    IF b (lin)(col).status /= free
      THEN RETURN FALSE;
    FI;
  OD:
  IF PRED (col) /= col_2
    THEN RETURN FALSE;
  FI;
  (FALSE), (TRUE): col := col_1;
    DO FOR lin := lin_1 -1 DOWN TO lin_2+1;
      col := SUCCESS (col);
    IF b (lin)(col).status /= free
      THEN RETURN FALSE;
    FI;
  OD:
  IF SuCC (col) /= col_2
    THEN RETURN FALSE;
  FI;
  (FALSE), (FALSE): col := col_1;
    DO FOR lin := lin_1 -1 DOWN TO lin_2+1;
      col := PREDECESS (col);
    IF b (lin)(col).status /= free
      THEN RETURN FALSE;
    FI;
  OD:
  IF PRED (col) /= col_2
    THEN RETURN FALSE;
  FI;
ESAC;
IF arriving.status /= free THEN RETURN TRUE;
ELSE RETURN arriving.p.color /= starting.p.color; FI;
OD;
END ok_bishop;
END chess_fragments;

IV.12 Building and manipulating a circularly linked list

circular_list:
MODULE handle_list:
MODULE
  GRANT insert, remove, node;
NEWMODE node = STRUCT (pred, suc REF node, value INT);
DCL pool ARRAY (1:1000) node;
DCL head node := (: NULL, NULL, 0 :);
IV.13 A region for managing competing accesses to a resource

allocate_resources:
REGION
   GRANT allocate, deallocate;
   NEWMODE resource_set = INT (0:9);
   DCL allocated ARRAY (resource_set)BOOL := (: (resource_set): FALSE:);
   DCL resource_freed EVENT;
allocate:
PROC () RETURNS (resource_set);
DO FOR EVER:
   DO FOR i IN resource_set;
      IF NOT allocated(i)
         THEN
            allocated(i) := TRUE;
            RETURN i;
      FI;
   OD:
   DELAY resource_freed;
   OD:
END allocate;
deallocate:
PROC (i resource_set);
   allocated(i) := FALSE;
   CONTINUE resource_freed;
END deallocate;
END allocate_resources;
IV.14 Queuing calls to a switchboard

module switchboard:
/* This example illustrates a switchboard which queues incoming calls
and feeds them to the operator at an even rate. Every time the
operator is ready one and only one call is let through. This is
handled by a call distributor which lets calls through at fixed
intervals. If the operator is not ready or there are other calls
waiting, a new call must queue up to wait for its turn. */
DCL operator_is_ready,
switch_is_closed EVENT;
call_distributor:
PROCESS ();
wait:
PROC (x INT);
/* some wait action */
END wait;
DO FOR EVER:
wait(10 /*seconds*/);
CONTINUE operator_is_ready;
OD;
END call_distributor;
call_process:
PROCESS ();
DELAY CASE
(operator_is_ready): /* some actions */;
(switch_is_closed): DO FOR i IN INT (1:100);
CONTINUE operator_is_ready;
/* empty the queue */
OD;
ESAC;
END call_process;
operator:
PROCESS ();
DCL time INT;
DO FOR EVER:
IF time = 1700
THEN CONTINUE switch_is_closed;
FI;
OD;
END operator;
START call_distributor();
START operator();
DO FOR i IN INT (1:100);
START call_process();
OD;
END switchboard;

IV.15 Allocating and deallocating a set of resources

definitions:
MODULE
SIGNAL
acquire,
release=(INSTANCE),
congested,
ISO-IEC 9496 : 2002(E)

/* To illustrate the use of signals and the receive case, (buffers
might have been used instead) we will look at an example where an
allocator manages a set of resources, in this case a set of
counters. The module is part of a larger system where there are
users, that can request the services of the counter_manager. The
module is made to consist of two process definitions, one for the
allocation and one for the counters. Initiate and terminate
are internal signals sent from the allocator
to the counters. All the other signals are external, being sent
from or to the users. */

SEIZE /* external signals */
acquire, release, congested,ready,advance,readout;

SIGNAL initiate = (INSTANCE),
terminate;

allocator:
PROCESS ();
NEWMODE no_of_counters = INT (1:100);
DCL counters ARRAY (no_of_counters)
    STRUCT (counter INSTANCE, status SET (busy,idle));

DO FOR each IN counters;
    each := (: START counter(), idle :);
OD;
DO FOR EVER;
BEGIN
    DCL user INSTANCE;
    await_signals:
    RECEIVE CASE SET user;
    (acquire):
    DO FOR each IN counters;
        DO WITH each;
        IF status = idle
            THEN
                status := busy;
                SEND initiate (user) TO counter;
                EXIT await_signals;
            FI;
        OD;
    SEND congested TO user;
    (release IN this_counter):
    SEND terminate TO this_counter;
    find_counter:
    DO FOR each IN counters;
        DO WITH each;
        IF this_counter = counter
            THEN
                status := idle;
                EXIT find_counter;
            FI;
        OD;
    OD find_counter;
    ESAC await_signals;
END;
OD;
END allocator;

PROCESS ();
DO FOR EVER;
BEGIN
  DCL user INSTANCE,
     count INT := 0;
  RECEIVE CASE
    (initiate IN received_user):
      SEND ready TO received_user;
      user := received_user;
  ESAC;

  work_loop:
  DO FOR EVER;
  RECEIVE CASE
    (advance): count + := 1;
    (terminate):
      SEND readout(count) TO user;
      EXIT work_loop;
  ESAC;

  OD work_loop;
END;
OD;
END counter;
START allocator();
END counter_manager;

IV.16 Allocating and deallocating a set of resources using buffers

user_world:
MODULE
/* This example is the same as no.15 except that buffers are
used for communication instead of signals.
The main difference is that processes are now identified
by means of references to local message buffers rather than
by instance values. There is one message buffer declared
local to each process. There is one set of message types
for each process definition. When started each process must
identify its buffer address to the starting process.
The user_world module sketches some of the environment in
which the counter_manager is used. */
SEIZE allocator;
GRANT user_buffers, user_messages,
   allocator_messages, allocator_buffers,
   counter_messages, counters_buffers;
NEWMODE
user_messages =
   STRUCT (type SET ( congested, ready,
      readout, allocator_id),
   CASE type OF
      (congested) : ,
      (ready) : counter REF counters_buffers,
      (readout) : count INT,
      (allocator_id): allocator REF allocator_buffers
   ESAC),
user_buffers = BUFFER (1) user_messages,
allocator_messages =
   STRUCT (type SET (acquire, release, counter_id),
   CASE type OF
      (acquire) : user REF user_buffers,
(release, counter_id): counter REF counters_buffers

   ESAC,
   allocator_buffers = BUFFER (1) allocator_messages,
   counter_messages =
   STRUCT (type SET (initiate, advance, terminate),
   CASE type OF
   (initiate) : user REF user_buffers,
   (advance, terminate):
   ESAC),
   counters_buffers = BUFFER (1) counter_messages;

DCL user_buffer user_buffers,
   allocator_buf REF allocator_buffers,
   counter_buf REF counters_buffers;

START allocator(–>user_buffer);

RECEIVE CASE
   (user_buffer IN u_msg): allocator_buf := u_msg allocator;

   ESAC;

END user_world;

MODULE

   SEIZE user_buffers, user_messages,
   allocator_messages, allocator_buffers,
   counter_messages, counters_buffers;

GRANT allocator;

 allocators:

PROCESS (starter REF user_buffers):

   DCL allocator_buffer allocator_buffers;
   NEWMODE no_of_counters = INT (1:10);
   DCL counters ARRAY (no_of_counters)
   STRUCT (counter REF counters_buffers,
   status SET (busy, idle)),
   message allocator_messages;

SEND starter–>([allocator_id, –>allocator_buffer]);

DO FOR each IN counters;
   START counter(–>allocator_buffer);
   RECEIVE CASE
   (allocator_buffer IN a_msg): each := [a_msg.counter, idle];

   ESAC;

OD;

DO FOR EVER;

BEGIN
   DCL user REF user_buffers;
   RECEIVE (allocator_buffer IN message);
   handle_messages:
   CASE message.type OF
   (acquire):
   user := message.user;
   DO FOR each IN counters;
     DO WITH each;
       IF status= idle
       THEN status := busy;
       SEND counter–>([initiate, user]);
       EXIT handle_messages;
     FI;

   OD;

   SEND user–>([congested]);

   (release):
   SEND message.counter–>([terminate]);
   find_counter:
DO FOR each IN counters;
DO WITH each;
    IF message.counter = counter
    THEN status := idle;
    EXIT find_counter;
FI;
OD;
OD find_counter;
(counter_id): ;
ESAC handle_messages;
END;
END allocator;

PROCESS (starter REF allocator_buffers);
DCL counter_buffer counters_buffers;
SEND starter–>([counter_id, –>counter_buffer]);
DO FOR EVER;
BEGIN
DCL user REF user_buffers,
    count INT := 0,
    message counter_messages;
RECEIVE (counter_buffer IN message);
CASE message.type OF
  (initiate): user := message.user;
    SEND user–>([ready, –>counter_buffer]);
  ELSE /* some error action */
    ESAC; work_loop:
DO FOR EVER:
RECEIVE (counter_buffer IN message);
CASE message.type OF
  (advance): count + := 1;
  (terminate): SEND user–>([readout, count]);
  EXIT work_loop;
  ELSE /* some error action */
    ESAC;
    OD work_loop;
END;
OD;
END counter;
END counter_manager;

IV.17 String scanner

string_scanner: /* This program implements strings by means
of packed arrays of characters. */

MODULE
SYN
  blanks ARRAY (0:9)CHAR PACK = [(*):' '], linelength = 132;
SYNMODE
  stringptr = ROW ARRAY (lineindex)CHAR PACK,
  lineindex = INT (0:linelength–1);
scanner:
PROC(string stringptr, scanstart lineindex INOUT,
    scanstop lineindex, stopset POWERSET CHAR)
   RETURNS (ARRAY (0:9)CHAR PACK);
DCL count INT := 0,
    res ARRAY (0:9)CHAR PACK := blanks;
DO
  FOR c IN string–>(scanstart:scanstop)
    WHILE NOT (c IN stopset);
IV.18 String scanner2

string_scanner2: /* This example is the same as no.17 but it uses character string instead of packed arrays */

MODULE
SYN
   blanks = (10)" ", linelength = 132;
SYNMODE
   stringptr = ROW CHARs (linelength),
   lineindex = INT (0:linelength–1);

scanner:
PROC(string stringptr, scanstart lineindex INOUT,
   scanstop lineindex, stopset POWERSET CHAR)
   RETURNS (CHARs (10));
DCL count INT:= 0;
DO FOR i := scanstart TO scanstop
   WHILE NOT (string–>(i) IN stopset);
   count + := 1;
OD:
IF count>0
   THEN
      IF count>=10
         THEN
            RESULT string–>(scanstart UP 10);
            ELSE
            RESULT string–>(scanstart:scanstart+count–1)
               //blanks(count:9);
         FI;
      ELSE
      RESULT blanks;
   FI;
   IF scanstart+count < scanstop
      THEN
      scanstart := scanstart+count+1;
      END scanner;
   END scanner;
GRANT scanner;
END string_scanner2;
IV.19 Removing an item from a double linked list

1 queue: MODULE
2 SYNMODE info=INT;
3 queue_removal:
4 MODULE
5 SEIZE info;
6 GRANT remove;
7 remove:
8 PROC (p PTR) RETURNS (info) EXCEPTIONS (EMPTY);
9 /* This procedure removes the item referred to
10  by p from a queue and returns the information
11  contents of that queue element */
12 SYNMODE element = STRUCT (i info POS (0,8:31),
13  prev PTR POS (1,0:15),
14  next PTR POS (1,16:31));
15 DCL x REF element LOC := element(p), prev, next PTR;
16 prev := x–>.prev;
17 next := x–>.next;
18 x–>.prev, x–>.next := NULL;
19 RESULT x–>.i;
20 END remove;
21 END queue_removal;
22 END queue;

IV.20 Update a record of a file

1 read_modify_write:
2 MODULE
3 /* this example indicates how the CHILL i/o concepts can be used */
4 /* to write an application where a record of a random accessible */
5 /* file can be updated or added if not yet in use */
6
7 NEWMODE
8 index_set = INT (1:1000),
9 record_type = STRUCT (free BOOL,
10  count INT,
11  name CHARS (20));
12
13 DCL
14 curindex index_set,
15 file_association ASSOCIATION,
16 record_file ACCESS (index_set) record_type,
17 record_buffer record_type;
18
19 ASSOCIATE (file_association,"DSK:RECORDS.DAT"); /* create association */
20 CONNECT (record_file,file_association,READWRITE); /* connect to file */
21 curindex := 123; /* position record */
22 READRECORD (record_file,curindex,record_buffer); /* read the record */
23 IF record_buffer.free /* if record is free */
24 THEN /* the claim and */
25 record_buffer.free := FALSE /* initialize it */
26 record_buffer.count := 0;
27 record_buffer.name := "CHILL I/O concept ";
FI,
record_buffer.count + := 1; /* increment its count */
WRITERECORD (record_file, curindex, record_buffer); /* write the record */
DISSOCIATE (file_association); /* end the association */
END read_modify_write;

IV.21 Merge two sorted files

MODULE

/* this example shows how two sorted files can be merged into one */
/* new sorted file, where the field 'key' is used for sorting */
/* the old sorted files are deleted after the merging has been done */

NEWMODE
record_type = STRUCT {
   key INT,
   name CHARS (50));

DCL
flag BOOL,
infiles ARRAY (BOOL) ACCESS record_type,
outfile ACCESS record_type,
buffers ARRAY (BOOL) record_type,
innames ARRAY (BOOL) CHARS (10) INIT:=["FILE.IN.1 ","FILE.IN.2 "],
outname CHARS (10) INIT:="FILE.OUT ",
inassoc ARRAY (BOOL) ASSOCIATION,
outassoc ASSOCIATION;

/* associate both sorted input files, connect an access to them for input */
/* and read their first record into a buffer */

DO
FOR curfile IN infiles,
   curbuffer IN buffers,
   curassoc IN inassoc,
   curname IN innames;
CONNECT (curfile, ASSOCIATE (curassoc,curname), READONLY);
READRECORD (curfile, curbuffer);
OD;

/* associate the output file, create a file for the association */
/* and connect an access to it for output */
ASSOCIATE (outassoc,outname);
CREATE (outassoc);
CONNECT (outfile, outassoc, WRITEONLY);
merge_files:

DO FOR EVER
/* determine which file, if any at all, to process next */
/* 'flag' indicates the file */

CASE OUTOFFILE (infiles(FALSE)),OUTOFFILE (infiles(TRUE)) OF
   (TRUE), (TRUE): /* both files are empty */
      EXIT merge_files;
   (TRUE), (FALSE): /* one file is empty */
      flag := TRUE;
   (FALSE), (TRUE): /* one file is empty */
      flag := FALSE;
   (FALSE), (FALSE): /* both files are empty */
   (TRUE), (FALSE): /* one file is empty */
   (FALSE), (TRUE): /* one file is empty */
   (FALSE), (FALSE): /* both files are empty */
IV.22 Read a file with variable length records

```plaintext
variable_length_records:

MODULE

/* This example shows how a file which consists of variable length */
/* records can be treated. */
/* The file consists of a number of strings of varying length; the */
/* algorithm will read a string, allocate an appropriate location */
/* for it, and put the reference to this location into a push down list */

NEWMODE
string = CHARS (80),
link_record = STRUCT /
    next_record    REF link_record,
    string_row    ROW string;

DCL
pushdownlist    REF link_record INIT := NULL,
length         INT (1:80),
temporaryrow   ROW string,
fileaccess     string DYNAMIC,
association    ASSOCIATION;
filename      CHARS (20) VARYING INIT := "INPUT.DATA";
ASSOCIATE (association, filename); /* associate the input file */
CONNECT (fileaccess, association, READONLY); /* connect access for input */
temporaryrow := READRECORD (fileaccess); /* read the first record */
DO
    WHILE NOT (OUTOFFILE (fileaccess));
        pushdownlist := ALLOCATE (link_record,
                        [pushdownlist, NULL]); /* get a new link record */
                        /* and initialize it */
        length := 1 + UPPER (temporaryrow->); /* determine length of string */
    OD
    WITH pushdownlist->;
        string_row := ALLOCATE (CHARS (length),
                            temporaryrow->); /* allocate space for string */
                            /* and fill it */
    OD
temporaryrow := READRECORD (fileaccess); /* get next record in file */
OD
DISSOCIATE (association); /* end the association */
END variable_length_records;
```
IV.23 The use of spec modules

/* The examples 23 and 24 are example 8 divided in two pieces. */
letter_count:

SPEC MODULE
/* This is a spec module for the corresponding module in example 8. */
SEIZE max;
count:
PROC (input ROW CHARS (max) IN, output ARRAY ('A':'Z') INT OUT) END;
GRANT count;
END letter_count;
letter_count: REMOTE "example 24";
test:

MODULE
/* This is the module 'test' from example 8. */
/* It can now be piecwise compiled together with */
/* the above spec module */
SYNMODE results = ARRAY ('A':'Z') INT;
DCL c CHARS (10) INIT:="A-B<ZAA9K" ;
DCL output results;
SYN max = 10_000;
GRANT max;
SEIZE count;
count (–> c, output);
ASSERT output = results [(\'A\') : 3, (\'B\', \'K\', \'Z\') : 1, (ELSE) : 0 ];
END test;

IV.24 Example of a context

CONTEXT
/* This is a context for the module "letter_count" */
/* as used in example 23, allowing the piecewise */
/* compilation of "letter_count" */
SYN max = 10_000;
FOR
letter_count:

MODULE
SEIZE max;
DCL letter POWERSET CHAR INIT:=[\'A\':\'Z\'];
count:
PROC (input ROW CHARS (max) IN, output ARRAY ('A':'Z') INT OUT);
output := [(ELSE) : 0 ];
DO FOR i := 0 TO UPPER (input ->);
IF input -> (i) IN letter THEN
output (input -> (i)) + := 1;
FI;
OD;
END count;
GRANT count;
END letter_count;

IV.25 The use of prefixing and remote modules

/* This example uses the module 'stack' from example 27 or 28. */
/* It shows how prefixes can be used to prevent name clashes. */
/* It uses the remote construct to share the source code. */
char_stack:

MODULE
SYNMODE element = CHAR;
GRANT (–> stack ! char) ! ALL;
stack: SPEC REMOTE "example 29";
IV.26 The use of text i/o

```plaintext
 MODULE

 /* This example shows the use of the text i/o features. */

 DCL outfile ASSOCIATION,
 output TEXT (80) DYNAMIC,
 size INT:= 12345,
 flag BOOL:= FALSE,
 set SET (a,b,c) := b,
 s1 CHAR (5) := "CHILL",
 s2 CHAR (5) VARYING := "text":

 ASSOCIATE (outfile,"OUTPUT.DATA");   - - associate the output file
 CREATE (outfile);   - - create it
 CONNECT (output,outfile,WRITEONLY); - - then connect text location
 WRITETEXT (output,"%B%",10);   - - 1010
 WRITETEXT (output,"%C%",set);   - - b
 WRITETEXT (output,"size = %C%",size);   - - size = 12345
 WRITETEXT (output,"%CL6%C i/o%",s1,s2);   - - CHILL text i/o
 WRITETEXT (output,"flag =%X%C",flag);   - - flag = FALSE
 size := GETTEXTINDEX (output);  - - 12
 DISSOCIATE (outfile);
 END textio;
```
A generic stack

/* This example implements a generic stack. Please note that the element mode has been left out. */
/* The element mode is defined in the surroundings. */
/* The context is a virtually introduced context, and it has no source. */

CONTEXT REMOTE FOR stack:

MODULE
  SEIZE element;
  NEWMODE cell = STRUCT (pred,succ REF cell,info element);
  DCL p,last,first REF cell INIT := NULL;

  push:
  PROC (e element) EXCEPTIONS (overflow)
  p := ALLOCATE (cell) ON (ALLOCATEFAIL): CAUSE overflow; END;
  IF last = NULL THEN
    first := p;
    last := p;
  ELSE
    last -> .succ := p;
    p -> .pred := last;
    last := p;
  FI;
  last -> .info := e;
  RETURN;
  END push;

  pop:
  PROC () EXCEPTIONS (underflow)
  IF last = NULL THEN
    CAUSE underflow;
  FI;
  p := last;
  last := last -> .pred;
  IF last = NULL THEN
    first := NULL;
  ELSE
    last -> .succ := NULL;
  FI;
  TERMINATE (p);
  RETURN;
  END pop;

  elem:
  PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds)
  IF first = NULL THEN
    CAUSE bounds;
  FI;
  p := first;
  DO FOR j := 2 TO i:
    IF p -> .succ = NULL THEN
      CAUSE bounds;
    FI;
  OD;
  RETURN p -> .info;
  END elem;

  GRANT push,pop,elem;
END stack;
IV.28 An abstract data type

/* This example implements a stack with the same functionality */
/* of example 27, demonstrating how an abstract data type */
/* can be implemented in two different ways in CHILL. */

CONTEXT REMOTE FOR

stack:
MODULE
SEIZE element;
SYN max = 10_000, min = 1;
DCL stack ARRAY (min : max) element,
stackindex INT INIT := min-1;
push:
PROC (e element) EXCEPTIONS (overflow)
IF stackindex = max THEN
CAUSE overflow;
FI;
stackindex +:= 1;
stack(stackindex) := e;
RETURN;
END push;
pop:
PROC () EXCEPTIONS (underflow)
IF stackindex = min THEN
CAUSE underflow;
FI;
stackindex–:= 1;
RETURN;
END pop;

elem:
PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds)
IF i < min OR i > max THEN
CAUSE bounds;
FI;
RETURN stack(i);
END elem;
GRANT push,pop,elem;
END stacks;

IV.29 Example of a spec module

/* This SPEC MODULE defines the interface of examples 27 and 28. */
stack: SPEC MODULE
SEIZE: element;
push: PROC (e element) EXCEPTIONS (overflow) END;
pop: PROC () EXCEPTIONS (underflow) END;
 elem: PROC (i INT) RETURNS (element LOC) EXCEPTIONS (bounds) END;
GRANT push,pop,elem;
END stack;

IV.30 Object-Orientation: Modes for Simple, Sequential Stacks

/* The examples show the application of object-orientation to the well known stack data structure.
Two different implementations of stack modes with identical interfaces are realized (Example 30).
Based on these modes extended modes with an additional operation (e.g. Top; Example 31) or
with other properties (e.g. mutual exclusive access to stacks (Example 32)) are realized. */

SYNMODE StackModel = MODULE SPEC /* -------------- Definition of the interface */
GRANT ElementMode, Push, Pop; /* Simple, sequential stack */

NEWMODE ElementMode = STRUCT (a INT, b BOOL);

Push: PROC(Elem ElementMode IN) EXCEPTIONS(Overflow) END Push;

Pop: PROC() RETURNS(ElementMode) EXCEPTIONS(Underflow) END Pop;

SYN Length = 10_000;

DCL StackData ARRAY (1:Length) ElementMode, /* Array implementation*/
TopOfStack RANGE(0:Length) INIT := 0; /* of the stack */

END StackMode1;

SYNMODE StackMode1 = MODULE BODY /* ------------------ Definition of the body */
Push: PROC(Elem ElementMode IN) EXCEPTIONS(Overflow)

IF TopOfStack = Length THEN
  CAUSE Overflow;
ELSE
  TopOfStack +:= 1;
  StackData(TopOfStack) := Elem;
FI;

END Push;

Pop: PROC() RETURNS(ElementMode) EXCEPTIONS(Underflow)

IF TopOfStack = 0 THEN
  CAUSE Underflow;
ELSE
  RESULT(StackData(TopOfStack));
  TopOfStack -=: 1;
FI;

END Pop;

END StackMode1;

MainProgram1: MODULE

SEIZE StackMode1;

DCL Stack1 StackMode1;

DCL Elem1 StackMode1!ElementMode;
Elem1 := [10, TRUE];
Stack1.Push(Elem1);
Stack1.Push( [20, FALSE] );

END MainProgram1;

SYNMODE StackMode2 = MODULE SPEC /* ----------------- Definition of the interface */
GRANT ElementMode, Push, Pop; /* Same interface as StackMode1 */

NEWMODE ElementMode = STRUCT (a INT, b BOOL);
Push: PROC(Elem ElementMode IN) EXCEPTIONS(Overflow) END Push;

Pop: PROC() RETURNS(ElementMode) EXCEPTIONS(Underflow) END Pop;

NEWMODE ListElem = STRUCT (next REF ListElem, /* List implementation */
  info ElementMode); /* of the stack */

DCL Stack REF ListElem INIT := NULL;

END StackMode2;

SYNMODE StackMode2 = MODULE BODY /* ---------------------- Definition of the body */
Push: PROC(Elem ElementMode IN) EXCEPTIONS(Overflow)

Stack := ALLOCATE (ListElem, [Stack, Elem])
  ON (ALLOCATEFAIL) : CAUSE Overflow; END;

END Push;

Pop: PROC() RETURNS(ElementMode) EXCEPTIONS(Underflow)

DCL Temp REF ListElem;

IF Stack = NULL THEN
  CAUSE Underflow;
ELSE
  RESULT (Stack->.info);
  Temp := Stack;
  Stack := Stack->.next;
END

TERMINATE (Temp);
IV.31 Object-Orientation: Mode Extension: Simple, Sequential Stack with Operation "Top"

```plaintext
SYNMODE StackWithTopMode2 = MODULE SPEC /* BASED_ON indicates */
  BASED_ON StackMode2 /* mode derivation or */
  GRANT Top; /* inheritance */
  Top: PROC( ) RETURNS (ElementMode) /* Top is an additional operation */
  EXCEPTIONS (EmptyStack) END Top;
END StackWithTopMode2;
```

```plaintext
SYNMODE StackWithTopMode2 = MODULE BODY BASED_ON StackMode2
  Top: PROC( ) RETURNS (ElementMode) EXCEPTIONS (EmptyStack)
  IF Stack = NULL THEN
    CAUSE EmptyStack;
  ELSE
    RETURN (Stack-->info);
  FI;
END Top;
END StackWithTopMode2;
```

IV.32 Object-Orientation: Modes for Stacks with Access Synchronization

```plaintext
/* Based on the mode StackWithTopMode2 defined in example 31 the mode
  RegionStackWithTopMode1 is defined whose objects behave like regions:
  at any point in time at most one of the public procedures may be in execution.
  Apart from this the behavior is essentially the same as for StackWithTopMode2:
  erroneous use of an object causes an exception. The second mode
  RegionStackWithTopMode2 uses the CHILL event mechanism to deal with
  erroneous use of a stack object. */
SYNMODE RegionStackWithTopMode1 = REGION SPEC BASED_ON StackWithTopMode2
  /* Just put the base mode into a "region envelope" */
  /* In case of an erroneous use same behaviour as StackWithTopMode2: cause an exception */
END RegionStackWithTopMode1;
```

```plaintext
SYNMODE RegionStackWithTopMode1 = REGION BODY BASED_ON StackWithTopMode2
END RegionStackWithTopMode1;
```
MainProgram4: MODULE
  SEIZE RegionStackWithTopMode1;
  DCL Stack1 RegionStackWithTopMode1;
  Producer: PROCESS ( ) ;
    DCL Elem1 RegionStackWithTopMode1!ElementMode;
    DO FOR EVER
      /* compute Elem1 */
      Stack1.Push(Elem1);
    OD;
  END Producer;
  Consumer: PROCESS ( ) ;
    DCL Elem1 RegionStackWithTopMode1!ElementMode;
    DO FOR EVER
      Elem1 := Stack1.Pop ( );
      /* process Elem1 */
    OD;
  END Consumer;
  START Producer ( );
  START Consumer ( );
  END MainProgram4;

SYNMODE RegionStackWithTopMode2 = REGION SPEC BASED_ON StackWithTopMode2
  /* In case of an erroneous use different behaviour as StackWithTopMode2:
     use the event mechanism */
  GRANT Push, Pop, Top;
  Push: PROC(Elem ElementMode IN) REIMPLEMENT END Push;
  Pop: PROC() RETURNS (ElementMode) REIMPLEMENT END Pop;
  Top: PROC() RETURNS (ElementMode) REIMPLEMENT END Top;
  DCL NotEmpty, NotFull EVENT;
  END RegionStackWithTopMode2 ;

SYNMODE RegionStackWithTopMode2 = REGION BODY BASED_ON StackWithTopMode2
  Push: PROC(Elem ElementMode IN) REIMPLEMENT
    PushLoop: DO
    BEGIN
      StackWithTopMode2!Push(Elem);
      EXIT PushLoop;
    END
    ON (Overflow): DELAY NotFull; END;
    OD PushLoop;
    CONTINUE NotEmpty;
    END Push;
  Pop: PROC() RETURNS(ElementMode) REIMPLEMENT
    PopLoop: DO
    BEGIN
      RESULT StackWithTopMode2!Pop ( );
      EXIT PopLoop;
    END
    ON (Underflow): DELAY NotEmpty; END;
    OD PopLoop;
    CONTINUE NotFull;
    END Pop;
  Top: PROC() RETURNS (ElementMode) REIMPLEMENT
    TopLoop: DO
    BEGIN
      RESULT StackWithTopMode2!Top();
      EXIT TopLoop;
    END
    ON (EmptyStack): DELAY NotEmpty; END;
    OD TopLoop;
    CONTINUE NotFull;
    END Top;
MainProgram5: MODULE /* -------------------------- Essentially the same as MainProgram4 */
  SEIZE RegionStackWithTopMode2;
  DCL Stack1 RegionStackWithTopMode2;
  Producer: PROCESS ( ) ;
    DCL Elem1 RegionStackWithTopMode2!ElementMode;
    DO FOR EVER
      /* compute Elem1 */
      Stack1.Push(Elem1);
    OD;
  END Producer;
  Consumer: PROCESS ( ) ;
    DCL Elem1 RegionStackWithTopMode2!ElementMode;
    DO FOR EVER
      Elem1 := Stack1.Pop (Elem1);
      /* process Elem1 */
    OD;
  END Consumer;
START Producer ( );
START Consumer ( );
END MainProgram5;

Appendix V

Decommitted features

The features described in this appendix are not part of this Recommendation, but were part of the Recommendation Z.200, (1984), Red Book, Volume VI – Fascicle VI.12 and Recommendation Z.200, (1988), Blue Book, Volume X – Fascicle X.6. In the following, a brief description is given; for a complete definition of them, refer to the relevant subclauses of Recommendation Z.200 (1984), that are hereafter mentioned. These features may be supported by an implementation. If no indication is given, the references are made to Recommendation Z.200 (1984).

V.1 Free directive (subclause 2.6)
A free directive freed the reserved simple name strings specified in the reserved simple name string list so that they could be redefined.

V.2 Integer modes syntax (subclause 3.4.2)
BIN was derived syntax for INT.

V.3 Set modes with holes (subclause 3.4.5)
A set mode defined a set of named or unnamed values. A set mode was a set mode with holes, if and only if the number of its set element names was less than the number of values of the set mode.

V.4 Procedure modes syntax (subclause 3.7)
A result spec without the optional reserved simple name string RETURNS was derived syntax for the result spec with RETURNS.
V.5 String modes syntax (subclause 3.11.2)
The notation $\text{CHAR} \ (n)$ and $\text{BIT} \ (n)$ denoted character strings and bit strings respectively.

V.6 Array modes syntax (subclause 3.11.3)
The reserved simple name string ARRAY was optional.

V.7 Level structure notation (subclause 3.11.5)
A level structure mode was derived syntax for a nested structure mode. In the level structure notation the fields were preceded by a level number. If a structure contained fields that were themselves structures or arrays of structures, a hierarchy of structures was formed and a level number could be associated with each field. Instead of writing nested structure modes, it was allowed in the level structure mode to write the level number in the front of the field name.

V.8 Map reference names (subclause 3.11.6)
Map reference names could be used to specify mapping in an implementation defined way.

V.9 Based declarations (subclause 4.1.4)
A based declaration without a bound or free reference location name was derived syntax for a synmode definition statement. A based declaration with a bound or free reference location name defined one or more access names. These names served as an alternative way of accessing a location by dereferencing the reference value contained in the specified reference location. This dereferencing operation was performed each time and only when an access was made via a declared based name.

V.10 Character string literals (subclause 5.2.4.6)
Character string literals were delimited by apostrophe characters. Apart from the printable representation, the hexadecimal representation could be used. Character string literals of length one served as character literals.

V.11 Receive expressions (see 5.3.9/Z.200 (1988))
Receive expressions were used to receive values from buffer locations. The executing process could become delayed and could re-activate another process, delayed on sending a value to the specified buffer location.

V.12 Addr notation (subclause 5.3.8)
ADDR ($<\text{location}>$) was derived syntax for $\rightarrow <\text{location}>$. 

V.13 Assignment syntax (subclause 6.2)
The $=$ symbol was derived syntax for the $:= \text{symbol}$. 

V.14 Case action syntax (subclause 6.4)
The range list of a case action could be specified more generally by a discrete mode, and not only by a discrete mode name.

V.15 Do for action syntax (subclause 6.5.2)
The range in the range enumeration of a do-for action could be specified more generally by a discrete mode, and not only by a discrete mode name.
V.16  Explicit loop counters (subclause 6.5.2)

If an access name was visible in the reach where the do action was placed, which was equal to one of the names defined by a loop counter, then the loop counter was explicit; otherwise it was implicit. In the former case, the value of the loop counter was stored into the denoted location just prior to abnormal termination. A distinction was made between normal and abnormal termination. Normal termination occurred if the evaluation of at least one of the loop counters indicated termination. Abnormal termination occurred if the evaluation of while condition delivered FALSE or if the do action was left by a transfer of control out of it.

V.17  Call action syntax (subclause 6.7)

The reserved simple name string CALL was optional. A call action with CALL was derived from a call action without CALL.

V.18  RECURSEFAIL exception (subclause 6.7)

The RECURSEFAIL exception was caused when a non-recursive procedure called itself recursively.

V.19  Start action syntax (subclause 6.13)

The start action with the SET option was derived syntax for the single assignment action:

\(<\text{instance location}>: := <\text{start expression}>.\)

V.20  Explicit value receive names (subclause 6.19)

A receive signal case action and a receive buffer case action could introduce value receive names. If a name was visible in the reach where the receive signal case action was placed, which was equal to one of the names introduced after IN, then the value receive name was explicit; otherwise it was implicit. In the former case, the received value was stored into the denoted location immediately before the execution of the action statement list.

V.21  Blocks (subclause 8.1)

The if action, case action, do action and delay case action were not defined to be blocks.

V.22  Entry statement (subclause 8.4)

A procedure could have multiple entry points by means of entry statements. These statements were considered to be additional procedure definitions. The defining occurrence in the entry statement defined the name of the entry point in the procedure in which reach it was placed. The entry point was determined by the textual position of the entry statement.

V.23  Register names (subclause 8.4)

Register specification could be given in the formal parameter of the procedure and in the result spec. In the pass by value case, it meant that the actual value was contained in the specified register; in the pass by location case, it meant that the (hidden) pointer to the actual location was contained in the specified register. If the specification was in the result spec it meant that the returned value or the (hidden) pointer to the returned location was contained in the specified register.

V.24  Recursive attribute (10.4/Z.200 (1988))

The recursivity of procedures was an implementation default, unless the attribute RECURSIVE was specified in a procedure attribute list.
V.25 Quasi cause statements and quasi handlers (subclause 8.10.3)

Quasi cause statements indicated the presence of cause statements in remote modules or remote regions directly enclosed in the reach of the spec module or spec region in which the quasi cause statement was placed. Quasi handlers indicated the presence of a handler in the program, reachable from the module, region or context directly enclosed in the context to which the quasi handler was appended.

V.26 Syntax of quasi statements (10.10.3/Z.200 (1988))

Quasi procedure and process definition statements were terminated by an END <simple name string>.

V.27 Weakly visible names and visibility statements (12.2.1/Z.200 (1988))

A name string which was not strongly visible in a reach was said to be weakly visible in it if it was implied by a name string which was strongly visible in the reach. The name string in the reach was linked to implied defining occurrences. If they did not define the same set element of similar set modes, a weak clash occurred, otherwise the name string was bound to them. Subclause 12.2.4 defined the implied defining occurrences for names.

V.28 Weakly visible names and visibility statements (subclause 10.2.4.3)

A name string NS weakly visible in reach R was said to be seizable by modulion M directly enclosed in R if NS was linked in R to a defining occurrence not surrounded by the reach of M. A name string NS weakly visible in reach R of modulion M was said to be grantable by M if NS was linked in R to a defining occurrence surrounded by R.

V.29 Pervasiveness (subclause 10.2.4.4)

When a grant statement contained (DIRECTLY) PERVASIVE, all name strings granted by it had the (directly) pervasive property in the surrounding reaches of the modulion M that directly enclosed the grant statement. The name strings:

- were strongly visible in a directly surrounding reach S of M;
- in case the name strings had the directly pervasive property in S, they had also the directly pervasive property in M;
- if they were not directly strongly visible in a reach R and were strongly visible in a reach that directly enclosed R and where they had the pervasive property, then they were indirectly strongly visible in R and had also the pervasive property in R.

V.30 Seizing by modulion name (subclause 10.2.4.5)

If a prefix rename clause in a seize statement had a seize postfix which contained a modulion name string and ALL, then the prefix rename clause was equivalent to a set of seize statements, for any name string that was strongly visible in the reach that directly enclosed the modulion in which the seize statement was placed and was seizable by this modulion, and was granted by the modulion attached to the modulion name in the reach directly enclosing the modulion in which the seize statement was placed.

V.31 Predefined simple name strings (subclause C.2)

AND, NOT, OR, REM, MOD, THIS and XOR were predefined simple name strings.
## Appendix VI

### Index of production rules

Page numbers in boldface are references to the defining occurrences of an item; normal font refers to applied occurrences of indexed items.

<table>
<thead>
<tr>
<th>A</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute time built-in routine call</td>
<td>124</td>
</tr>
<tr>
<td>absolute time mode</td>
<td>28</td>
</tr>
<tr>
<td>absolute time mode name</td>
<td>28</td>
</tr>
<tr>
<td>absolute time primitive value</td>
<td>123, 125</td>
</tr>
<tr>
<td>absolute timing action</td>
<td>9.3.2 122, 123</td>
</tr>
<tr>
<td>access attr built-in routine call</td>
<td>7.4.8 105, 109</td>
</tr>
<tr>
<td>access location</td>
<td>98, 107, 110, 120</td>
</tr>
<tr>
<td>access mode</td>
<td>26</td>
</tr>
<tr>
<td>access mode name</td>
<td>26, 98</td>
</tr>
<tr>
<td>access name</td>
<td>4.2.2 47, 48</td>
</tr>
<tr>
<td>action</td>
<td>79</td>
</tr>
<tr>
<td>action statement</td>
<td>6.1 79, 127</td>
</tr>
<tr>
<td>action statement list</td>
<td>81, 82, 83, 92, 95, 96, 121, 122, 123, 127</td>
</tr>
<tr>
<td>actual generic parameter</td>
<td>10.11 142, 143</td>
</tr>
<tr>
<td>actual generic parameter list</td>
<td>142</td>
</tr>
<tr>
<td>actual generic procedure</td>
<td>143</td>
</tr>
<tr>
<td>actual parameter</td>
<td>87</td>
</tr>
<tr>
<td>actual parameter list</td>
<td>38, 69, 87</td>
</tr>
<tr>
<td>allocate built-in routine call</td>
<td>6.20.4 97, 101</td>
</tr>
<tr>
<td>alternative field</td>
<td>32</td>
</tr>
<tr>
<td>arithmetic additive operator</td>
<td>74, 80</td>
</tr>
<tr>
<td>arithmetic multiplicative operator</td>
<td>5.3.7 75, 76, 80</td>
</tr>
<tr>
<td>array element</td>
<td>4.2.8 47, 51</td>
</tr>
<tr>
<td>array expression</td>
<td>84, 98</td>
</tr>
<tr>
<td>array location</td>
<td>51, 52, 84, 98</td>
</tr>
<tr>
<td>array mode</td>
<td>3.13.3 23, 29, 30</td>
</tr>
<tr>
<td>array mode name</td>
<td>30, 98</td>
</tr>
<tr>
<td>array primitive value</td>
<td>66</td>
</tr>
<tr>
<td>array slice</td>
<td>4.2.9 47, 52</td>
</tr>
<tr>
<td>array tuple</td>
<td>61</td>
</tr>
<tr>
<td>assert action</td>
<td>6.10 79, 91</td>
</tr>
<tr>
<td>assertion part</td>
<td>130</td>
</tr>
<tr>
<td>assigning operator</td>
<td>79</td>
</tr>
<tr>
<td>assignment action</td>
<td>79</td>
</tr>
<tr>
<td>assignment symbol</td>
<td>6.2 46, 47, 79, 80, 83</td>
</tr>
<tr>
<td>associate built-in routine call</td>
<td>105</td>
</tr>
<tr>
<td>associate parameter</td>
<td>105</td>
</tr>
<tr>
<td>associate parameter list</td>
<td>105</td>
</tr>
<tr>
<td>association attr built-in routine call</td>
<td>7.4.4 105, 106</td>
</tr>
<tr>
<td>association location</td>
<td>105, 106, 107</td>
</tr>
<tr>
<td>association mode</td>
<td>26</td>
</tr>
<tr>
<td>association mode name</td>
<td>26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin-end block</td>
<td>10.3 79, 129</td>
</tr>
<tr>
<td>begin-end body</td>
<td>10.2 127, 129</td>
</tr>
<tr>
<td>binary bit string literal</td>
<td>61</td>
</tr>
<tr>
<td>binary integer literal</td>
<td>57</td>
</tr>
<tr>
<td>bit string literal</td>
<td>5.2.4.9 56, 61</td>
</tr>
<tr>
<td>boolean expression</td>
<td>39, 71, 81, 86, 91, 130</td>
</tr>
<tr>
<td>boolean literal</td>
<td>56</td>
</tr>
<tr>
<td>boolean mode</td>
<td>17</td>
</tr>
<tr>
<td>Term</td>
<td>Page(s)</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>boolean mode name</td>
<td>17</td>
</tr>
<tr>
<td>bound reference mode</td>
<td>22</td>
</tr>
<tr>
<td>bound reference mode name</td>
<td>22</td>
</tr>
<tr>
<td>bound reference moreta location</td>
<td>87</td>
</tr>
<tr>
<td>primitive value</td>
<td></td>
</tr>
<tr>
<td>bound reference primitive value</td>
<td>48</td>
</tr>
<tr>
<td>bracketed action</td>
<td>79</td>
</tr>
<tr>
<td>bracketed comment</td>
<td>9</td>
</tr>
<tr>
<td>buffer element mode</td>
<td>25</td>
</tr>
<tr>
<td>buffer length</td>
<td>25</td>
</tr>
<tr>
<td>buffer location</td>
<td>94, 96, 98</td>
</tr>
<tr>
<td>buffer location</td>
<td>96</td>
</tr>
<tr>
<td>buffer mode</td>
<td>25</td>
</tr>
<tr>
<td>buffer mode name</td>
<td>25, 98</td>
</tr>
<tr>
<td>buffer receive alternative</td>
<td>96</td>
</tr>
<tr>
<td>built-in routine call</td>
<td>87</td>
</tr>
<tr>
<td>built-in routine name</td>
<td>87</td>
</tr>
<tr>
<td>built-in routine parameter</td>
<td>87</td>
</tr>
<tr>
<td>built-in routine parameter list</td>
<td>87</td>
</tr>
</tbody>
</table>

**C**

<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>call action</td>
<td>6.7, 79, 87</td>
</tr>
<tr>
<td>case action</td>
<td>6.4, 79, 81</td>
</tr>
<tr>
<td>case alternative</td>
<td>6.4, 81, 82</td>
</tr>
<tr>
<td>case label</td>
<td>167</td>
</tr>
<tr>
<td>case label list</td>
<td>62, 167</td>
</tr>
<tr>
<td>case label specification</td>
<td>12.3, 32, 71, 82, 167</td>
</tr>
<tr>
<td>case selector list</td>
<td>6.4, 71, 81, 82</td>
</tr>
<tr>
<td>cause action</td>
<td>6.12, 79, 91</td>
</tr>
<tr>
<td>character</td>
<td>9, 59, 115</td>
</tr>
<tr>
<td>character literal</td>
<td>56</td>
</tr>
<tr>
<td>character mode</td>
<td>3.4.4, 17, 18</td>
</tr>
<tr>
<td>character mode name</td>
<td>18</td>
</tr>
<tr>
<td>character string</td>
<td>9</td>
</tr>
<tr>
<td>character string expression</td>
<td>113</td>
</tr>
<tr>
<td>character string literal</td>
<td>5.2.4.8, 56, 60, 137</td>
</tr>
<tr>
<td>character string location</td>
<td>113, 120</td>
</tr>
<tr>
<td>CHILL built-in routine call</td>
<td>6.20, 97</td>
</tr>
<tr>
<td>CHILL location built-in routine call</td>
<td>97</td>
</tr>
<tr>
<td>CHILL simple built-in routine call</td>
<td>97</td>
</tr>
<tr>
<td>CHILL value built-in routine call</td>
<td>97</td>
</tr>
<tr>
<td>clause width</td>
<td>115, 118</td>
</tr>
<tr>
<td>closed dyadic operator</td>
<td>6.2, 79, 80</td>
</tr>
<tr>
<td>comment</td>
<td>2.4, 9</td>
</tr>
<tr>
<td>common module component</td>
<td>39, 41, 45</td>
</tr>
<tr>
<td>component name</td>
<td>2.7, 11</td>
</tr>
<tr>
<td>component name defining occurrence</td>
<td>2.7, 11</td>
</tr>
<tr>
<td>composite mode</td>
<td>3.13.1, 16, 29</td>
</tr>
<tr>
<td>composite object</td>
<td>84</td>
</tr>
<tr>
<td>conditional expression</td>
<td>71</td>
</tr>
<tr>
<td>connect built-in routine call</td>
<td>7.4.6, 105, 107</td>
</tr>
<tr>
<td>constant value</td>
<td>46, 54, 139</td>
</tr>
<tr>
<td>constructor actual parameter list</td>
<td>46, 101</td>
</tr>
<tr>
<td>context</td>
<td>138</td>
</tr>
<tr>
<td>context body</td>
<td>10.2, 127, 136, 138</td>
</tr>
<tr>
<td>context list</td>
<td>134, 135, 138, 141</td>
</tr>
<tr>
<td>context module</td>
<td>10.10.1, 79, 137</td>
</tr>
<tr>
<td>continue action</td>
<td>6.15, 79, 92</td>
</tr>
<tr>
<td>control code</td>
<td>7.5.4, 114, 115</td>
</tr>
<tr>
<td>control part</td>
<td>83</td>
</tr>
<tr>
<td>control sequence</td>
<td>59, 60</td>
</tr>
<tr>
<td>conversion clause</td>
<td>115</td>
</tr>
<tr>
<td>ISO-IEC 9496 : 2002(E)</td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td></td>
</tr>
<tr>
<td>conversion code</td>
<td>115</td>
</tr>
<tr>
<td>conversion qualifier</td>
<td>115</td>
</tr>
<tr>
<td>cyclic timing action</td>
<td>9.3.3 122, 123</td>
</tr>
</tbody>
</table>

**D**
- data statement                  | 127 |
- data statement list              | 127 |
- day expression                   | 124 |
- day location                     | 125 |
- decimal integer literal          | 57  |
- declaration                      | 45  |
- declaration statement            | 39, 41, 45, 127 |
- defining mode                    | 13  |
- defining occurrence              | 2.7 10, 79, 83, 96, 129, 130, 134, 135, 139, 140, 141, 148 |
- defining occurrence list         | 2.7 10, 13, 45, 47, 54, 95, 130, 139, 142, 143 |
- definition statement             | 127 |
- delay action                     | 6.16 79, 92 |
- delay alternative                | 92  |
- delay case action                | 6.17 79, 92 |
- dereferenced bound reference     | 4.2.3 47, 48 |
- dereferenced free reference      | 4.2.4 47, 49 |
- dereferenced row                  | 4.2.5 47, 49 |
- digit                            | 8, 57, 114, 115 |
- digit sequence                   | 57, 58 |
- directive                        | 10  |
- directive clause                 | 2.6 10 |
- disconnect built-in routine call  | 7.4.7 105, 109 |
- discrete expression              | 82, 83, 84, 97, 98, 113 |
- discrete literal expression      | 19, 30, 32, 167 |
- discrete location                | 98, 113 |
- discrete mode                    | 3.4.1 16, 17, 22, 26 |
- discrete mode name               | 19, 82, 84, 98, 167 |
- discrete range mode              | 3.4.6 17, 19 |
- discrete range mode name         | 19  |
- dissociate built-in routine call  | 7.4.3 105, 106 |
- do action                        | 6.5.1 79, 83 |
- duration built-in routine call    | 124 |
- duration mode                    | 28  |
- duration mode name               | 28  |
- duration primitive value          | 122, 123 |

**E**
- editing clause                  | 7.5.6 115, 118 |
- editing code                     | 118 |
- element layout                   | 3.13.5 30, 35 |
- element mode                     | 30  |
- else alternative                 | 71  |
- else clause                      | 81  |
- emptiness literal                | 5.2.4.7 56, 60 |
- emptiness literal name           | 60  |
- empty                            | 91, 127, 137, 139, 162 |
- empty action                     | 6.11 79, 91 |
- end bit                          | 35  |
- end value                        | 6.5.2 83, 84 |
- end-of-line                      | 9  |
- event length                     | 25  |
- event location                   | 25  |
- event list                       | 25  |
- event mode                       | 25  |
- event mode name                  | 25, 98 |
- exception list                   | 3.8 23, 24, 121, 129, 130, 139, 142 |
<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit action</td>
<td>11, 24, 91</td>
</tr>
<tr>
<td>exponent</td>
<td>79, 86</td>
</tr>
<tr>
<td>exponent width</td>
<td>58</td>
</tr>
<tr>
<td>exponentiation operator</td>
<td>76, 77</td>
</tr>
<tr>
<td>expression</td>
<td>51, 52, 61, 68, 70, 71, 79, 98, 107, 110</td>
</tr>
<tr>
<td>expression conversion</td>
<td>55, 68</td>
</tr>
<tr>
<td>expression list</td>
<td>51, 66, 98</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>field</td>
<td>32</td>
</tr>
<tr>
<td>field layout</td>
<td>32, 35</td>
</tr>
<tr>
<td>field name</td>
<td>10, 53, 62, 67, 164</td>
</tr>
<tr>
<td>field name defining occurrence</td>
<td>10</td>
</tr>
<tr>
<td>field name defining occurrence list</td>
<td>10, 32</td>
</tr>
<tr>
<td>field name list</td>
<td>62</td>
</tr>
<tr>
<td>first element</td>
<td>52, 66</td>
</tr>
<tr>
<td>fixed field</td>
<td>32</td>
</tr>
<tr>
<td>floating point range</td>
<td>97, 98, 113</td>
</tr>
<tr>
<td>floating point expression</td>
<td>97, 98, 113</td>
</tr>
<tr>
<td>floating point location</td>
<td>98, 113</td>
</tr>
<tr>
<td>floating point mode</td>
<td>20</td>
</tr>
<tr>
<td>floating point mode name</td>
<td>20, 21</td>
</tr>
<tr>
<td>floating point range mode</td>
<td>20, 21</td>
</tr>
<tr>
<td>floating point range mode name</td>
<td>21</td>
</tr>
<tr>
<td>for control</td>
<td>83</td>
</tr>
<tr>
<td>forbid clause</td>
<td>12.2.3.4</td>
</tr>
<tr>
<td>forbid name list</td>
<td>164</td>
</tr>
<tr>
<td>formal generic mode</td>
<td>142</td>
</tr>
<tr>
<td>formal generic mode indication</td>
<td>16, 142</td>
</tr>
<tr>
<td>formal generic mode list</td>
<td>142</td>
</tr>
<tr>
<td>formal generic parameter</td>
<td>142</td>
</tr>
<tr>
<td>formal generic parameter list</td>
<td>142</td>
</tr>
<tr>
<td>formal generic procedure spec</td>
<td>142</td>
</tr>
<tr>
<td>formal generic synonym</td>
<td>142</td>
</tr>
<tr>
<td>formal generic synonym list</td>
<td>142</td>
</tr>
<tr>
<td>formal parameter</td>
<td>130</td>
</tr>
<tr>
<td>formal parameter list</td>
<td>10.4</td>
</tr>
<tr>
<td>format argument</td>
<td>129, 130, 134, 142, 165</td>
</tr>
<tr>
<td>format clause</td>
<td>114</td>
</tr>
<tr>
<td>format control string</td>
<td>114</td>
</tr>
<tr>
<td>format element</td>
<td>114</td>
</tr>
<tr>
<td>format specification</td>
<td>114</td>
</tr>
<tr>
<td>format text</td>
<td>114</td>
</tr>
<tr>
<td>fractional width</td>
<td>115</td>
</tr>
<tr>
<td>free reference mode</td>
<td>22, 23</td>
</tr>
<tr>
<td>free reference mode name</td>
<td>23</td>
</tr>
<tr>
<td>free reference primitive value</td>
<td>49</td>
</tr>
<tr>
<td>friend clause</td>
<td>12.2.3.4</td>
</tr>
<tr>
<td>friend name</td>
<td>164</td>
</tr>
<tr>
<td>friend name list</td>
<td>164</td>
</tr>
<tr>
<td>friend procedure or process name</td>
<td>164</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>General Procedure Name</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>general procedure name</td>
<td>56</td>
</tr>
<tr>
<td>generality</td>
<td>130</td>
</tr>
<tr>
<td>generic interface mode template</td>
<td>10.11</td>
</tr>
<tr>
<td>generic module instantiation</td>
<td>141, 142</td>
</tr>
<tr>
<td>generic module mode template</td>
<td>134, 142</td>
</tr>
<tr>
<td>generic module name</td>
<td>141</td>
</tr>
</tbody>
</table>

**ISO-IEC 9496 : 2002(E)**

**ITU-T Rec. Z.200 (1999 E)**
generic module template 10.11
generic moreta mode instantiation 10.11
generic moreta mode name 142
generic part 141, 142
generic procedure instantiation 10.11
generic procedure name 142
generic procedure template 141
generic process instantiation 10.11
generic process name 142
generic process template 141
generic region instantiation 10.11
generic region mode template 141
generic region name 142
generic region template 141
generic task mode template 10.11
gettext built-in routine call 7.5.8
goto action 6.9
grant postfix 162, 163
grant statement 12.2.3.4
grant window 163
guarded procedure attribute list 130
guarded procedure definition statement 10.4
guarded procedure signature statement 10.4

H
handler 8.2
hexadecimal bit string literal 61
hexadecimal digit 57, 61
hexadecimal integer literal 57
hour expression 124
hour location 125

I
if action 6.3
implementation clause 39, 41, 43
implementation directive 10
index expression 107, 110, 112
index mode 26, 27, 30
initialization 45
inline component procedure attribute list 130
inline guarded procedure definition statement 39
input-output mode 3.11.1
instance location 92, 95, 96
instance mode 3.9
instance mode name 24
instance primitive value 93
integer expression 50, 83, 98, 120, 124
integer literal expression 18, 19, 21, 25, 27, 29, 35, 59, 77, 92
integer location 125
integer mode 17
integer mode name 17
interface component 3.15.2
interface inheritance 44, 45
interface mode 3.15.2
interface mode name 39, 44, 142
invariant part 39, 41, 43
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
<th>Line Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>io clause</td>
<td>7.5.7</td>
<td>115, 119</td>
</tr>
<tr>
<td>io code</td>
<td></td>
<td>119</td>
</tr>
<tr>
<td>io list</td>
<td>7.5.3</td>
<td>112, 113</td>
</tr>
<tr>
<td>io list element</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>io location built-in routine call</td>
<td>7.4.1</td>
<td>97, 105</td>
</tr>
<tr>
<td>io simple built-in routine call</td>
<td>7.4.1</td>
<td>97, 105</td>
</tr>
<tr>
<td>io value built-in routine call</td>
<td>7.4.1</td>
<td>97, 105</td>
</tr>
<tr>
<td>irrelevant</td>
<td></td>
<td>167</td>
</tr>
<tr>
<td>isassociated built-in routine call</td>
<td></td>
<td>105</td>
</tr>
<tr>
<td>iteration</td>
<td></td>
<td>83</td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>label name</td>
<td></td>
<td>86, 90</td>
</tr>
<tr>
<td>labelled array tuple</td>
<td>5.2.5</td>
<td>61, 62</td>
</tr>
<tr>
<td>labelled structure tuple</td>
<td></td>
<td>62</td>
</tr>
<tr>
<td>left element</td>
<td></td>
<td>50, 65</td>
</tr>
<tr>
<td>length</td>
<td></td>
<td>35</td>
</tr>
<tr>
<td>length argument</td>
<td>6.20.3</td>
<td>97, 98</td>
</tr>
<tr>
<td>letter</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>lifetime-bound initialization</td>
<td>4.1.2</td>
<td>45, 46</td>
</tr>
<tr>
<td>line-end comment</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>literal</td>
<td></td>
<td>55</td>
</tr>
<tr>
<td>literal expression</td>
<td></td>
<td>139</td>
</tr>
<tr>
<td>literal expression list</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>literal range</td>
<td></td>
<td>19, 26, 167</td>
</tr>
<tr>
<td>location</td>
<td></td>
<td>47, 78, 79, 87, 90, 95, 96, 97, 105, 107</td>
</tr>
<tr>
<td>location argument</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>location built-in routine call</td>
<td>4.2.12</td>
<td>47, 53</td>
</tr>
<tr>
<td>location contents</td>
<td></td>
<td>55</td>
</tr>
<tr>
<td>location conversion</td>
<td>4.2.13</td>
<td>47, 54</td>
</tr>
<tr>
<td>location declaration</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>location do-with name</td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>location enumeration</td>
<td>6.5.2</td>
<td>83, 84</td>
</tr>
<tr>
<td>location enumeration name</td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>location list</td>
<td></td>
<td>95</td>
</tr>
<tr>
<td>location name</td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>location procedure call</td>
<td>4.2.11</td>
<td>47, 53</td>
</tr>
<tr>
<td>loc-identity declaration</td>
<td>4.1.3</td>
<td>45, 47</td>
</tr>
<tr>
<td>loc-identity name</td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>loop counter</td>
<td></td>
<td>83, 84</td>
</tr>
<tr>
<td>lower bound</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>lower element</td>
<td></td>
<td>52, 66</td>
</tr>
<tr>
<td>lower float bound</td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>member mode</td>
<td></td>
<td>22</td>
</tr>
<tr>
<td>membership operator</td>
<td></td>
<td>73</td>
</tr>
<tr>
<td>minute expression</td>
<td></td>
<td>124</td>
</tr>
<tr>
<td>minute location</td>
<td></td>
<td>125</td>
</tr>
<tr>
<td>mode</td>
<td>3.3</td>
<td>13, 16, 22, 24, 25, 26, 30, 32, 45, 47, 54, 139, 140, 142, 148</td>
</tr>
<tr>
<td>mode argument</td>
<td>6.20.3</td>
<td>97, 98, 101</td>
</tr>
<tr>
<td>mode definition</td>
<td>3.2.1</td>
<td>13, 14, 15</td>
</tr>
<tr>
<td>mode name</td>
<td></td>
<td>48, 49, 54, 59, 61, 68, 98</td>
</tr>
<tr>
<td>modification built-in routine call</td>
<td>7.4.5</td>
<td>105, 106</td>
</tr>
<tr>
<td>modify parameter</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>modify parameter list</td>
<td>7.4.5</td>
<td>106, 107</td>
</tr>
<tr>
<td>module</td>
<td>10.6</td>
<td>79, 134, 135</td>
</tr>
<tr>
<td>module body</td>
<td>10.2</td>
<td>127, 134, 141</td>
</tr>
<tr>
<td>module body component</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>module inheritance</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>module inheritance clause</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>Term</td>
<td>Page(s)</td>
<td>Page(s)</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>module mode</td>
<td>3.15.2</td>
<td>38, 39</td>
</tr>
<tr>
<td>module mode body</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>module mode name</td>
<td>39, 41, 43</td>
<td></td>
</tr>
<tr>
<td>module mode specification</td>
<td>39, 141</td>
<td></td>
</tr>
<tr>
<td>module spec</td>
<td>138</td>
<td></td>
</tr>
<tr>
<td>module specification component</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>modulion name</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>modulion or moreta mode name</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>monadic operator</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>month expression</td>
<td>124</td>
<td></td>
</tr>
<tr>
<td>month location</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>moreta component name</td>
<td>2.7</td>
<td>10, 11</td>
</tr>
<tr>
<td>moreta component procedure call</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>moreta declaration statement</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>moreta location</td>
<td>11, 87</td>
<td></td>
</tr>
<tr>
<td>moreta mode</td>
<td>3.15.1</td>
<td>29, 38</td>
</tr>
<tr>
<td>moreta mode name</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>moreta mode name</td>
<td>38, 142</td>
<td></td>
</tr>
<tr>
<td>moreta newmode definition statement</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>moreta synmode definition statement</td>
<td>135</td>
<td></td>
</tr>
<tr>
<td>moreta-bound initialization</td>
<td>4.1.2</td>
<td>45, 46</td>
</tr>
<tr>
<td>multiple assignment action</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>2.7</td>
<td>10</td>
</tr>
<tr>
<td>name string</td>
<td>10, 163, 165</td>
<td></td>
</tr>
<tr>
<td>narrow character literal</td>
<td>5.2.4.5</td>
<td>59</td>
</tr>
<tr>
<td>narrow character string literal</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>narrow text mode</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>new prefix</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>newmode definition statement</td>
<td>3.2.3</td>
<td>15, 39, 127, 139, 143</td>
</tr>
<tr>
<td>newmode name string</td>
<td>163</td>
<td></td>
</tr>
<tr>
<td>non-composite mode</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>non-percent character</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>non-reserved character</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>non-reserved name</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>non-reserved wide character</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>non-special character</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>numbered set element</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>numbered set list</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>numeric expression</td>
<td>6.20.3</td>
<td>97, 98</td>
</tr>
<tr>
<td>octal bit string literal</td>
<td></td>
<td>61</td>
</tr>
<tr>
<td>octal digit</td>
<td>57, 61</td>
<td></td>
</tr>
<tr>
<td>octal integer literal</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>old prefix</td>
<td>162</td>
<td></td>
</tr>
<tr>
<td>on-alternative</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>operand–0</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>operand–1</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>operand–2</td>
<td>5.3.5</td>
<td>72, 73</td>
</tr>
<tr>
<td>operand–3</td>
<td>5.3.6</td>
<td>73, 74</td>
</tr>
<tr>
<td>operand–4</td>
<td>5.3.7</td>
<td>74, 75</td>
</tr>
<tr>
<td>operand–5</td>
<td>5.3.8</td>
<td>75, 76</td>
</tr>
<tr>
<td>operand–6</td>
<td>5.3.9</td>
<td>76, 77</td>
</tr>
<tr>
<td>operand–7</td>
<td>5.3.10</td>
<td>77, 78</td>
</tr>
<tr>
<td>operator–3</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>operator–4</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>origin array mode name</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>origin string mode name</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>origin variant structure mode name</td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>
P

parameter attribute  24
parameter list  130, 163
parameter list  3.8  23, 24, 164
parameter spec  24, 130, 139
parameterized array mode  30
parameterized array mode name  30
parameterized string mode  29
parameterized string mode name  29
parameterized structure mode  32
parameterized structure mode name  32
parenthesized clause  7.5.4  114, 115
parenthesized expression  5.2.17  55, 70
percent  114
piece designator  136, 137
pos  35
postfix  162
powerset difference operator  74, 80
powerset enumeration  6.5.2  83, 84
powerset expression  84, 97
powerset inclusion operator  73
powerset mode  3.6  16, 22
powerset mode name  22
powerset tuple  61
predefined moreta location  4.2.14  47, 54
prefix  10, 162, 163, 165
prefix clause  163, 165
prefix rename clause  12.2.3.3
prefixed name string  10, 11
primitive value  5.2.1  55, 78
priority  87, 92, 93, 94
proc body  10.2  127, 129, 130
procedure attribute list  10.4  129, 130, 139
procedure call  87
procedure definition  129, 141
procedure definition statement  10.4  127, 129
procedure mode  3.8  16, 23
procedure mode name  23
procedure name  87, 143, 164
procedure primitive value  87
process body  10.2  127, 134
process definition  134, 141
process definition statement  10.5  39, 127, 134
process name  69, 148, 164
process specification statement  39, 45
program  10.8  135

Q

qualified name  2.7  10, 11
quasi data statement  10.10.3  127, 139
quasi declaration  139
quasi declaration statement  139
quasi definition statement  139
quasi formal parameter  139
quasi formal parameter list  139
quasi location declaration  139
quasi loc-identity declaration  139
quasi procedure definition statement  139
quasi process definition statement  139
quasi signal definition  140
quasi signal definition statement  10.10.3  139, 140
quasi synonym definition 139
quasi synonym definition statement 139
quote 60

R
range 61
range enumeration 6.5.2 83, 84
range list 6.4 81, 82
reach-bound initialization 4.1.2 45, 46
readrecord built-in routine call 7.4.9 105, 110
real mode 3.5 16, 20
receive buffer case action 6.19.3 94, 96
receive case action 6.19.1 79, 94
receive signal case action 6.19.2 94, 95
record mode 26
reference mode 3.7.1 16, 22
reference primitive value 101
referenced location 78
referenced mode 22
region 10.7 127, 135
region body 10.2 127, 135, 141
region body component 41, 43
region inheritance 41
region inheritance clause 41
region mode 3.15.3 38, 41
region mode body 41
region mode name 41
region mode specification 41, 141
region spec 138
region specification component 41, 43
relational operator 73
relative timing action 122
remote context 10.10.1 136, 138
remote modulion 10.10.1 134, 135, 136
remote program unit 10.10.1 14, 15, 137, 141
remote spec 10.10.1 136, 138
repetition factor 114
representation conversion 5.2.12 55, 68
result 90
result action 6.8 79, 90
result attribute 24
result spec 130
result spec 3.8 23, 24, 129, 130, 139, 142, 163, 164, 165
return action 6.8 79, 90
right element 50, 65
row mode 3.7.4 22, 23
row mode name 23
row primitive value 49

S
second expression 124
second location 125
seize postfix 162, 165
seize statement 12.2.3.5 39, 142, 162, 165
seize window 165
send action 6.18.1 79, 93
send buffer action 6.18.3 93, 94
send signal action 93
set element 18
set element name 2.7 10, 59
set element name defining occurrence 2.7 10, 18
set list 18
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>set literal</td>
<td>56</td>
</tr>
<tr>
<td>set mode</td>
<td>3.4.5, 17, 18</td>
</tr>
<tr>
<td>set mode name</td>
<td>18</td>
</tr>
<tr>
<td>settext built-in routine call</td>
<td>7.5.8, 105, 120</td>
</tr>
<tr>
<td>signal definition</td>
<td>148</td>
</tr>
<tr>
<td>signal definition statement</td>
<td>11.5, 39, 41, 45, 127, 139, 148</td>
</tr>
<tr>
<td>signal name</td>
<td>93, 95, 95</td>
</tr>
<tr>
<td>signal receive alternative</td>
<td>58, 77, 58</td>
</tr>
<tr>
<td>signed floating point literal</td>
<td>57, 77</td>
</tr>
<tr>
<td>signed integer literal</td>
<td>21</td>
</tr>
<tr>
<td>significant digits</td>
<td>21</td>
</tr>
<tr>
<td>simple component procedure attribute list</td>
<td>130</td>
</tr>
<tr>
<td>simple guarded procedure definition statement</td>
<td>39, 41</td>
</tr>
<tr>
<td>simple guarded procedure signature statement</td>
<td>39, 41, 45</td>
</tr>
<tr>
<td>simple name string</td>
<td>8, 10, 11, 39, 41, 43, 44, 79, 129, 130, 134, 135, 136, 137, 138, 139, 141, 142</td>
</tr>
<tr>
<td>simple prefix</td>
<td>10</td>
</tr>
<tr>
<td>simple spec module</td>
<td>138</td>
</tr>
<tr>
<td>simple spec region</td>
<td>138</td>
</tr>
<tr>
<td>single assignment action</td>
<td>79</td>
</tr>
<tr>
<td>slice size</td>
<td>50, 52, 65, 66</td>
</tr>
<tr>
<td>spec module</td>
<td>10.10.2, 79, 127, 135, 138</td>
</tr>
<tr>
<td>spec module body</td>
<td>10.2, 127, 138</td>
</tr>
<tr>
<td>spec region</td>
<td>10.10.2, 127, 135, 138</td>
</tr>
<tr>
<td>spec region body</td>
<td>10.2, 127, 138</td>
</tr>
<tr>
<td>start action</td>
<td>6.13, 79, 91</td>
</tr>
<tr>
<td>start bit</td>
<td>35</td>
</tr>
<tr>
<td>start element</td>
<td>50, 65, 50</td>
</tr>
<tr>
<td>start expression</td>
<td>5.2.15, 55, 69, 91</td>
</tr>
<tr>
<td>start value</td>
<td>83</td>
</tr>
<tr>
<td>static mode location</td>
<td>35</td>
</tr>
<tr>
<td>step</td>
<td>83</td>
</tr>
<tr>
<td>step size</td>
<td>35</td>
</tr>
<tr>
<td>step value</td>
<td>83</td>
</tr>
<tr>
<td>stop action</td>
<td>6.14, 79, 91</td>
</tr>
<tr>
<td>store location</td>
<td>110</td>
</tr>
<tr>
<td>string concatenation operator</td>
<td>4.2.6, 74, 80</td>
</tr>
<tr>
<td>string element</td>
<td>47, 50</td>
</tr>
<tr>
<td>string expression</td>
<td>84, 98, 113</td>
</tr>
<tr>
<td>string length</td>
<td>29</td>
</tr>
<tr>
<td>string location</td>
<td>50, 84, 98, 113</td>
</tr>
<tr>
<td>string mode</td>
<td>23, 29, 23</td>
</tr>
<tr>
<td>string mode name</td>
<td>29, 98, 29</td>
</tr>
<tr>
<td>string primitive value</td>
<td>65</td>
</tr>
<tr>
<td>string repetition operator</td>
<td>77</td>
</tr>
<tr>
<td>string slice</td>
<td>4.2.7, 47, 50</td>
</tr>
<tr>
<td>string type</td>
<td>29</td>
</tr>
<tr>
<td>structure field</td>
<td>4.2.10, 47, 53</td>
</tr>
<tr>
<td>structure location</td>
<td>53, 86, 53</td>
</tr>
<tr>
<td>structure mode</td>
<td>3.13.4, 29, 32</td>
</tr>
<tr>
<td>structure mode name</td>
<td>32</td>
</tr>
<tr>
<td>structure primitive value</td>
<td>67, 86, 67</td>
</tr>
<tr>
<td>structure tuple</td>
<td>5.2.5, 61, 62</td>
</tr>
<tr>
<td>sub expression</td>
<td>71</td>
</tr>
<tr>
<td>sub operand–0</td>
<td>72</td>
</tr>
<tr>
<td>sub operand–1</td>
<td>72</td>
</tr>
<tr>
<td>sub operand–2</td>
<td>73</td>
</tr>
<tr>
<td>sub operand–3</td>
<td>74</td>
</tr>
<tr>
<td>sub operand–4</td>
<td>75</td>
</tr>
<tr>
<td><strong>ISO-IEC 9496 : 2002(E)</strong></td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>sub operand–5</strong></td>
<td>76</td>
</tr>
<tr>
<td><strong>synchronization mode</strong></td>
<td>3.10.1</td>
</tr>
<tr>
<td><strong>synmode definition statement</strong></td>
<td>3.2.2</td>
</tr>
<tr>
<td><strong>synonym definition</strong></td>
<td>54</td>
</tr>
<tr>
<td><strong>synonym definition statement</strong></td>
<td>5.1</td>
</tr>
</tbody>
</table>

### T
- **tag field name**: 32
- **tag list**: 32
- **task body component**: 43
- **task inheritance**: 43
- **task inheritance clause**: 43
- **task mode**: 3.15.4 | 38, **43** |
- **task mode body**: 43
- **task mode name**: 43
- **task mode specification**: 43, 142
- **task specification component**: 43
- **template**: 10.11 | 127, 135, **141** |
- **terminate built-in routine call**: 6.20.4 | 97, **101** |
- **text argument**: 7.5.3 | 112, **113** |
- **text built-in routine call**: 7.5.3 | 105, **112** |
- **text io argument list**: 112
- **text length**: 27
- **text location**: 98, 107, 113, 120
- **text mode**: 3.11.4 | 26, **27** |
- **text mode name**: 98
- **text reference name**: 2.7 | 11, 137 |
- **then alternative**: 71
- **then clause**: 81
- **time value built-in routine call**: 9.4 | 97, **124** |
- **timing action**: 9.3 | 79, **122** |
- **timing handler**: 122, 123
- **timing mode**: 3.12.1 | 16, **28** |
- **timing simple built-in routine call**: 9.4.3 | 97, **125** |
- **transfer location**: 107, 109
- **tuple**: 5.2.5 | 55, **61** |

### U
- **undefined synonym name**: 70
- **undefined value**: 70
- **unlabelled array tuple**: 61
- **unlabelled structure tuple**: 62
- **unnumbered set list**: 18
- **unsigned floating point literal**: 5.2.4.3 | **58** |
- **unsigned integer literal**: 5.2.4.2 | **57** |
- **upper bound**: 19
- **upper element**: 52, 66
- **upper float bound**: 21
- **upper index**: 30
- **upper lower argument**: 6.20.3 | 97, **98** |
- **usage expression**: 107

### V
- **value**: 5.3.1 | 46, 61, 62, **70**, 79, 87, 90, 93, 94, 101, 105, 107
- **value argument**: 113
- **value array element**: 5.2.8 | 55, **66** |
- **value array slice**: 5.2.9 | 55, **66** |
- **value built-in routine call**: 5.2.14 | 55, **69** |
- **value case alternative**: 71
- **value do-with name**: 56
value enumeration 83
value enumeration name 56
value name 55
value procedure call 5.2.13 55, 69
value receive name 56
value string element 5.2.6 55, 65
value string slice 5.2.7 55, 65
value structure field 5.2.10 55, 67
variant alternative 32
variant field 32
variant structure mode 23
variant structure mode name 32, 98
visibility statement 12.2.3.2 127, 162

W
where expression 107
while control 6.5.3 83, 86
wide character 9
wide character literal 59
wide character string literal 60
wide text mode 27
with control 86
with part 6.5.4 83, 86
word 35
write expression 110
writerecord built-in routine call 7.4.9 105, 110

Y
year expression 124
year location 125

Z
zero-adic operator 5.2.16 55, 70
<table>
<thead>
<tr>
<th>Series</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Organization of the work of the ITU-T</td>
</tr>
<tr>
<td>B</td>
<td>Means of expression: definitions, symbols, classification</td>
</tr>
<tr>
<td>C</td>
<td>General telecommunication statistics</td>
</tr>
<tr>
<td>D</td>
<td>General tariff principles</td>
</tr>
<tr>
<td>E</td>
<td>Overall network operation, telephone service, service operation and human factors</td>
</tr>
<tr>
<td>F</td>
<td>Non-telephone telecommunication services</td>
</tr>
<tr>
<td>G</td>
<td>Transmission systems and media, digital systems and networks</td>
</tr>
<tr>
<td>H</td>
<td>Audiovisual and multimedia systems</td>
</tr>
<tr>
<td>I</td>
<td>Integrated services digital network</td>
</tr>
<tr>
<td>J</td>
<td>Transmission of television, sound programme and other multimedia signals</td>
</tr>
<tr>
<td>K</td>
<td>Protection against interference</td>
</tr>
<tr>
<td>L</td>
<td>Construction, installation and protection of cables and other elements of outside plant</td>
</tr>
<tr>
<td>M</td>
<td>TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits</td>
</tr>
<tr>
<td>N</td>
<td>Maintenance: international sound programme and television transmission circuits</td>
</tr>
<tr>
<td>O</td>
<td>Specifications of measuring equipment</td>
</tr>
<tr>
<td>P</td>
<td>Telephone transmission quality, telephone installations, local line networks</td>
</tr>
<tr>
<td>Q</td>
<td>Switching and signalling</td>
</tr>
<tr>
<td>R</td>
<td>Telegraph transmission</td>
</tr>
<tr>
<td>S</td>
<td>Telegraph services terminal equipment</td>
</tr>
<tr>
<td>T</td>
<td>Terminals for telematic services</td>
</tr>
<tr>
<td>U</td>
<td>Telegraph switching</td>
</tr>
<tr>
<td>V</td>
<td>Data communication over the telephone network</td>
</tr>
<tr>
<td>X</td>
<td>Data networks and open system communications</td>
</tr>
<tr>
<td>Y</td>
<td>Global information infrastructure and Internet protocol aspects</td>
</tr>
<tr>
<td>Z</td>
<td><strong>Languages and general software aspects for telecommunication systems</strong></td>
</tr>
</tbody>
</table>